Lesson 1, Week 2: Arithmetic

Although, as we have said, this course is about text, not about numbers, some competence with arithmetic is essential in almost all programming projects.

AIM

— To describe and illustrate Julia's rules for arithmetic

After this lesson, you will be able to

* name and type the five arithmetic operators
* use parentheses to ensure that an arithmetic expression is not ambiguous
* combine values, names, operators and delimiters to make valid arithmetical expressions

REPL examples

In the REPL, one can just enter a valid arithmetic expression and its value is returned, e.g. `1+1`, `1 / 2 + 1`, `(5 - (25 - 9)^(1/2) )/2`.
(The latter can even be typed using the $\sqrt{}$ character)

[DEMO these expressions in the REPL, and many more.]

1 The arithmetic operators in Julia

Basic arithmetic in Julia uses the operators `+ - * /` (addition, subtraction, multiplication, division) and `^` (raising to a power).

Only left-right pairs of parentheses (round brackets) may be used as delimiters. In written and typeset arithmetic, other kinds of brackets often appear, but in Julia they throw errors inside arithmetic expressions.

Note that in Julia, the expression `-4` is meaningful[1].

[1] And so is `+4`, but of course we are not that interested in it.

2 How to ensure your arithmetic expression does what you want it to do

There is full set of rules in Julia, called "Operator precedence and associativity" that give a unique interpretation to the arithmetic of a given sequence of numbers and arithmetic operators. I recommend that you DON'T try to learn them at this stage. Instead, use brackets (round ones) to ensure that your arithmetic expression means what you want it to mean.

When an arithmetic expression is very long, making sure it has unique meaning requires a lot of brackets. For this reason, most programmers make frequent use of the rules of operator precedence to avoid using parentheses. However, on this course we don't use much arithmetic and when we do, the expressions are short. For that reason, we will insert many pairs of round brackets instead. In any case the rules are quite hard to learn and even experts make mistakes with long expressions. As a beginner, keep your arithmetic expressions short and use lots of brackets to make sure the rules are not needed.

The game: how many values can a sequence of numbers have?

Given a sequence of numbers like `1 4`, how many different arithmetic values can one create by inserting operators and delimiters?

For these two numbers, and no sign changes, the operators `+ - * /` create the values `5 -3 4` and `0.25`. Judicious use of minus symbols make it possible to create values with the opposite sign. That's eight, and `1^4` gives `1`, while `-(1^4)` gives -1. Ten values in all.

Does the sequence `4 1` give the same set of values? How does `2 3` compare?

Finally, try how many values you can get from the sequence `2 3 5` — note that every expression will have to use at least one pair of parentheses to ensure unique interpretation.

Using variable names to supply the values in an arithmetical expression

You do this just as you do in mathematics. For example, the expression `(2-3)*4` is equivalent to the lines

```
a, b, c = 2, 3, 4
(a-b)*c
```

There is more: the expression `2*a` can be replaced by `2a`. Similarly, `4*(2-3)` is equivalent to `4(2-3)`. The rule is: a number followed without whitespace by a name or by the start of parenthesis is interpreted as a multiplier—that is, a the multiply operator is inserted between the number and what follows it. Here's an example: the famous quadratic formula for the solution to $ax^2 + bx + c = 0$ gives two formulae for the roots, namely $x = (-b + (b^2 - 4ac)^{1/2})/2a$ and $x = (-b - (b^2 - 4ac)^{1/2})/2a$.

In Julia we can write them as
`root1 = (-b + (b^2 - 4a*c)^(1/2) )/2a` and
`root2 = (-b - (b^2 - 4a*c)^(1/2) )/2a` ;
note that the multiply operator is still necessary between `a` and `c`.

Review and summary

* Arithmetic expressions combine numbers and operators

* The arithmetic operators are `+` for adding, `-` for subtrating, `*` for multiplying, `/` for dividing and `^` for raising to a power.

* Arithmetic expressions are formed like in standard maths: start with a number, end with a number, with exactly one operator between adjacent numbers[2]

* Expressions with many numbers and operators can be ambiguous. Err on the side of caution: add enough `( )` pairs to ensure that only one interpretation of the computation is possible.

[2]For completeness: forms like `-7` use only one number, on the right of the operator.

Lesson 2, Week 2: Number Types

AIM

— To describe the main number types of Julia

After this lesson, you will be able to

* Motivate why we do number types very early in this course
* Describe the types `Int64` and `Float64`
* Use `bitstring` to display the bits pattern of a number
* Show that a floating point number equal to an integer has a different bits pattern
* Briefly describe integer and floating point types that use fewer than 64 bits per number

Why mention number types so early in the course?

Reason 1: many error messages mention number types.
Efficient debugging relies on understanding them well enough to decide whether the problem really is with the type a numerical value has.
[DEMO: indexing with the value `1.0`; the decimal point makes all the difference!]

Reason 2: they make a good introduction to the Julia type system.

The `Int64` type

You've seen that a character has variable width of one or more code units. By contrast, number values of a given type have the same width. It is measured in bits[1] and an `Int64` is an integer value that occupies 64 bits.

If you simply enter an integer you get a 64-bit value.

DEMO: `[1, 2]`, `bitstring(1)`
We explain the function `bitstring` in more detail after the examples.

[1] A bit has the value 0 or the value 1.

Let's show the raw bits of a few more 64-bit integers. [DEMO: 0, 7 and 13. And -0, -7 and -13]

`Int64` represents all integers from -9223372036854775808 to 9223372036854775807 but none outside that range[2].

The function `bitstring` returns, if possible, the bits representation of its argument. Note that this includes character values[3].

[DEMO: explain the result of `[bitstring(x) for x in "aα±"]`.]

Again, we emphasise that these examples rely crucially on the absence of decimal points, because that is how Julia knows to use the `Int64` type.

Floating point numbers: `Float64`

These numbers are very different from the signed integers above: we use them to approximate all numbers in a range, whether whole numbers, rational numbers, or irrational numbers. They can go much larger than `Int64` values, and they can represent fractions as well as integers[4]. Note the presence of the decimal in all the floating point values we create below.

[DEMO bitstrings of 1.0, 0.1, 1.1 and their negative counterparts]

As you can see, `Float64` also has a width of 64 bits, but uses a much more complicated arrangement of bits than one sees in `Int64`. The details of this difference do not matter on this course, only the implication at machine level. Without going into electronic detail, it is reasonably obvious that the actual manipulation of bits for adding two `Int64` numbers will be very different from adding two `Float64` numbers. Moreover, it is simply not possible to add a `Float64` to a `Int64`, as the result must be one of the two, it can't be a hybrid.

[DEMO: arithmetic with mixed types]

We see that in all cases, Julia gives the result with type `Float64`. This is because all `Int64` values can be fairly accurately represented by `Float64` values, but not the other way round.

Besides representing quite a large range of numbers, `Float64` helps in other ways. The biggest actual positive number it can represent is near $1.7 * 10^{308}$, much larger than the largest possible `Int64`. Something perhaps unexpected happens for numbers that go bigger:

[DEMO: `1.7*(10.0^308)`, `1.8*(10.0^308)`, `(1.7*(10.0^308)) * 2` — note ALL decimal points!]

The value `Inf` is mathematically very interesting, but we don't pursue it. A similarly interesting

[2] An easier way to think of this interval is that it is approximately $[-9 * 10^{18}, 9 * 10^{18}]$

[3] Note that the bitstring of every character contains 32 bits. Does this mean fixed width? No, not when characters combine to form a string. This is because the bitstring of a character contains information about how many code units it contains. Everything outside these code units are zeros anyway and are discarded when the characters are joined to form a string.

[4] Not all of them exactly, though

value is `NaN`, which represents the indeterminate result of attempted calculations like `0.0/0.0` and `Inf * 0.0` . These values will occasionally crop up not only in your results but also in your error messages.

There is a very convenient shorthand for for numbers of the form `1.7*(10.0^308)`: we can simply write `1.7e308`. The error message for `1.8e308` is new, and note that the number following the `e` need not be an integer[5].

Number types with reduced width: faster computation but less accurate

On most modern systems, numbers by default use 64 bits, and so if you enter an integer it is by default of type `Int64` and a float by default is of type `Int64`. Using so many bits makes them very powerful in many ways, but it can be more efficient to use shorter bitstrings. Julia makes available types like `Int8` and `Float32`, which use 8 and 32 bits, respectively. The tradeoff works as follows: using 8 bits instead of 64 means you need only 1/8th of the capacity. However, and this is the reason modern systems have gone on to 64 bit defaults, you have many fewer numbers available. For example, the type `Int8` consists of the integers from -128 to 127, and no others.

To make sure a number is of a given type, you can call the type as if it is a function: `Int32(13)`, `Int16(13)`, `Int8(13)`.
But you won't see any difference until you use them, `bitstring` being a reliable indicator:

[DEMO of this fact and a few more illustrations]

Other number types

Julia has a great many more number types, including complex and exact rational numbers. It is not necessary for a beginner to master them.

Review and summary

* Julia's main number types are `Int64` integers and `Float64` floating point numbers
* The `Int64` integers range from approximately $-9 * 10^{18}$ to $9 * 10^{18}$
* The numerical values of `Float64` range from approximately $-1.7 * 10^{308}$ to $1.7 * 10^{308}$
* `Float64` numbers can approximate fractions
* Arithmetic that uses both `Float64` and `Int64` numbers always result in `Float64` number
* `Float64` values include `Inf` and `NaN`, which do not represent actual numbers

[5] And it can be negative, of course.

Lesson 3, Week 2: Functions II (user-defined functions)

AIM

— To teach the basics of user-defined functions in Julia

After this lesson, you will be able to

* Explain how to use the reserved keywords `function` and `end` to create a Julia function
* Specify what names are acceptable for user-defined function, and how to specify the argument list in a user-defined function
* Explain how to use the optional keyword `return` to specify the value a user-defined function returns
* Explain how to write an inline function in Julia, omitting all reserved keywords

Functions are hugely important in Julia: they are the main way to organise your code into short bits, they are the main way Julia achieves its spectacular speed of computation and they are also the main reason why large computational projects can be very successful in Julia[1].

The basic `function() ... end` syntax

We start with an example of making a function out of code you've seen before, for reversing a string:

```
str = "abcd"
str[end:-1:1]
```

Recall that the range `end:-1:1` takes steps of length -1 to go from the index `end` to the index `1`.

There are several ways to make functions. The most flexible way uses the reserved keyword `function`.

```
function spin(str)
    y = str[end:-1:1]
    return y
end
```

1. After `function` must come name followed by round brackets (no whitespace), inside the brackets is the *argument list*.
2. The `end` keyword is essential to close the code block opened by `function`

[1] This course is not about Julia's strengths, so we don't discuss these topics any further.

Here we use `return` keyword to return the value of the variable(s) *after* it is what the function returns (note that if you use more than one variable name, the rule is to use commas to separate them). We deal with omitting it later.

Actually, what `spin` does is available from the built-in function[2] `reverse`, which is faster[3]. But the beauty of Julia is that functions are generic: we can easily extend them with new methods. Let's make a function that reverses only a middle bit of the string.

```
function spin(str,k)
    init = str[1:k]
    finish = str[end-k+1:end]
    mid = str[k+1:end-k]
    y = join([init, spin(mid), finish])
    return y
end
```

DEMO:

Note the message after the function is created : `spin (generic function with 2 methods)`.

This illustrates that in Julia, one function name can be used to name functions that do different things. We say that user-defined functions in Julia are *generic*[4] because means they can always be extended with one more method. Julia determines which method to use for a given call from the argument list in the call.

Acceptable function names

Any string that is a valid variable name is also a valid function name, except reserved keywords, type names and a few others special words in Julia that are not important to know[5].

On this course, we discourage variable names that use exclamation marks and underscores. Although they are valid, Julian style is to use them in special ways that aren't part of this course.

Using a `.jl` file for one or more functions

Of course, if you want to retain the code of a function, you should put it into a `.jl` file, and use `include` to make the function available at the REPL. If you want to put several functions in one `.jl` file, no problem, just do it.

DEMO: both functions in a single file; notice how the name of the file that contains the functions doesn't have to use the function names at all.
Using `include` to make the functions available.

[2]There are very many built-in functions in basic Julia, and many more in the packages. If you continue to code in Julia, you should certainly study and use them.

[3]Speed however is not our aim in this course.

[4]So are almost all Julia functions, and the exceptions are highly technical and far beyond this course.

[5]If you try to use one of them, Julia will give you a clear error message.

Working with functions: testing, debugging

Testing is essential when coding[6].

We should at least test that `spin` does do the reversals we expect—both methods!
We should also check that they fail in the ways we expect.

DEMO in the REPL

Debugging now involves more than just fixing invalid code: we must also fix valid code that does something other than what we want.

Let us change the `.jl` file so that the code is valid but the output is wrong.

DEMO involving both the REPL and the text editor

Omitting the `return` keyword; inline functions

If there is no `return` keyword in your function body, it will still return a value, namely the very last one that was calculated.

[DEMO: modify one or both `spin` functions this way]

This paves the way for *inline functions*: you can omit all the keywords, and just use round brackets after the name, an equals sign, and an expression.

[DEMO: `spin(str)` as an inline function]

This is often very convenient convenient in writing code: instead of one long complicated function, use inline functions for code that occurs often in your function.

There is one more way to write functions: anonymous functions, where you don't even have to use a name. However, this is is mainly for iteration, so we prefer to discuss them in the lecture on iterators[7].

[6] The Julia community has an extremely strong culture of testing and has lots of ways to support and encourage testing.

[7] Of which you've already seen one, namely comprehension.

Review and summary

* You can write a function using the `function() ... end` syntax
* The function body consists of the lines *after* the input argument list and *before* the closing `end`
* The `return` keyword specifies what the function returns; if it is omitted, the function returns the value returned by its final line
* Functions are generic: they can have more than one method; the pattern of input values decides which one is used
* Testing is essential for newly created functions
* A `.jl` file can contain many functions
* With inline functions, you omit the `function` and `end` keywords; it is intended for functions with one-line code bodies

It is essential that you become very comfortable with writing function code in the two ways discussed in this lesson. Please make sure you do the exercises and the self-assessed assignment.

Lesson 4, Week 2: Functions III (generic functions)

AIM

— Illustrate and explain Julia's use of generic functions

After this lesson, you will be able to

* Use the `::` operator to specify acceptable input types for the values of variables in a function call
* Describe what a generic function is and exactly what constitutes the code of a method of a generic function
* Explain what is meant by the type signature of a function method
* Use `methods` to determine the type signatures of all the methods of a function

The concept of multiple dispatch is much easier to understand from a few examples than from a first-principles discussion.

Example: specifying the value type for input to a function

To double a number means to multiply by two, but to double a word means to make a word twice as long[1]. Let's create a function called `double` with two methods, one for each type of input. We will use the types `Int64` and `String`, and the operator `::` to specify which type we mean. Inlining the functions is best here:

```
double(x::Int64) = 2x
double(x::String) = x^2
```

DEMO: show that `double(7)` and `double("7")` work, but `double(7.0)` throws an error.

By putting `x::Int64` in the argument list of `double`, we specify that this function body should be used for that type of variable. In other words, we specify that if the argument passed to `double` is a value of type `Int64`, then multiplying by 2 is the method to use.

[1] As in the South Africanisms "now-now" and "what-what".

What `methods` tell you about a function

The function `methods` is what you use to find out how many methods a function has and how the argument list of each method is specified.

DEMO: interrogate `include`, `join`

Explain why `join(777, 'A', "wins")` doesn't throw an error.

What is the type signature of a function method?

As you know, the argument list in function call specifies a number of values[2]. The *type signature* of the call is the list of types of these values, in order. For example, if we define a function with `eg3(x,y) = x*y` then the calls `eg3(1, 2.0)` and `eg3(1.0, 2)` have different signatures. But we define a function before we call it. So what is the type signature of a function at the time we define it—that is, what is the type signature of a function method, before it is called?

Let us use the `methods` to interrogate `spin` and `double`, which tell us what the type signatures are that were defined for each of their methods.

[DEMO in the REPL]

We see that `double` is very clear: there is a method for a value of type `Int64`, a method for a value of type `String`. For `spin` the anwer is not as definite: it is true that the signatures cannot be confused, because they use different number input values, but it says nothing about their types. What does this mean?

Well, it means they show differences in their error behaviour: [DEMO: `double('x')` and `spin('x')`]

In the case of the call `double('x')` Julia never even starts executing the function, because none of its methods fit. In the case of `spin('x')` Julia passes the value `'x'` to the only method that uses a single value in its type signature. However, the code in that method throws an error when it gets a character instead of a string.

Review and summary

* The type operator :: inside a function argument list is used to specify the type of an input value
* The method of a function is the code in its code body
* A generic function is a function that can have more than one method
* The function `methods` lists all the valid type signatures of a generic function; each signature corresponds to exactly one method.

[2] Yes, some of the items may be variable names, but it is always the actual value of the variable that is used in the call.

Lesson 5, Week 2: Type System (as it relates to multiple dispatch)

AIM

— To describe enough of Julia's type system for explaining how Julia does multiple dispatch

After this lesson, you will be able to

* Describe how Julia's type system consists abstract and concrete types
* Use the `supertype` function on a type to find its supertype
* Use the `subtypes` function on a type to find its subtypes
* Describe the abstract type `Any`
* Explain how multiple dispatch matches the type signature of a function call to the type specification of a method

An illlustrative example

Let's go back to the function `double` defined in the previous lecture, with the two methods
`double(x:Int64) = 2x` and
`double(x::String) = x^2`
and recall that this throws an error for `double(1.2)` .

We could add a method for type `Float64`, but Julia offers us a different route: abstract types. In this case, we can use the type `Number`, as follows:

`double(x::Number) = 2x` DEMO: now `double(3.3) works`

In fact, we could even go further and provide a default method, with `double(x) = 2x`. This is a fall-back method, used only when the others do not apply. That is, it is used only when `x` is not `Int64`, nor `String`, nor any type that falls under the abstract type `Number`. The fall-back method accepts the value of `x` whatever its type may be. There is a way to specify this: the code `double(x::Any) = 2x` is exactly the same as `double(x) = 2x` .

Abstract versus concrete types

An abstract type is simply a way of talking about several types at the same time. If a Julia type has one or more subtypes, then it is an abstract type. We can check for this using the function `subtypes`. For example `Number` has the subtypes `Complex` and `Real`: [DEMO].

This means that although `Number` is a distant supertype of both `Int64` and `Float64`, it is not the immediate supertype of either of them. [DEMO: repeatedly use `subtype` to show that eventually the types `Int64` and `Float64` turn up.]

Every type in Julia is a subtype of exactly one type. We check for it using the function `supertype`.

DEMO:
verify that both `Complex` and `Real` have the supertype `Number`

On the other hand, some Julia types are *concrete types*: they have no subtypes.

DEMO: verify that `Int64`, `Float64` and `String` have no subtypes.

The most abstract type of all in Julia is `Any` which is its own supertype[1] and the eventual supertype of all Julia's types.

Here is a crucial point: every *value* in Julia has a concrete type. A variable does not in itself have a type. Sometimes we do speak of the type of a variable, for example for the function definition `double(x::Int64)` we might say that `x` is of type `Int64`, but that is simply for convenience[2].

What then of arrays? Julia, as far possible, tries to ensure that every element of an array has the same type. [DEMO: `[1, 1.0]` ends up with two values of type `Float64`.]

This makes for efficient computing[3], but it isn't always what we want. To allow arrays of mixed type, we explicitly have to declare the array to be of an abstract type that includes all the required types. The most general is of course `Any[...]`; less general but still abstract types can often be used.

[DEMO: `Number[1, 1.0]` is a convenient way to specify the required array[4].]

Julia's type system enables multiple dispatch

We can now answer the question: when a function has many methods, how does Julia choose which one to use? The answer lies in the type system, and in particular in the idea of subtypes.

Consider the call `double(7)`. The type signature of this call is "one variable of type `Int64`". There

[1] For completeness, we note that Julia's type system is a tree: if you start with any concrete type, and repeatedly ask for supertype, you always end up at `Any`.

[2] It is hopelessly pedantic to say "This method applies when the variable `x` has a value of type `Int64`", but that is what is accurate.

[3] The ability to ensure efficient computing is one of Julia's very strong points. It requires careful coding of functions to ensure type stability, but that is a topic beyond the scope of this course.

[4] For completeness, we note that there are other ways to do the same thing. We won't study them in this course.

are three methods which may apply, with type specifications `double(x::Int64)`, `double(x::Float64)` and `double(x::Any)`. Julia chooses the one with the more concrete type specification, in this case the method for `x::Int64`.

It works like this. Firstly, Julia chooses a method for a function only when it calls that function while it is actually running the code[5]. Since all values in Julia have a concrete type, the type signature of function call consists of concrete types. Secondly, Julia searches the methods for type specifications that match. For each value in the type signature of the call, it searches for a method that matches that value—if there is no method for the concrete type, it searches for a method with the supertype of that concrete type, if none such then for a method with the supertype of the supertype, and so on[6].

Ideally, the search described above finds exactly one method. Here is an example with two methods where it doesn't:

```
disptype(x::Int64, y) = println("x requires Int64, y can be Any")
disptype(x, y::Int64) = println("x can be Any, y requires Int64")
```

[DEMO: verify that `disptype(7, 7.0)` works, `disptype(7.0, 7.0)` has no method, `disptype(7.0, 7)` works, and `disptype(7, 7)` is flagged as ambiguous, with two possible methods.]

It is part of your job as a programmer to make sure all of your methods apply exactly where and when you want them to. On this course, that will be easy.

A final remark

Julia's type system includes thousands of types. Expert Julia programmers know very many of them. Moreover, in their code they often extend Julia's type system with their own user-defined types. It is a very powerful system and at the heart of what makes Julia a language that is easy to extend while still being very, very fast.

However, you need not specify type ever, in any of the code you write, and on this course only a few exercises ask you to do so. This makes code in Julia easy to write, even for a beginner[7], which is why this course is possible. You need to know about the type system for two reasons only: to help you with debugging code that throws type errors, and to understand how multiple dispatch works.

Review and summary

* `Number` is an abstract type
* The immediate supertype of a type is given by the function `supertype`
* The type `Number` is an eventual supertype of both `Int64` and `Float64`

[5] This is because Julia uses just-in-time compiling. This fascinating topic is beyond the scope of this course.

[6] This ends with the type `Any`, which includes all of Julia's types.

[7] The price is that you sometimes your code will not run anywhere near the maximum speed that Julia could deliver if types were carefully used.

* The type `Any` is the eventual supertype of all values in Julia

* The subtypes of a type is given by the function `subtypes`

* A concrete type has no subtypes

* Julia's values all belong to concrete types; all actual computations are done with concrete types

* Multiple dispatch is Julia's way of matching type specification of function method to type signature of function call; all the values in the call must match the type specification in the code.

* It is possible for a function call to be ambiguous, in the sense of matching more than one function specification.

Lesson 6, Week 2: Scope I (functions)

AIM

— To introduce the concept of scope as it applies in Julia programs

After this lesson, you will be able to

* Say what the scope is of a block of code
* Say what it means for a block of code to have local scope
* Explain the difference between local scope and global scope
* Explain how a function in Julia treats variables from the global scope
* Use the keyword `global` to make global variables accessible to writing inside a function
* Explain why, even though they're global, the values inside an array can be modified inside a function

What is scope, in the sense of programming?

The term *scope* refers to which variables can be accessed in a particular block of code.

So what is a block of code? Well, any portion of a program that doesn't leave out any lines between the first and last lines in the block. So the program as a whole is a code block, and the variables that are accessible to the whole program are called the *global* variables[1].

However, some code blocks introduce *local scope*. Variables that are local to that scope cannot be accessed by code outside the code block that introduces the local scope[2].

You have already seen one way to make a local scope: comprehensions.

DEMO:
```
xyz = "abcd"
vecxyz = [locvar^4 for locvar in xyz]
```

We see that the variable `locvar` is available inside the comprehension, but not outside it.

[1] Roughly speaking, the global variables are those in the namespace of the program as a whole. If there is only one namespace, then all its variables are global.

[2] Whether global variables are accessible in the local scope is a delicate matter, as we will see, and the rules are not the same for all ways to create local scope.

Please be aware that a variable may be accessible for reading or for writing or for both. On this course, all code blocks that do not use local scope are part of the global scope[3].

Which keywords introduce a new scope?

For us at the moment, the important point is that the keyword `function` creates a local scope[4]. All variables that are created inside the function body are local to that scope and for all practical purposes are invisible to any code outside the function body.

Note that even if you re-use the name of a global variable to create a local variable with the same name, then you have created two different variables, one local, one global. This is obviously a conflict[5], and should be avoided.

Global variables are accessible for reading inside a function, but not for writing—with the exception that the keyword `global` can be used to make a global variable accessible for writing in a block with local scope.

Some perhaps unexpected effects of Julia's scoping rules for functions

Consider the functions `eg1` and `eg2` defined below

```
eg1(x) = x+y
```

```
eg2(x, y) = x+y
```

DEMO: define `x,y = 2,3` and explain what happens in the cases `eg1(x)`, `eg2(x, y)`, `eg1(6)` and `eg2(6, 7)`

We see that in `eg1`, the value of `y` is the global value, while in `eg2`, the value is whatever is passed to the function in the second position. In other words, it is not necessary to use the variable names `x` or `y` when writing a function call[6]. It should be clear that, by putting the name `y` in the argument list of `eg2` when we write the code that defines the function, we make `y` a *local* variable, visible only in the local scope of `eg2`.

Let us now try to change the value of the global variable `y` in these functions. We have to use the keyword `function` in order to get multiline bodies for both functions:

[3] This is because we work only in the REPL, which has only one namespace.
[4] We'll see some of the others later; there are 10 such keywords (as of Julia 1.1).
[5] Alas, it happens very easily and can lead to bugs that are very hard to find.
[6] Reminder: a function call is code where you tell Julia to execute the function.

```
function eg1(x)
    z = x+y
    y = 11
    return z
end
```

```
function eg2(x, y)
    z = x+y
    y = 11
    return z
end
```

[DEMO: test the same values as above, but also test whether the global value of `y` changes. Discuss.]

[DEMO: for `eg1`, replace `y = 11` with `global y = 1`, repeat test, discuss.]

[DEMO: explain why the same change doesn't work for `eg2`.]

We see that specifying that `y` as `global` insde the function body works only if it hasn't already been specified as local in the argument list. And by the way, we don't have to use combine `global` with assignment as we did there, we can have the line `global y` anywhere in the function body and just use `y=11` as before.

The special case of arrays

We have said that global variables cannot be modified inside a function (at least, not without using the keyword `global`). But arrays may seem to violate that rule:

```
function modaa()
  aa = 7
end
```

```
function modbb()
  bb[2] = 7
end
```

DEMO:
```
aa, bb = 123, [1, 2]
modaa(); modbb()
aa, bb
```

We see that as expected `modaa` cannot modify the scalar variable `aa`, but `modbb` succeeds in modifying one of the elements of `bb`.

However, in this case the variable `bb` as a vector doesn't change: it remains two `Int64` numbers. In that sense, it satisfies the rule. It is the values inside the array that are not subject to the rule.

This is not behaviour peculiar to functions; it is general. When you create an array, Julia tries to not to have to allocate new memory. Compare the code on the left to the code on the right:

```
aa = 2
bb = aa
bb = 3
println("$aa, \t$bb")
```

```
aa = [1, 2]
bb = aa
bb[2] = 3
println("$aa, \t$bb")
```

On the left, when `bb=aa` is executed, the name `bb` binds to same bit of memory as the name `aa`. When `bb=3` is executed, the name `bb` simply binds to a new bit of memory. So the result is that `aa` and `bb` end up with different values.

On the right, when `bb=aa` is executed, it also binds to the same memory as `aa`. But this `aa` is an array, it is potentially enormous, because there is almost no limit to the size an array can have in Julia. The designers of Julia favour performance, so they wish to avoid copying arrays whenever they can. This means that when `bb[2]=3` is executed, no new memory is used. Instead, the memory that `aa` and `bb` share is changed. That is why changing an element in `bb` changes the corresponding element `aa`, so that they remain equal[7].

Review and summary

* Any part of code can be a block, all the is needed is that it doesn't leave out any lines from its first to its last line
* The scope of a block of code is all the variables that are visible inside that code
* Variables can be visible for reading only or for reading and writing
* A block with local scope means that variables created in that block are not visible outside it
* Comprehension creates local scope
* The `function` keyword creates local scope that ends with its `end` keyword[8]
* Global variables are visible inside local scope for reading only
* Any variables that are mentioned in the input to a function call are in the local scope of that function
* The keyword `global` can be applied to a variable name inside the function body[9], and makes the global variable of that name available to writing inside the function body
* Values inside arrays that are in global scope can be modified in the local scope of a function
* When an array-valued variable is assigned to a second names, both names always bind to the same value; changing elements in that value changes both variables.

[7] This is one of the ways in which Julia differs from many other languages.
[8] In other words, the code in the function body has local scope
[9] Provided the function body does not have a local variable of that name.