Table of Contents

Introduction 1.1

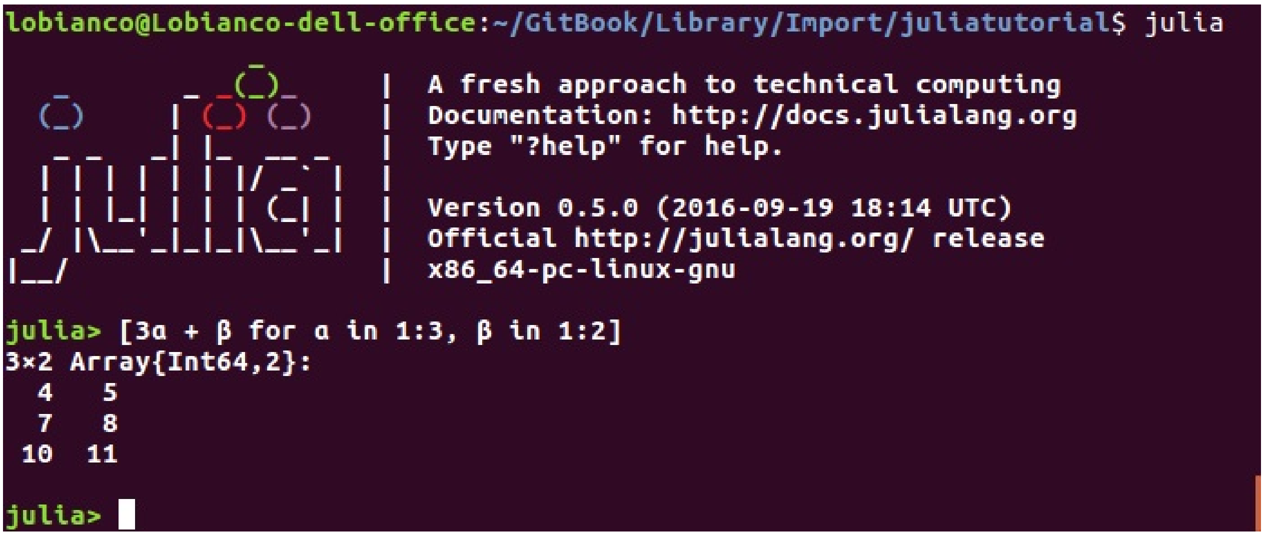
Language core

1 - Getting started 2.1
2 - Data types 2.2
3 - Control flow 2.3
4 - Functions 2.4
5 - Custom structures 2.5
6 - Input - Output 2.6
7 - Managing run-time errors (exceptions) 2.7
8 - Interfacing Julia with other languages 2.8
9 - Metaprogramming 2.9
10 - Performances (parallelisation, debugging, profiling..) 2.10
11 - Developing Julia packages 2.11

Useful packages

Plotting 3.1
DataFrames 3.2
JuMP 3.3
SymPy 3.4
Weave 3.5
LAJuliaUtils 3.6
IndexedTables 3.7
Pipe 3.8

Introduction



Compatibilities table of this tutorial with Julia versions:

- **Julia 1.0:** From 5 September 2018
- **Julia 0.6:** 19 July 2017 - 15 August 2018 versions
- **Julia 0.5:** Versions before 19 July 2017

The purposes of this tutorial are (a) to store things I learn myself about Julia and (b) to help those who want to start coding in Julia before reading the 982 pages of the (outstanding) official documentation.

This document started as a compendium of several tutorials (plus the official documentation), in particular Chris Rackauckas's A Deep Introduction to Julia, the Quantecon tutorial, the WikiBook on Julia and Learn X in Y minutes, from which I did borrow several examples.

The focus is on Julia as a generic programming language rather than on domain-specific issues (but some domain-specific topics are covered in the "Useful packages" section). The format is in the middle between a classical tutorial and a cheatsheet: the tutorial describes the elements of the language following the typical sections of a programming language tutorial (*data types, control flows, functions..*), but the information is given in a pretty concise way, suitable for people that already have some knowledge in other programming languages (e.g. what a `for` loop does is not explained, but how it is implemented in Julia is).

English is not my primary language, so please be understanding and report me of any errors, both in the language and in the content.

Happy coding with Julia !

Antonello Lobianco

Latest version

- The latest version of this tutorial can be found online on GitBook, at https://syl1.gitbook.io/julia-language-a-concise-tutorial
- PDF version (if it works)
- A legacy interface (if it works)
- Corresponding GIT repository

I am considering migrating to other documentation systems, as the new GitBook is pretty limited. If so, I will nevertheless update the link here.

Citations

Please cite this tutorial as:

```
A. Lobianco, (2018), “Julia language: a concise tutorial", GitBook,
https://syl1.gitbook.io/julia-language-a-concise-tutorial , retrieved xx/xx/xxxx
```

Acknowledgements

Development of this tutorial was supported by:

- the French National Research Agency through the Laboratory of Excellence ARBRE, a part of the “Investissements d'Avenir” Program (ANR 11 – LABX-0002-01).

1 - Getting started

Why Julia

Without going into long discussions, Julia (partially thankful for the recent development in *just-in-time* compilers) solves a trade-off that has long been existed in programming: *fast coding* vs. *fast execution*.
On one side, Julia allows to code in a dynamic language like Python, R or Matlab, allowing interaction with the code and powerful expressivity (see the Metaprogramming chapter for example).
On the other side, with minimum efforts (see Performances), programs written in Julia can run (almost) as fast as C.
While still young, Julia allows to easily interface your code with all the major programming languages (see Interfacing Julia with other languages), hence reusing their huge set of libraries (when these are not already being ported in Julia).
Julia has its roots in the domain of scientific, high performances programming, but it is becoming more and more mature as a general purpose programming language.

Installing Julia

All you need to run the code in this tutorial is a working Julia interpreter console (aka REPL - *Read Eval Print Loop*).
In a recent version of Linux you could simply use your package manager to install `julia` but for more up-to-date version, or for Windows/Mac packages, I strongly suggest to download the binaries available on the download section of the Julia web-site.

For Integrated Development Editor, checkout either Juno or IJulia, the Julia Jupiter backend. Here you can find their detailed setup instructions:

- Juno (an useful tip I always forget: the key binding for block selection mode is `ALT+SHIFT`)
- IJulia (in a nutshell: if you already have Jupiter installed, just run `using Pkg; Pkg.update();Pkg.add("IJulia")` from the Julia console. That's all! ;-))

You can also choose, at least to start with, *not* to install Julia at all, and try instead JuliaBox, a free online IJulia notebook server that you access from your browser.

Running Julia

There are several ways to run Julia code:

1. Julia can be run interactively in a console.

   Just write (after having installed it) `julia` in a console and then type your commands there (and type `exit()` when you have finished);

2. Alternatively, create a script, that is a text file ending in `.jl`, and let Julia parse and run it with `julia myscript.jl [arg1, arg2,..]`;

3. Script files can also be run from within the Julia console, just type `include("myscript.jl")`;
4. In Linux, you could instead add at the top of the script the location of the Julia interpreter on your system, preceded by `#!` and followed by an empty row , e.g. `#!/usr/bin/julia` (you can find the full path of the Julia interpreter by typing `which julia` in a console). Be sure that the file is executable (e.g. `chmod 755 myscript.jl`). Then you can run the script with `./myscript.jl`;
5. Use an Integrated Development Editor (such as [Juno](include("test_script.jl") or Jupiter), open your Julia script and use the run command of the editor.

Julia keeps many things in memory within the same work session. If this create problems in the execution of your code, you can restart Julia or use the Revise.jl package for a finer control.

You can check which version of Julia you are using with `versioninfo().`

Syntax elements

Single line comments start with `#` and multi-line comments can be placed in between `#=` and `=#` (and can be nested).

In console mode, `;` after a command suppresses the output (this is done automatically in scripting mode), and typed alone switches to one-time command shell prompt.

Indentation doesn't matter, but empty spaces sometimes do, e.g. functions must have the curved parenthesis with the inputs strictly attached to them, e.g.:

```
println (x)   ERROR
println(x)    OK
```

If you come from C or Python, one important thing to remember is that Julia is one-based indexing (arrays start counting from `1` and not `0`).

Packages

Many functions are provided in Julia by external "packages". Also, many standard functionalities that were in core before Julia 1.0 has been moved to a separate standard library, shipped with Julia itself, but that requires the user to load the package explicitly.

For example, the same package functionality requires the user to type `using Pkg` to access the Pkg functionalities (alternatively, only for the package module, you can type `]` to enter a "special" Pkg mode).

You can then run the desired command, directly if you are in a terminal in the Pkg mode, or `pkg"cmd to run"` in a script (notice that there is no space between `pkg` and the quoted command to run).

Some useful package commands:

1. `status` Retrieves a list with name and versions of locally installed packages
2. `update` Updates your local index of packages and all your local packages to the latest version
3. `add myPkg` Automatically downloads and installs a package
4. `rm myPkg` Removes a package and all its dependent packages that has been installed automatically only for it
5. `add pkgName#master` Checkouts the master branch of a package (and `free pkgName` returns to the released version)
6. `add pkgName#branchName` Checkout a specific branch
7. `add git@github.com:userName/pkgName.jl.git` Checkout a non registered pkg

To use the functions provided by a package, just include a `using mypackage` statement in the console or at the beginning of the script. If you prefer to import the package but keep the namespace clean, use `import mypackage` (you will then need to refer to a package function as `myPkg.myFunction`). You can also include other files using `include("MyFile.jl")`: when that line is run, the included file is completely ran (not only parsed) and any symbol defined there becomes available in the namespace relative to where include has been called.

`Winston` or `Plots` (plotting) and `DataFrames` (R-like tabular data) are example of packages that you will pretty surely want to consider.

For example (see the Plotting section for specific Plotting issues):
(note: as of writing, the Plot package has not yet be ported to Julia 1.0)

```
using Plots
pyplot()
plot(rand(4,4))
```

or

```
import Plots
const pl = Plots # this create an an alias, equivalent to Python "import Plots as pl".
 Declaring it constant may improve the performances.
pl.pyplot()
pl.plot(rand(4,4))
```

or

```
import Plots: pyplot, plot
pyplot()
plot(rand(4,4))
```

You can read more about packages in the relevant section of the Julia documentation, and if you are interested in writing your own package, skip to the "Developing Julia package" section.

Help system (Julia and package documentation)

Typing `?` in the console leads you to the Julia help system where you can search for function's API (in non-interactive environment you can use `?search_term` instead). If you don't remember exactly the function name, Julia is kind enough to return a list of similar functions.

2 - Data types

Scalar types

In Julia, variable names can include a subset of Unicode symbols, allowing a variable to be represented, for example, by a Greek letter.
In most Julia development environments (including the console), to type the Greek letter you can use a LaTeX-like syntax, typing `\` and then the LaTeX name for the symbol, e.g. `\alpha` for `α`. Using LaTeX syntax, you can also add subscripts, superscripts and decorators.

The main types of scalar are `Int64`, `Float64`, `Char` (e.g. `x = 'a'`), `String` [1] (e.g. `x="abc"`) and `Bool`.

Strings

Julia supports most typical string operations, for example: `split(s)` *(default on whitespaces)*, `join([s1,s2], "")`, `replace(s, "toSearch" => "toReplace")` and `strip(s)` *(remove leading and trailing whitespaces)* Attention to use the single quote for chars and double quotes for strings. c

Concatenation

There are several ways to concatenate strings:

- Concatenation operator: `*`;
- Function `string(str1,str2,str3)`;
- Combine string variables in a bigger one using the dollar symbol: `a = "$str1 is a string and $(myobject.int1) is an integer"` ("interpolation")

Note: the first method doesn't automatically cast integer and floats to strings.

Arrays (lists)

Arrays are N-dimensional mutable containers. In this section, we deal with 1-dimensional arrays, in the next one we consider2 or more dimensional arrays.

There are several ways to create an array:

- Empty (zero-elements) arrays: `a = []`. Alternative ways:
    - `a = T[]`, e.g. `a = Int64[]`;
    - using explictitly the contructor `a = Array{T,1}()`;
    - using the `Vector` alias: `c = Vector{T}()`;
- 5-elements zeros array: `a=zeros(5)` (or a=`zeros(Int64,5)`) (same with `ones()`)
- Column vector (*Vector* container, alias for 1-dimensions array) : `a = [1;2;3]` or `a=[1,2,3]`
- Row vector (*Matrix* container, alias for 2-dimensions array, see next section "Multidimensional and nested arrays"): `a = [1 2 3]`

Arrays can be heterogeneous (but in this case the array will be of `Any` type and in general much slower): `x = [10, "foo", false]`.

If you need to store a limited set of types in the array, you can use the `Union` keyword to still have an efficient implementation, e.g. `a = Union{Int64,String,Bool}[10, "Foo", false]`.

`a = Int64[]` is just a shorthand for `a = Array{Int64,1}()` (e.g. `a = Any[1,1.5,2.5]` is equivalent to `a = Array{Any,1}([1,1.5,2.5])`). Attention that `a = Array{Int64,1}` (without the round brackets) doesn't create an Array at all, but just assign the "DataType" `Array{Int64,1}` to `a`. You can also declare an array of size *n* (with garbage content) with `a=Array{T,1}(undef,n)`.

Square brackets are used to access the elements of an array (e.g. `a[1]`). The slice syntax `[from:step:to]` is generally supported and in several contexts will return a (fast) iterator rather than a list (you can use the keyword `end`, but not `begin`). To then transform the iterator in a list use `collect(myiterator)`. You can initialise an array with a mix of values and ranges with either `y=[2015; 2025:2030; 2100]` (note the semicolon!) or `y=vcat(2015, 2025:2030, 2100)`.

The following methods are useful while working with arrays:

- Push an element to the end of a: `push!(a,b)` (as a single element even if it is an Array. Equivalent to python `append`)
- To append all the elements of b to a: `append!(a,b)` (if b is a scalar obviously push! and append! are the same. Attention that a string is treated as a list!. Equivalent to Python `extend` or `+=`)
- Concatenation of arrays (new array): `a = [1,2,3]; b = [4,5]; c = vcat(1,a,b)`
- Remove an element from the end: `pop!(a)`
- Removing an element at the beginning (left): `popfirst!(a)`
- Remove an element at an arbitrary position: `deleteat!(a, pos)`
- Add an element (b) at the beginning (left): `pushfirst!(a,b)` (no, `appendfirst!` doesn't exists!)
- Sorting: `sort!(a)` or `sort(a)` (depending on whether we want to modify or not the

original array)
- Reversing an arry: `a[end:-1:1]`
- Checking for existence: `in(1, a)`
- Getting the length: `length(a)`
- Get the maximum value: `maximum(a)` or `max(a...)` (`max` returns the maximum value between the given arguments)
- Get the minimum value: `minimum(a)` or `min(a...)` (`min` returns the maximum value between the given arguments)
- Empty an array: `empty!(a)` (only column vector, not row vector)
- Transform row vectors in column vectors: `b = vec(a)`
- Random-shuffling the elements: `shuffle(a)` (or `shuffle!(a)` . From Julia 1.0 this require `using Random` before)
- Checking if an array is empty: `isempty(a)`
- Find the index of a value in an array: `findall(x -> x == value, myarray)` . This is a bit tricky. The first argument is an anonymous function that returns a boolean value for each value of `myarray` , and then `find()` returns the index position(s).
- Delete a given item from a list: `deleteat!(myarray, findall(x -> x == myunwanteditem, myarray))`

Multidimensional and nested arrays

In Julia, an array can have 1 dimension (a column, also known as `Vector`), 2 dimensions (that is, a `Matrix`) or more. Then each element of the Vector or Matrix can be a scalar, a vector or an other Matrix.
The main difference between a `Matrix` and an *array of array* is that in the former the number of elements on each column (row) must be the same and rules of linear algebra applies.

There are two ways to create a Matrix:

- `a = [[1,2,3] [4,5,6]]` [[elements of the first column] [elements of the second column] ...] (note that this is valid only if wrote in a single line. Use `hcat(col1, col2)` to write matrix by each column)
- `a = [1 4; 2 5; 3 6]` [elements of the first row; elements of the second row; ...] (here you can also use `vcat(row1, row2)` to concatenate several rows)

Attention to this difference:

- `a = [[1,2,3],[4,5,6]]` creates a 1-dimensional array with 2-elements (each of those is again a vector);
- `a = [[1,2,3] [4,5,6]]` creates a 2-dimensional array (a matrix with 2 columns) with three elements (scalars).

Empty matrices can be constructed as:

`m = Array{Float64}(undef, 0, 0)`

for an *(0,0)-size* 2-D Matrix of type `Float64` and more in general:

`m = Array{T}(undef, a, b, ...,z)`

for an *(a,b,...,z)-size* multidimensional Matrix (whose content, of type `T,` is garbage)

A 2x3 matrix can be constructed in one of the following ways:

- `a = [[1,2] [3,4] [5,6]]`
- `a = zeros(2,3)` or `a = ones(2,3)` (the zeros and ones are strored as `Float64`)
- `a = fill("abc",2,3)` (content is "abc")

Nested arrays can be accessed with double square brackets, e.g. `a[2][3]` .
Elements of bidimensional arrays can be accessed instead with the `a[row,col]` syntax, where again the slice syntax can be used, for example, given `a` is a 3x3 Matrix, `a[1:2,:]` would return a 2x3 Matrix with all the column elements of the first and second row.

Boolean selection is implemented using a boolean array/matrix for the selection:

```
a = [[1,2,3] [4,5,6]]
mask = [[true,true,false] [false,true,false]]
```

`a[mask]` returns an 1-D array with 1, 2 and 5. Note that boolean selection results always in a flatted array, even if delete a whole row or a whole column of the original data. It is up to the programmer to then reshape the data accordingly.

Note: for row vectors, both `a[2]` or `a[1,2]` returns the second element.

n-D arrays support several methods:

- `size(a)` returns a tuple with the sizes of the *n* dimensions
- `ndims(a)` returns the number of dimensions of the array (e.g. 2 for a Matrix)
- Arrays can be changed dimension with either `reshape(a, nElementsDim1, nElementsDim2)` or `dropdims(a, dims=(dimToDrop1,dimToDrop2))` (where the dim(s) to drop must all have a single element for all the other dimensions, e.r. be of `size` 1) the transpose `'` operator. These operations perform a shadow copy, returning just a different "view" of the underlying data (so modifying the original matrix modifies also the reshaped/transposed matrix). You can use `collect(reshape/dropdims/transpose)` to force a deepcopy.

`AbstractVector{T}` is just an alias to `AbstractArray{T,1}` , as `AbstractMatrix{T}` is just an alias to `AbstractArray{T,2}` .

Multidimensional Arrays can arise for example from using list comprehension: `a = [3x + 2y + z for x in 1:2, y in 2:3, z in 1:2]`

For further operations on arrays and matrices have a look at the QuantEcon tutorial.

Tuples

Use tuples to have a list of immutable elements: `a = (1,2,3)` or even without parenthesis `a = 1,2,3`

Tuples can be easily unpacked to multiple variable: `var1, var2 = (x,y)` (this is useful, for example, to collect the values of functions returning multiple values)

Useful tricks:

- Convert a tuple in a vector: `a=(1,2,3); v = [a...]` or `v = [i[1] for i in a]` or `v=collect(a)`
- Convert an array in tuple: `a = (v...,)`

NamedTuples

NamedTuples are collections of items whose position in the collection (index) can be identified not only by the position but also by name.

- Define a NamedTuple: `aNamedTuple = (a=1, b=2)`
- Access them with the dot notation: `aNamedTuple.a` .
- Get a tuple of the keys: `keys(aNamedTuple)`
- Get a tuple of the values: `values(aNamedTuple)`
- Get an Array of the values: `collect(aNamedTuple)`
- Get a iterable of the pairs (k,v): `pairs(aNamedTuple)` . Useful for looping: `for (k,v) in pairs(aNamedTuple) [...] end`

As "normal" tuples, NamedTuples can hold any values, but cannot be modified (i.e. are "immutable").

Before Julia 1.0 Named Tuples were implemented in a separate package (NamedTuple.jl). The idea is that, like for the Missing type, the separate package provides additional functionality to the core `NamedTuple` type, but there is still a bit of confusion over it and, at time of writing, the additional package still provide its own implementation (and many other external packages require it), resulting in crossed incompatibilities.

Dictionaries

Dictionaries store mappings from keys to values, and they have an apparently random sorting.

You can create an empty (zero-elements) dictionary with `mydict = Dict()`, or initialize a dictionary with values: `mydict = Dict('a'=>1, 'b'=>2, 'c'=>3)`

There are some useful methods to work with dictionaries:

- Add pairs to the dictionary: `mydict[akey] = avalue`
- Add pairs using maps (i.e. from vector of keys and vector of values to dictionary): `map((i,j) -> mydict[i]=j, [1,2,3], [10,20,30])`
- Look up values: `mydict['a']` (it raises an error if looked-up value doesn't exist)
- Look up value with a default value for non-existing key: `get(mydict,'a',0)`
- Get all keys: `keys(mydict)` (the result is an iterator, not an Array. Use `collect()` to transform it.)
- Get all values: `values(mydict)` (result is again an iterator)
- Check if a key exists: `haskey(mydict, 'a')`
- Check if a given key/value pair exists (that it, if the key exists and has that specific value): `in(('a' => 1), mydict)`

You can iterate trough both the key and the values of a dictionary at the same time:

```
for (k,v) in mydict
   println("$k is $v")
end
```

While named tuples and dictionaries can look similar, there are some important difference between them:

- NamedTuples are immutable while Dictionaries are mutable
- Dictionaries are type unstable if different type of values are stored, while NamedTuples remain type-stable:
  - `d = Dict(:k1=>"v1", :k2=>2) # Dict{Symbol,Any}`
  - `nt = (k1="v1", k2=2,) # NamedTuple{(:k1, :k2),Tuple{String,Int64}}`
- The syntax is a bit less verbose and readable with NamedTuples: `nt.k1` vs `d[:k1]`

Overall, NamedTuple are generally more efficient and should be tought more as anonymous `struct` (see the "Custom structure" section) than Dictionaries.

Sets

Use sets to represent collections of unordered, unique values.

Some methods:

- Empty (zero-elements) set: `a = Set()`
- Initialize a set with values: `a = Set([1,2,2,3,4])`
- Set intersection, union, and difference: `intersect(set1,set2)` , `union(set1,set2)` , `setdiff(set1,set2)`

Memory and copy issues

In order to unnecessarily copying large amount of data, Julia by default copy only the memory address of large objects, unless the programmer explicitly request a so-called "deep" copy. In detail:

Equal sign (a=b)

- "simple" types (e.g. `Float64, Int64,` but also `String`) are deep copied
- containers of simple types (or other containers) are shadow copied (their internal is only referenced, not copied)

copy(x)

- simple types are deep copied
- containers of simple types are deep copied
- containers of containers: the content is shadow copied (the content of the content is only referenced, not copied)

deepcopy(x)

- everything is deep copied recursively

You can check if two objects have the same values with `==` and if two objects are actually the same with `===` (in the sense that immutable objects are checked at the bit level and mutable objects are checked for their memory address):

- given `a = [1, 2]; b = [1, 2];` , `a == b` and `a === a` are true, but `a === b` is false;
- given `a = (1, 2); b = (1, 2);` , all `a == b, a === a` and `a === b` are true.

Various notes on Data types

While boolean values `true` and `false` are evaluated to `1` and `0` respectively, the opposite is not true. So, `if 0 [...] end` brings a *non-boolean (Int64) used in boolean context* `TypeError` .

Attention to the keyword `const`. When applied to a variable (e.g. `const x = 5`) doesn't mean that the variable can't change value (as in C), but simply that it can not change type. Only global variables can be declared constant.

To convert ("cast") between types, use `convertedObj = convert(T,x)`. Still, when conversion is not possible, e.g. trying to convert a 6.4 Float64 in a Int64 value, an error, will be risen (`InexactError` in this case).

To convert strings (representing numbers) to integers or floats use `myInt = parse(Int,"2017")`.

The opposite, to convert integers or floats to strings, use `myString = string(123)`.

You can "broadcast" a function to work over an Array (instead of a scalar) using the dot (`.`) operator.
For example, to broadcast `parse` to work over an array use: `myNewList = parse.(Float64, ["1.1","1.2"])` (see also Broadcast in the "Functions" Section)

Variable names have to start with a letter, as if they start by a number there is ambiguity if the initial number is a multiplier or not, e.g. in the expression `6ax` the variable `ax` is multiplied by 6, and it is equal to `6 * ax` (and note that `6 ax` would result in a compile error). Conversely, `ax6` would be a variable named `ax6` and not `ax * 6`.

You can import data from a file to a matrix using `readdlm()` (in standard library package `DelimitedFiles`). You can skip rows and/or columns using slice operator and then converting to the desidered type, e.g.

`myData = convert(Array{Float64,2},readdlm(myinputfile,'\t')[2:end,4:end]); # skip the first 1 row and the first 3 columns`

Random numbers

- Random float in [0,1]: `rand()`
- Random integer in [a,b]: `rand(a:b)`
- Random float in [a,b] with "precision" to the second digit : `rand(a:0.01:b)`

  This last can be executed faster and more elegantly using the `Distribution` package:

  ```
  using Pkg; Pkg.add("Distributions")
  import Distributions: Uniform
  rand(Uniform(a,b))
  ```

You can obtain an Array or a Matrix of random numbers simply specifying the requested size to rand(), e.g. `rand(2,3)` or `rand(Uniform(a,b),2,3)` for a 2x3 Matrix.

Missing, nothing and NaN

Julia supports different concepts of missingness:

- **`nothing`** (type `Nothing`): is the value returned by code blocks and functions which do not return anything. It is a single instance of the singleton type `Nothing`, and the closer to C style NULL (sometimes it is referred as to the "software engineer's null"). Most operations with `nothing` values will result in a run-type error. In some contexts it is printed as `#NULL`;
- **`missing`** (type `Missing`): represents a missing value in a statistical sense: there should be a value but we don't know which is (so it is sometimes referred to as the "data scientist's null"). Most operations with missing values will result in missing propagate (silently). Containers can handle missing values efficiently when are declared of type `Union{T,Missing}`. The Missing.jl package provides additional methods to handle missing elements;
- **`NaN`** (type `Float64`): represents when an operation result in a Not-a-Number value (e.g. 0/0). It is similar to `missing` in the fact that it propagates silently. Similarly, Julia also offers `Inf` (e.g. `1/0`) and `-Inf` (e.g. `-1/0`).

[1]: Technically a `String` is an array in Julia (try to append a String to an array!), but for most uses it can be thought as a scalar type.

3 - Control flow

All typical control flow (`for` , `while` , `if` / `else` , `do`) are supported, and parenthesis around the condition are not necessary. Multiple conditions can be specified in the for loop, e.g.:`

```
for i=1:2,j=2:4
 println(i*j)
end
```

`break` and `continue` are supported and works as expected.

Julia support list comprehension and maps:

- `[myfunction(i) for i in [1,2,3]]`
- `[x + 2y for x in [10,20,30], y in [1,2,3]]`
- `mydict = Dict(); [mydict[i]=value for (i, value) in enumerate(mylist)]` (`enumerate` returns an iterator to tuples with the index and the value of elements in an array)
- `[students[name] = sex for (name,sex) in zip(names,sexes)]` (`zip` returns an iterator of tuples pairing two or multiple lists, e.g. [("Marc","M"),("Anne","F")])
- `map((n,s) -> students[n] = s, names, sexes)` (`map` applies a function to a list of arguments) When mapping a function with a single parameter, the parameter can be omitted: `a = map(f, [1,2,3])` is equal to `a = map(x->f(x), [1,2,3])` .

Ternary operator is supported as `a ? b : c` (if `a` is true, then `b` , else `c`). Put attention to wrap the `?` and `:` operators with space.

Logical operators

- And: `&&`
- Or: `||`
- Not: `!`

Not to be confused with the bitwise operators `&` and `|` .

Currently `and` and `or` aliases to respectively `&&` and `||` has not being imlemented.

Do blocks

Do blocks allow to define anonymous functions that are passed as first argument to the outer functions. For example, `findall(x -> x == value, myarray)` expects the first argument to be a function. Every time the first argument is a function, this can be written at posteriori with a do block:

```
findall(myarray) do x
       x == value
end
```

This defines `x` as a variable that is passed to the inner contend of the `do` block. It is the task of the outer function to where to apply this anonymous function (in this case to the `myarray` array) and what to do with its return values (in this case boolean values used for computing the indexes in the array). More infos on the do blocks: https://en.wikibooks.org/wiki/Introducing_Julia/Controlling_the_flow#Do_block , https://docs.julialang.org/en/stable/manual/functions/#Do-Block-Syntax-for-Function-Arguments-1

4 - Functions

Functions can be defined inline or using the `function` keyword, e.g.:

`f(x,y) = 2x+y`

```
function f(x)
  x+2
end
```

(a third way is to create an anonymous function and assign it to a nameplace, see later)

Arguments

Arguments are normally specified by position, while arguments given after a semicolon are instead specified by name.
The call of the function must respect this distinction, calling positional argument by position and keyword arguments by name (e.g., it is not possible to call positional arguments by name).
The last argument(s) (whatever positional or keyword) can be specified together with a default value.

`myfunction(a,b=1;c=2) = (a+b)*3 # definition with 2 position arguments and one keyword argument myfunction(1,c=3) # calling (1+2)*3`

To declare a function parameter as being either a scalar type `T` or a vector `T` you can use an Union: `function f(par::Union{Float64, Vector{Float64}} = Float64[]) [...] end`

The ellipsis (splat `...`) can be uses in order to both specify a variable number of arguments and "splicing" a list or array in the parameters of a function call:

```
values = [1,2,3]
function average(init, args...) #The parameter that uses the ellipsis must be the last
 one
  s = 0
  for arg in args
    s += arg
  end
  return init + s/length(args)
end
a = average(10,1,2,3)        # 12.0
a = average(10, values ...)  # 12.0
```

Return value

Return value using the keyword `return` is optional: by default it is returned the last computed value.
The return value can also be a tuple (so returning effectively multiple values):

```
myfunction(a,b) = a*2,b+2
x,y = myfunction(1,2)
```

Multiple-dispatch (aka polymorphism)

The same function can be defined with different number and type of parameters (this is useful when the same kind of logical operation must be performed on different types).
When calling such functions, Julia will pick up the correct one depending from the parameters in the call (by default the stricter version).
These different versions are named "methods" in Julia and, if the function is type-safe, dispatch is implemented at compile time and very fast.
You can inspect the methods of a function with `methods(f)`.

The multiple-dispatch polymorphism is a generalisation of object-oriented run-time polymorphism where the same function name can performs different tasks depending on which is the owner's object's class, i.e. the polymorphism is applied only to a single parameter (it remains true however that OO languages have usually multiple-parameters polymorphism at compile-time).

Templates (type parametrisation)

Functions can be further specified regarding on which types they works with, using templates:

`myfunction(x::T, y::T2, z::T2) where {T <: Number, T2} = 5x + 5y + 5z`

The above function first defines two types, T (a subset of Number) and T2, and then specify each parameter of which of these two types must be.
You can call it with `(1,2,3)` or `(1,2.5,3.5)` as parameter, but not with `(1,2,3.5)` as the definition of `myfunction` constrains that the second and third parameter must be the same type (whatever it is).

Functions as objects

Functions themselves are objects and can be assigned to new variables, returned, or nested. E.g.:

```
f(x) = 2x # define a function f inline
a = f(2)  # call f and assign the return value to a
a = f     # bind f to a new variable name (it's not a deep copy)
a(5)      # call again the (same) function
```

Call by reference / call by value

Parameters given to functions are normally passed by reference.
Functions that do change their arguments have their name, BY CONVENTION, postponed by a `!`, e.g.:

`myfunction!(ref_par, other_pars)` (the parameter that will be changed is by convention the first one)

Anonymous functions

Sometimes we don't need to give a name to a function (e.g. within the `map` function). To define anonymous (nameless) functions we can use the `->` syntax, like this:

```
x -> x^2 + 2x - 1
```

This defines a nameless function that takes an argument, calls it `x`, and produces `x^2 + 2x - 1`. Multiple arguments can be provided using tuples: `(x,y,z) -> x + y + z`

You can still assign an anonymous function to a variable: `f = (x,y) -> x+y`

Broadcast

You can "broadcast" a function to work over each elements of an array (singleton): `myArray = broadcast(i -> replace(i, "x" => "y"), myArray)`. This is equivalent to (note the dot): `myArray = replace.(myArray, Ref("x" => "y"))` (`Ref()` is needed to protect the pair (x,y) from trying to be broadcasted itself as well).

While in the past broadcast was available on a limited number of core functions only, the `f.()` syntax is now automatically available for any function, including the ones you define.

5 - Custom structures

Structures (previously known in Julia as "Types") are, for the most (see later for the difference), what in other languages are called classes, or "structured data": they define the kind of information that is embedded in the structure, that is a set of fields (aka "properties" in other languages), and then individual instances (or "objects") can be produced each with its own specific values for the fields defined by the structure.

They are "composite" types, in the sense that are not made of just a fixed amound of bits as instead "primitive" types.

Some syntax that will be used in the examples:

- `a::B` means "a must be of type B"
- `A<:B` means "A must be a subtype of B".

Defining a structure

```
mutable struct MyOwnType
  property1
  property2::String
end
```

For increasing performances in certain circumstances, you can optionally specify the type of each field, as done in the previous example for `property2.`

You can use templates also in structure declaration:

```
mutable struct MyOwnType{T<:Number}
 property1
 property2::String
 property3::T
end
```

You can omit the `mutable` keyword in front of `struct` when you want to enforce that once an object of that type has been created, its fields can no longer be changed (i.e. , structures are immutable by default. Note that mutable objects -as arrays- remain themselves mutable also in a immutable structure). Although obviously less flexible, immutable structures are much faster.

You can create abstract types using the keyword `abstract type`. Abstract types do not have any field, and objects can not be instantiated from them, although concrete types (structures) can be defined as subtypes of them (an issue to allow abstract classes to have fields is currently open and may be implemented in the future).

Actually you can create a whole hierarchy of abstract types:

```
abstract type MyOwnGenericAbstractType end
abstract type MyOwnAbstractType <: MyOwnGenericAbstractType end
mutable struct AConcreteType <: MyOwnAbstractType
  property1
  property2::String
end
```

Initialising an object and accessing its fields

```
myObject = MyOwnType("something","something",10)
a = myObject.property3 # 10
```

Note that you initialise the object with the values in the order that has been specified in the structure definition.

Implementation of the OO paradigm in Julia

Let's take the following example:

```
struct Person
  myname::String
  age::Int64
end

struct Shoes
   shoesType::String
   colour::String
end

struct Student
   s::Person
   school::String
   shoes::Shoes
end

function printMyActivity(self::Student)
   println("I study at $(self.school) school")
end

struct Employee
   s::Person
   monthlyIncomes::Float64
   company::String
   shoes::Shoes
end

function printMyActivity(self::Employee)
  println("I work at $(self.company) company")
end

gymShoes = Shoes("gym","white")
proShoes = Shoes("classical","brown")

Marc = Student(Person("Marc",15),"Divine School",gymShoes)
MrBrown = Employee(Person("Brown",45),1200.0,"ABC Corporation Inc.", proShoes)

printMyActivity(Marc)
printMyActivity(MrBrown)
```

There are three big elements that distinguish Julia implementation from a pure Object-Oriented paradigm:

1. Firstly, in Julia **you do not associate functions to a type**. So, you do not call a function over a method (`myobj.func(x,y)`) but rather you pass the object as a parameter (`func(myobj, x, y)`);
2. In order to extend the behaviour of any object, Julia doesn't use *inheritance* (**only abstract classes can be inherited**) but rather *composition* (a field of the subtype is of the higher type, allowing access to its fields). I personally believe that this is a bit a limit

in the expressiveness of the language, as the code can not consider directly different concepts of relations between objects (e.g. Person->Student *specialisation*, Person->Arm *composition*, Person->Shoes *weak relation*);
3. **Multiple-inheritance is not supported** (yet).

More on types

Some useful type-related functions:

1. `supertype(MyType)` Returns the parent types of a type
2. `subtypes(MyType)` Lists all children of a type
3. `fieldnames(MyType)` Queries all the fields of a structure
4. `isa(obj,MyType)` Checks if `obj` is of type `MyType`
5. `typeof(obj)` Returns the type of `obj`

This is the complete type hierarchy of `Number in Julia (credits to Wikipedia):`

6 - Input - Output

File reading/writing

File reading/writing is similar to other languages where you first `open` the file, specify the modality (`r` read, `w` write or `a` append) and bind the file to an object, and finally operate on this object and `close()` it when you are done.

A better alternative is however to encapsulate the file operations in a `do` block that closes the file automatically when the block ends:

Write:

```
open("afile.txt", "w") do f  # "w" for writing
  write(f, "test\n")         # \n for newline
end
```

Read the whole file in a single operation:

```
open("afile.txt", "r") do f   # "r" for reading
  filecontent = read(f,String) # attention that it can be used only once. The second t
ime, without reopening the file, read() would return an empty string
  print(filecontent)
end
```

or, reading line by line:

```
open("afile.txt", "r") do f
  for ln in eachline(f)
    println(ln)
  end
end
```

or, if you want to keep track of the line numbers:

```
open("afile.txt", "r") do f
   for (i,ln) in enumerate(eachline(f))
     println("$i $ln")
   end
end
```

Other IO

Some packages that deals with IO are:

- CSV: CSV.jl
- Web stream: HTTP.jl
- Spreadsheets (OpenDocument): OdsIO.jl
- HDF5: HDF5.jl

Some basic examples that use them are available in the DataFrame section.

7 - Managing run-time errors (exceptions)

Run-time errors can be handled with the try/catch block:

```
try
  # ..some dangerous code..
catch
  # ..what to do if an error happens, most likely send an error message using:
  error("My detailed message")
end
```

You can also check for a specific type of exception, e.g.:

```
function volume(region, year)
    try
        return data["volume",region,year]
    catch  e
        if isa(e, KeyError)
          return missing
        end
        rethrow(e)
    end
end
```

8 - Interfacing Julia with other languages

Julia can natively call C and Fortran libraries and, through packages, C++, R (1,2) and Python.
This allows Julia to use the huge number of libraries of these more established languages.

C

mylib.h:

```
#ifndef _MYLIB_H_
#define _MYLIB_H_

extern float iplustwo (float i);
extern float getTen ();
```

mylib.c:

```
float
iplustwo (float i){
 return i+2;
}
```

Compiled with:

- `gcc -o mylib.o -c mylib.c`
- `gcc -shared -o libmylib.so mylib.o -lm -fPIC`

Use in julia with:

```
i = 2
j = ccall((:iplustwo, "[MY FULL PATH]/libmylib.so"), Float32, (Float32,), i)
```

Python

We show here an example with Python. The following code converts an ODS spreadsheet in a Julia DataFrame, using the Python ezodf module (of course this have to be already be available in the local installation of python):

```
using PyCall
using DataFrames

@pyimport ezodf
doc = ezodf.opendoc("test.ods")
nsheets = length(doc[:sheets])
println("Spreadsheet contains $nsheets sheet(s).")
for sheet in doc[:sheets]
    println("---------")
    println("   Sheet name : $(sheet[:name])")
    println("Size of Sheet : (rows=$(sheet[:nrows]()), cols=$(sheet[:ncols]()))")
end

# convert the first sheet to a DataFrame
sheet = doc[:sheets][1]
df_dict = Dict()
col_index = Dict()
for (i, row) in enumerate(sheet[:rows]())
  # row is a list of cells
  # assume the header is on the first row
  if i == 1
      # columns as lists in a dictionary
      [df_dict[cell[:value]] = [] for cell in row]
      # create index for the column headers
      [col_index[j]=cell[:value]  for (j, cell) in enumerate(row)]
      continue
  end
  for (j, cell) in enumerate(row)
      # use header instead of column index
      append!(df_dict[col_index[j]],cell[:value])
  end
end
# and convert to a DataFrame
df = DataFrame(df_dict)
```

The first thing, is to declare we are using PyCall and to `@pyimport` the python module we want to work with. We can then directly call its functions with the usual Python syntax `module.function()`.

Type conversions are automatically performed for numeric, boolean, string, IO stream, date/period, and function types, along with tuples, arrays/lists, and dictionaries of these types.

Other types are instead converted to the generic PyObject type, as it is the case for the `doc` object returned by the module function.
You can then access its attributes and methods with `myPyObject[:attibute]` and `myPyObject[:method]()` respectively.

9 - Metaprogramming

Julia represents its own code as a data structure accessible from the language itself. Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to transform and generate its own code, that is to create powerful macros (the term "metaprogramming" refers to the possibility to write code that write codes that is then evaluated).

Note the difference with C or C++ macros. There, macros work performing textual manipulation and substitution before any actual parsing or interpretation occurs.

In Julia, macros works when the code has been already parsed and organised in a syntax tree, and hence the semantic is much richer and allows for much more powerful manipulations.

Expressions

There are really many way to create an expression:

Colon prefix operator

The colon `:` prefix operator refers to an unevaluated expression. Such expression can be saved and then evaluated in a second moment using `eval(myexpression)`:

```
expr = :(1+2) # save the `1+2` expression in the `expr` expression
eval(expr)    # here the expression is evaluated and the code returns 3
```

Note that $ interpolation (like for strings) is supported:

```
a = 1
expr = :($a+2) # expr is now :(1+2)
```

Quote block

An alternative of the `:([...])` operator is to use the `quote [...] end` block.

Parse a string

Or also, starting from a string (that is, the original representation of source code for Julia):

```
expr = Meta.parse("1+2") # parses the string "1+2" and saves the `1+2` expression in t
he `expr` expression, same as expr = :(1+2)
eval(expr)          # here the expression is evaluated and the code returns 3
```

Use the Exp constructor with a tree

The expression can be also directly constructed from the tree: `expr = Expr(:call, :+, 1, 2)` is equivalent to `expr = parse("1+2")` or `expr = :(1+2)` .

But what there is in an expression? Using `fieldnames(typeof(expr))` or `dump(expr)` we can find that `expr` is an `Expr` object made of two fields: `:head` and `:args` :

- `:head` defines the type of Expression, in this case `:call`
- `:args` is an array of elements that can be symbols, literal values or other expressions. In this case they are `[:+, 1, 1]`

Symbols

The second meaning of the `:` operator is to create symbols, and it is equivalent to the `Symbol()` function that concatenate its arguments to form a symbol:

`a = :foo10` is equal to `a=Symbol("foo",10)`

A useful example to highlight what a symbol is:

```
a = 2;
ex = Expr(:call, :*, a, :b) # ex is equal to :(2 * b). Note that b doesn't even need t
o be defined
a = 0; b = 2;               # no matter what now happens to a, as a is evaluated at th
e moment of creating the expression and the expression stores its value, without any m
ore reference to the variable
eval(ex)                    # returns 4, not 0
```

- To convert a string to symbol: `Symbol("mystring")`
- To convert a Symbol to string: `String(mysymbol)`

Macros

The possibility to represent code into expressions is at the heart of the usage of macros. Macros in Julia take one or more input expressions and return a modified expressions (at parse time). This contrast with normal functions that, at runtime, take the input values (arguments) and return a computed value.

Macro definition

```
macro unless(test_expr, branch_expr)
  quote
    if !$test_expr
      $branch_expr
    end
  end
end
```

Macro call

```
array = [1, 2, 'b']
@unless 3 in array println("array does not contain 3") # here test_expr is "3 in array
" and branch_expr is "println("array does not contain 3")"
```

Like for strings, the `$` interpolation operator will substitute the variable with its content, in this context the expression. So the "expanded" macro will look in this case as:

```
if !(3 in array)
println("array does not contain 3")
end
```

Attention that the macro doesn't create a new scope, and variables declared or assigned within the macro may collide with variables in the scope of where the macro is actually called.

You can review the content of this section in this notebook.

10 - Performances (parallelisation, debugging, profiling..)

Julia is relatively fast when working with `Any` data, but when we restrict a variable to a specific type (or a Union of a few types) it runs with the same order of magnitude of C.

This mean you can code quite quickly and then, only on the parts that constitute a bottleneck, you can concentrate and add specific typing.

Type safety

NOTE: This function in Julia 1.0 works very fast in both the two versions presented here (with `s=0` or `s=0.0`. I leave this discussion to highlight the improvements made in the compiler subset in Julia 1.0, that allow to optimise also type unsafe functions when the set of possible types is limited, like in this case.

Take this function (from the Performance tips of the Julia documentation).

```
function f(n)
   s = 0
   for i = 1:n
       s += i/2
   end
   s
end
```

This is not optimised code, as it is not type-safe. A function is said to be type-safe when its return type depends only from the type of the input, not from its values. Type-safe functions can be optimised by the compiler. In this case, if `n` is <=0, the result is an `Int64` (test it with `typeof(f(0))`), while if `n` is > 0, it is a `Float64`.

The simplest way to make type-safe the function is to declare `s` as `0.0` so to force the result to be always a `Float64`:

```
function f2(n)
   s = 0.0
   for i = 1:n
       s += i/2
   end
   s
end
```

The improvements are huge:

```
    @time f(1000000000) 38.316970 seconds (3.00 G allocations: 44.704 GB, 32.15% gc ti
me)
    @time f2(1000000000) 0.869386 seconds (5 allocations: 176 bytes)
```

Benchmarking

For comparison, the same function can be written in C++, Python and Julia,

g++

```
#include <iostream>
#include <chrono>

using namespace std;
int main() {
    chrono::steady_clock::time_point begin = chrono::steady_clock::now();
    int steps=1000000000;
    double s = 0;
    for (int i=1;i<(steps+1);i++){
       s +=  (i/2.0) ;
    }
    cout << s << endl;
    chrono::steady_clock::time_point end= chrono::steady_clock::now();

    cout << "Time difference (sec) = " << (chrono::duration_cast<std::chrono::microsec
onds>(end - begin).count()) /1000000.0  << endl;
}
```

Non optimised: 2.48 seconds Optimised (compiled with the -O3 switch) : 0.83 seconds

Python

```
from numba import jit
import time

# jit decorator tells Numba to compile this function.
# The argument types will be inferred by Numba when function is called.
@jit
def main():
    steps=1000000000;
    s = 0;
    for i in range(1,steps+1):
       s +=  (i/2.0)
    print(s)

start_time = time.time()
main()
print("--- %s seconds ---" % (time.time() - start_time))
```

Non optimised (wihtout using numba and the @jit decorator): 98 seconds Optimised (using the just in time compilation):0.88 seconds

R

```
f <- function(n){
  # Start the clock!
  ptm <- proc.time()
  s <- 0
  for (i in 1:n){
    s <- s + (i/2)
  }
  print(s)
  # Stop the clock
  proc.time() - ptm
}
```

Non optimised: 287 seconds Optimised (vectorised): the function returns an error (on my 8GB laptop), as too much memory is required to build the arrays!

Human mind

Of course the result is just n*(n+1)/4, so the best programming language is the human mind.. but still compilers are doing a pretty smart optimisation!

Code parallelisation

Julia provides core functionality to parallelise code using processes. These can be even in different machines, where connection is realised trough SSH. Threads instead (that are limited to the same CPU but, contrary to processes, share the same memory) are not yet implemented (as it is much more difficult to "guarantee" safe multi-threads than safe multi-processes).

This notebook shows how to use several functions to facilitate code parallelism:

Debugging

Unfortunately the availability of debugging capabilities like graphical step-by-step in a function or setting breackpoints depends on the versions of Julia. Julia evolved quickly, so many debugging tools previously available doesn't work (yet) in Julia 1.0 (a promising package is Rebugger).

Still, this is somehow mitigated by Julia being a interactive environment, so you can still run your code piece-by-piece.

Here you can find some common operations concerning introspection and debugging:

- Retrieve function signatures: `methods(myfunction)`
- Discover which specific method is used (within the several available, as Julia supports multiple-dispatch aka polymorfism): `@which myfunction(myargs)`
- Discover which fields are part of an object: `fieldnames(myobj`
- Discover which type (loosely a "class" in OO languages) an object instance is: `typeof(a)`
- Get more information about an object: `dump(myobj)`

Profiling

Profiling is the "art" of finding bottlenecks in the code.

A simple way to time a part of the code is to simply type `@time myFunc(args)` (but be sure you ran that function at least once, or you will measure compile time rather than run-time) or `@benchmark myFunc(args)` (from package `BenchmarkTools`)

For more extensive coverage, Julia comes with a integrated statistical profile, that is, it runs every x milliseconds and memorize in which line of code the program is at that moment.

Using this sampling method, at a cost of loosing some precision, profiling can be very efficient, in terms of very small overheads compared to run the code normally.

- Profile a function: `Profile.@profile myfunct()` (best after the function has been already ran once for JIT-compilation).

- Print the profiling results: `Profile.print()` (number of samples in corresponding line and all downstream code; file name:line number; function name;)
- Explore a chart of the call graph with profiled data: `ProfileView.view()` (from package `ProfileView`, not yet available to Julia 1 at time of writing).
- Clear profile data: `Profile.clear()`

11 - Developing Julia packages

Patching other people packages:

- pkg> `develop pkgName`
- [patch & commit]
- `using PkgDev; PkgDev.submit(pkgName)`

Develop your own project and publish a new version

- pkg> `develop git@github.com:userName/pkgName.jl.git` to checkout master from GitHub
- [...work on the project..]
- `PkgDev.tag(pkg, v"0.X.X")`
- `PkgDev.publish(pkg)`

or (much better) use the package attobot that automatise the workflow (after you installed attobot on your GitHub repository, just create a new GitHub release in order to spread it to the Julia package ecosystem).

In case of problems: http://stackoverflow.com/questions/9646167/clean-up-a-fork-and-restart-it-from-the-upstream

- Testing a package: `Pkg.test("pkg")`

It is a good practice to document your own functions. You can use triple quoted strings (""") just before the function to document and use Markdown syntax in it. The Julia documentation recommends that you insert a simplified version of the function, together with an `Arguments` and an `Examples` sessions.
For example, this is the documentation string of the `ods_readall` function within the `OdsIO` package:

```
"""
    ods_readall(filename; <keyword arguments>)

Return a dictionary of tables|dictionaries|dataframes indexed by position or name in t
he original OpenDocument Spreadsheet (.ods) file.

# Arguments
* `sheetsNames=[]`: the list of sheet names from which to import data.
* `sheetsPos=[]`: the list of sheet positions (starting from 1) from which to import d
ata.
* `ranges=[]`: a list of pair of touples defining the ranges in each sheet from which
to import data, in the format ((tlr,trc),(brr,brc))
* `innerType="Matrix"`: the type of the inner container returned. Either "Matrix", "Di
ct" or "DataFrame"

# Notes
* sheetsNames and sheetsPos can not be given together
* ranges is defined using integer positions for both rows and columns
* individual dictionaries or dataframes are keyed by the values of the cells in the fi
rst row specified in the range, or first row if `range` is not given
* innerType="Matrix", differently from innerType="Dict", preserves original column ord
er, it is faster and require less memory
* using innerType="DataFrame" also preserves original column order

# Examples
``julia
julia> outDic  = ods_readall("spreadsheet.ods";sheetsPos=[1,3],ranges=[((1,1),(3,3)),(
(2,2),(6,4))], innerType="Dict")
Dict{Any,Any} with 2 entries:
  3 => Dict{Any,Any}(Pair{Any,Any}("c",Any[33.0,43.0,53.0,63.0]),Pair{Any,Any}("b",Any
[32.0,42.0,52.0,62.0]),Pair{Any,Any}("d",Any[34.0,44.0,54.…
  1 => Dict{Any,Any}(Pair{Any,Any}("c",Any[23.0,33.0]),Pair{Any,Any}("b",Any[22.0,32.0
]),Pair{Any,Any}("a",Any[21.0,31.0]))
``
"""
```

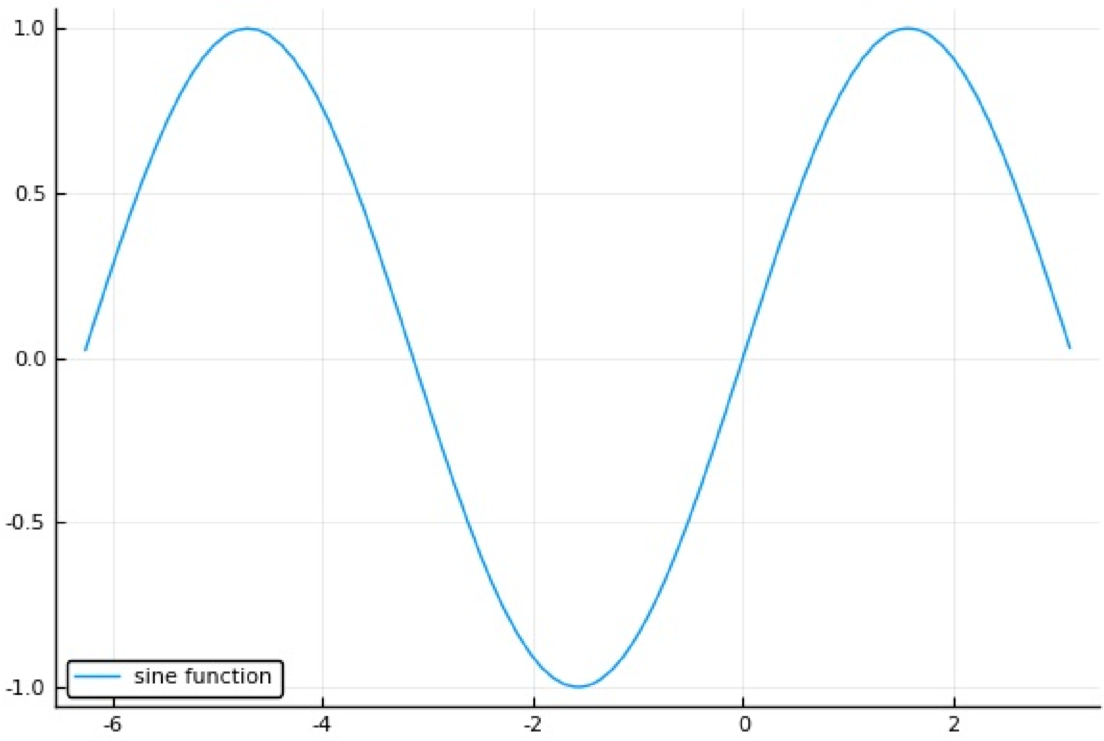
Plotting

Plotting in julia can be obtained using a specific plotting package (e.g. Gadfly, Winston) or, as I prefer, use the Plots package that provide a unified API to several supported backends

Backends are chosen running `chosenbackend()` (that is, the name of the corresponding backend package, but written all in lower case) before calling the `plot` function. You need to install at least one backend before being able to use the `Plots` package. My preferred one is PlotlyJS (a julia interface to the plotly.js visualization library.), but you may be interested also in PyPlot (that use the excellent python matplotlib **VERSION 2**).

For example:

```
Pkg.add("Plots")
Pkg.add("PyPlot.jl") # or  Pkg.add("PlotlyJS")
using Plots
pyplot()             # or plotlyjs()
plot(sin, -2pi, pi, label="sine function")
```



Temporary note: as of writing, the `plotlyjs` backend doesn't work. `pyplot` backend works, but require the user to manually add the `PyCall` and `LaTeXStrings` packages.

Attention not to mix using different plotting packages (e.g. `Plots` and one of its backends). I had troubles with that. If you have already imported a plot package and you want to use an other package, always restart the julia kernel (this is not necessary, and it is one of the advantage, when switching between different bakends of the `Plots` package).

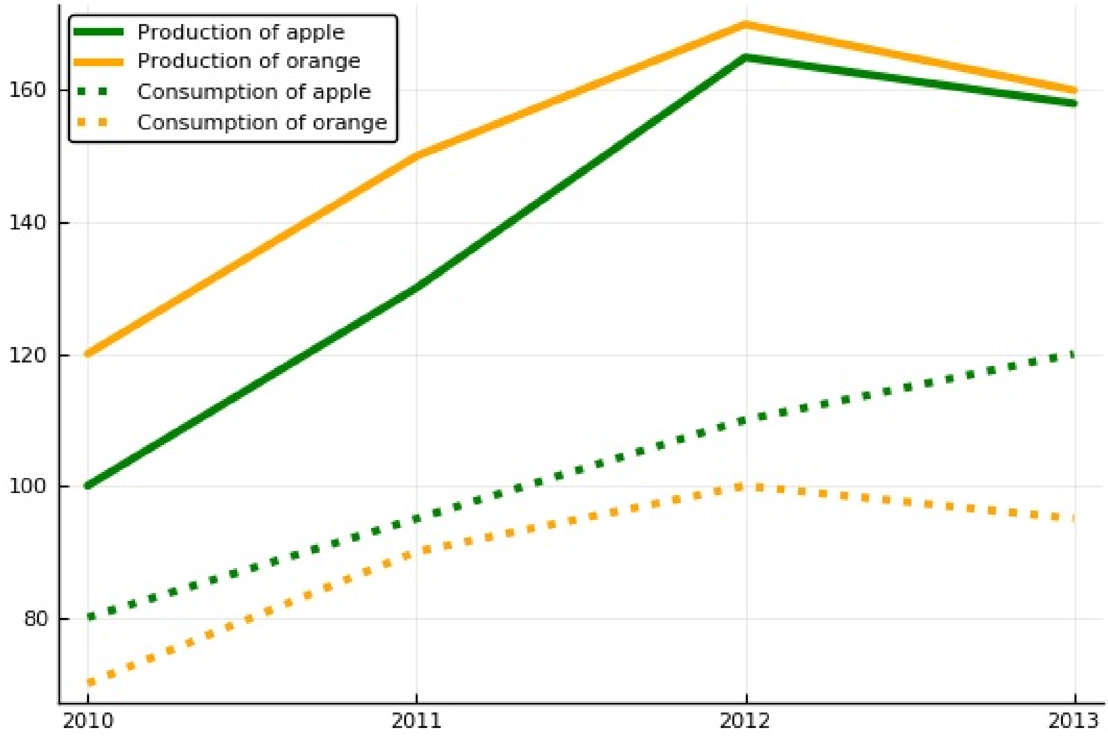
You can find some useful documentation on `Plots` backends:

- Which backend to choose ?
- Charts and attributes supported by the various backends

Plotting multiple groups of series from a DataFrame

The following example uses StatPlots in order to work directly on DataFrames (rather than on arrays). Passing the dataFrame as first argument of the `@df` macro, you can access its columns by name and split the overall series using a third column.

```
using DataFrames, Plots, StatPlots
df = DataFrame(
  fruit       = ["orange","orange","orange","orange","apple","apple","apple","apple"],
  year        = [2010,2011,2012,2013,2010,2011,2012,2013],
  production  = [120,150,170,160,100,130,165,158],
  consumption = [70,90,100,95,   80,95,110,120]
)
pyplot()
mycolours = [:green :orange] # note that the serie is piled up alphabetically
fruits_plot = @df df plot(:year, :production, group=:fruit, linestyle = :solid, linewi
dth=3, label= reshape(string.("Production of ", sort(unique(:fruit))),(1,:)), color=my
colours)
@df df plot!(:year, :consumption, group=:fruit, linestyle = :dot, linewidth=3, label =
"Consumption of " .* reshape(sort(unique(:fruit)),(1,:)), color=mycolours)
```



The first call to `plot()` create a new plot. Calling `plot!()` modify instead the plot that is passed as first argument (if none, the latest plot is modified)

Printing area charts

Use the `fill(fillrange,fillalpha,fillcolor)` attribute, e.g. `fill = (0, 0.5, :blue)` .

Saving

To save the figure just call one of the following:

```
savefig("fruits_plot.svg")
savefig("fruits_plot.pdf")
savefig("fruits_plot.png")
```

A note on saving with the plotlyjs backend

Julia 1.0 note: As `plotlyjs` still doesn't work on Julia 1.0, this subsection needs still to be checked!

Only for the `plotlyjs` backend, you need to first install the `Rsvg` package (`Pkg.add("Rsvg")`) (or rely to the saving button on the widget). Still there will be some problems:

- svg: an html file with embedded svg is actually created, not a svg file
- pdf: this currently doesn't work in Julia 0.6
- png: this works

DataFrames

Dataframes

Julia has a library to handle tabular data, in a way similar to R or Pandas dataframes. The name is, no surprises, DataFrames. The approach and the function names are similar, although the way of actually accessing the API may be a bit different.
For complex analysis, DataFramesMeta adds some helper macros.

Documentation:

- DataFrames: http://juliadata.github.io/DataFrames.jl/stable/, https://en.wikibooks.org/wiki/Introducing_Julia/DataFrames
- DataFramesMeta: https://github.com/JuliaStats/DataFramesMeta.jl
- Stats in Julia in general: http://juliastats.github.io/

Install and import the library

- Install the library: `Pkg.add(DataFrames)`
- Load the library: `using DataFrames`

Create a df or load data:

- From a table defined in code:

```
using CSV
supplytable = CSV.read(IOBuffer("""
prod Epinal Bordeaux Grenoble
Fuelwood 400 700 800
Sawnwood 800 1600 1800
Pannels 200 300 300
"""),delim=" ") # an option to ignore repeated delimiters so to allow a better formatt
ing is coming to CSV.jl
```

- Read a CSV file: `myData = CSV.read(file; delim=';', missingstring="NA", delim=";", decimal=',')` (use `CSV.read(file; delim='\t')` for tab delimited files)

If a column has in the first top rows used by type-autorecognition only missing values, but then has non-missing values in subsequent rows, an error may appear. The trick is to manually specify the column value with the `type` parameter (Vector or Dictionary, e.g. `types=Dict("freeDim" => Union{Missing,Int64})`)

- From a stream, use the package `HTTP` :

```
using DataFrames, HTTP, CSV
resp = HTTP.request("GET", "https://data.cityofnewyork.us/api/views/kku6-nxdu/rows
.csv?accessType=DOWNLOAD")
df = CSV.read(IOBuffer(String(resp.body)))
```

- From a OpenDocument Spreadsheet file (OpenOffice, LibreOffice, MS Excel and others): Use the `OdsIO` package together with the `retType="DataFrame"` argument: `df = ods_read("spreadsheet.ods";sheetName="Sheet2",retType="DataFrame",range=((tl_row,tl_col),(br_row,br_col)))`
- Crate a df from scratch:

```
df = DataFrame(
colour = ["green","blue","white","green","green"],
shape  = ["circle", "triangle", "square","square","circle"],
border = ["dotted", "line", "line", "line", "dotted"],
area   = [1.1, 2.3, 3.1, missing, 5.2])
```

- Create an empty df: `df = DataFrame(A = Int64[], B = Float64[])`
- Convert from a Matrix of data and a vector of column names: `df = DataFrame([[mat[:,i]...] for i in 1:size(mat,2)], Symbol.(headerstrs))`
- Convert from a Matrix with headers in the first row: `df = DataFrame([[mat[2:end,i]...] for i in 1:size(mat,2)], Symbol.(mat[1,:]))`

Get insights about your data:

- `head(df)`
- `showall(df)`
- `tail(df)`
- `describe(df)`
- `unique(df[:fieldName])` or `[unique(df[i]) for i in names(df)]`
- `names(df)` returns array of column names
- `colwise(eltype, df)` returns an array of column types
- `size(df)` (r,c), `size(df)[1]` (r), `size(df)[2]` (c)
- `ENV["LINES"] = 60` change the default number of lines before the content is truncated (default 30). Also COLUMNS. May not work with terminal.

- `for r in eachrow(df)` iterates over each row

Column names are Julia symbols. To programmatically compose a column name you need hence to use the Symbol(String) constructor, e.g.: `df[Symbol("value_",0)] = "aa"`

Edit data

- Replace values based to a dictionary : `mydf[:col1] = map(akey->myDict[akey], mydf[:col1])` (the original data to replace can be in a different column or a totally different DataFrame
- Concatenate (string) values for several columns to create the value a new column: `df[:c] = df[:a] .* " " .* df[:b]`
- To compute the value of a column based of other columns you need to use elementwise operations using the dot, e.g. `df[:a] = df[:b] .* df[:c]` (note that the equal sign doesn't have the dot.. but if you have to make a comparison, the `==` operator wants also the dot, i.e. `.==`)
- Append a row: `push!(df, [1 2 3])`
- Delete a given row: use `deleterows!(df,rowIdx)` or just copy a df without the rows that are not needed, e.g. `df2 = df[[1:(i-1);(i+1):end],:]`
- Empty a dataframe: `df = similar(df,0)`

Filter (aka "selection" or "query")

- Filter by value, based on a field being in a list of values using boolean selection trough list comprehension: `df[ [i in ["blue","green"] for i in df[:colour]], :]`
- Combined boolean selection: `df[([i in ["blue","green"] for i in df[:colour]] .> 0) .& (df[:shape] .== "triangle"), :]` (the dot is needed to vectorize the operation. Note the usage of the bitwise and the single ampersand).
- Filter using `@where` (`DataFrameMeta` package): `@where(df, :x .> 2, :y .== "a") # the two expressions are "and-ed"`. If the column name is stored in a variable, you need to wrap it using the `cols()` function, e.g. `col = Symbol("x"); @where(df, cols(col) .> 2)`
- Change a single value by filtering columns: `df[ (df[:product] .== "hardWSawnW") .& (df[:year] .== 2010) , :consumption] = 200`
- Filter based on initial pattern: `filteredDf = df[startswith.(df[:field],pattern),:]`
- A benchmark note: using `@with()` or boolean selection is ~ the same, while "querying" an equivalent Dict with categorical variables as tuple keys is around ~20% faster than querying the dataframe.
- A further (and perhaps more elegant, although longer) way to query a DataFrame is to use the `Query` package. The first example above let you select a subsets of both rows and columns, the second one highlight instead how you can mix multiple selection

criteria:

```
dfOut = @from i in df begin
           @where i.col1 > 1
           @select {aNewColName=i.col1, i.col3}
           @collect DataFrame
        end
 dfOut = @from i in df begin
            @where i.value != 1 && i.cat1 in ["green","pink"]
            @select i
            @collect DataFrame
        end
```

Edit structure

- Delete columns by name: `delete!(df, [:col1, :col2])`
- Rename columns: `names!(df, [:c1,:c2,:c3])` (all) `rename!(df, Dict(:c1 => :neCol))` (a selection)
- Change column order: `df = df[[:b, :a]]`
- Add an "id" column (useful for unstacking): `df[:id] = 1:size(df, 1)` # this makes it easier to unstack
- Add a Float64 column (all filled with NA by default): `df[:a] = Array{Union{Missing,Float64},1}(missing,size(df,1))`
- Add a column based on values of other columns: `df[:c] = df[:a]+df[:b]` (as alternative use map: `df[:c] = map((x,y) -> x + y, df[:a], df[:b])`)
- Insert a column at a position i: `insert!(df, i, [colContent], :colName)`
- Convert columns:
  - from Int to Float: `df[:A] = convert(Array{Float64,1},df[:A])`
  - from Float to Int: `df[:A] = convert(Array{Int64,1},df[:A])`
  - from Int (or Float) to String: `df[:A] = map(string, df[:A])`
  - from String to Float: `string_to_float(str) = try parse(Float64, str) catch; return(missing) end; df[:A] = map(string_to_float, df[:A])`
  - from Any to T (including String, if the individual elements are already strings): `df[:A] = convert(Array{T,1},df[:A])`
- You can "pool" specific columns in order to efficiently store repeated categorical variables with `categorical!(df, [:A, :B])`. Attention that while the memory decrease, filtering with categorical values is not quicker (indeed it is a bit slower). You can go back to normal arrays wih `collect(df[:A])`.

Merge/Join/Copy datasets

- Concatenate different dataframes (with same structure): `` `df = vcat(df1,df2,df3) `` or

`df = vcat([df1,df2,df3]...)` (note the three dots at the end, i.e. the splat operator).
- Join dataframes horizontally: `fullDf = join(df1, df2, on = :commonCol)`
- Copy the structure of a DataFrame (to an empty one): `df2 = similar(df1, 0)`

Manage Missing values

Starting from Julia 1, `Missings` type is defined in core (with some additional functionality still provided by the additional package `Missings.jl`). At the same time, a `DataFrame` changes from being a collection of `DataArrays` to a collection of standard `Arrays`, eventually of type `Union{T,Missing}` if missing data is present.

- The missing value is simply `missing`
- Remove missing values with: `a = collect(skipmissing(df[:col1]))` (returns an `Array`) or `b = dropmissing(df[[:col1,:col2]])` (returns a `DataFrame` even for a single column)
- `dropmissing!(df)` (in both its version with or without question mark) and `completecases(df)` select only rows without missing values. The first returns the skimmed `DataFrame`, while the second return a boolean array, and you can also specify on which columns you want to limit the application of this filter `completecases(df[[:col1,:col2]])`. You can then get the df with `df2 = df[completecases(df[[:col1,:col2]]),:]`)
- Within an operation (e.g. a sum) you can use `dropmissing()` in order to skip `missing` values before the operation take place.
- Remove missing values on all string and numeric columns: `[df[ismissing.(df[i]), i] = 0 for i in names(df) if Base.nonmissingtype(eltype(df[i])) <: Number]` `[df[ismissing.(df[i]), i] = "" for i in names(df) if Base.nonmissingtype(eltype(df[i])) <: String]`
- To make comparison (e.g. for boolean selection or within the `@where` macro in `DataFramesMeta`) where missing values could be present you can use `isequal.(a,b)` to NOT propagate the missing (i.e. `isequal("green",missing)` is true) or the confrontation operator (`==`)to preserve missingness (i.e. `"green" == missing` is neither `true` nor `false` but `missing`)
- Count the `missing` values: `nMissings = length(findall(x -> ismissing(x), df[:col]))`

Split-Apply-Combine strategy

The DataFrames package supports the Split-Apply-Combine strategy through the `by` function, which takes in three arguments: (1) a DataFrame, (2) a column (or columns) to split the DataFrame on, and (3) a function or expression to apply to each subset of the DataFrame.

The function can return a value, a vector, or a DataFrame. For a value or vector, these are merged into a column along with the `cols` keys. For a DataFrame, `cols` are combined along columns with the resulting DataFrame. Returning a DataFrame is the clearest because it allows column labelling.

`by` function can take the function as first argument, so to allow the usage of do blocks. Inside, it uses the groupby() function, as in the code it is defined as nothing else than:

```
by(d::AbstractDataFrame, cols, f::Function) = combine(map(f, groupby(d, cols)))
by(f::Function, d::AbstractDataFrame, cols) = by(d, cols, f)
```

Aggregate

Aggregate by several fields:

- `aggregate(df, [:field1, :field2], sum)`

  Attention that all categorical fields have to be included in the list of fields over which to aggregate, otherwise Julia will try to compute a sum also over them (but them being string, it will raice an error) instead of just ignoring them.

  The workaround is to remove the fields you don't want before doing the operation.

- Alternatively (and without the problem of the previous point):

```
by(df, [:catfield1,:catfield2]) do df
  DataFrame(m = sum(df[:valueField]))
end
```

Compute cumulative sum by categories

- Manual method (very slow):

```
df = DataFrame(region=["US","US","US","US","EU","EU","EU","EU"],
             year = [2010,2011,2012,2013,2010,2011,2012,2013],
             value=[3,3,2,2,2,2,1,1])
df[:cumValue] = copy(df[:value])
[r[:cumValue] = df[(df[:region] .== r[:region]) .& (df[:year] .== (r[:year]-1)),:c
umValue][1] + r[:value]  for r in eachrow(df) if r[:year] != minimum(df[:year])]
```

- Using by and the split-apply-combine strategy (fast):

```
using DataFramesMeta, DataArrays, DataFrames
df = DataFrame(region  = ["US","US","US","US","EU","EU","EU","EU"],
             product = ["apple","apple","banana","banana","apple","apple","banana"
,"banana"],
             year    = [2010,2011,2010,2011,2010,2011,2010,2011],
             value   = [3,3,2,2,2,2,1,1])
df[:cumValue] = copy(df[:value])
by(df, [:region,:product]) do dd
  dd[:cumValue] = cumsum(dd[:value])
     return
end
```

- Using @linq (from DataFramesMeta) and the split-apply-combine strategy (fast):

```
using DataFramesMeta, DataFrames
df = DataFrame(region  = ["US","US","US","US","EU","EU","EU","EU"],
             product = ["apple","apple","banana","banana","apple","apple","banana"
,"banana"],
             year    = [2010,2011,2010,2011,2010,2011,2010,2011],
             value   = [3,3,2,2,2,2,1,1])
df = @linq df |>
groupby([:region,:product]) |>
transform(cumValue = cumsum(:value))
```

- Using groupby (fast):

```
using DataFramesMeta, DataArrays, DataFrames
df = DataFrame(region  = ["US","US","US","US","EU","EU","EU","EU"],
             product = ["apple","apple","banana","banana","apple","apple","banana"
,"banana"],
             year    = [2010,2011,2010,2011,2010,2011,2010,2011],
             value   = [3,3,2,2,2,2,1,1])
df[:cumValue] = 0.0
for subdf in groupby(df,[:region,:product])
  subdf[:cumValue] = cumsum(subdf[:value])
end
```

Pivot

Stack

Move columns to rows of a "variable" column, i.re. moving from wide to long format. For `stack(df,[cols])` you have to specify the column(s) that have to be stacked, for `melt(df,[cols])` at the opposite you specify the other columns, that represent the id columns that are already in stacked form.

Finally `stack(df)` - without column names - automatically stack all float columns.
Note that the stacked columns are inserted as data in a "variable" column (with names of the variables not strings but symbols) and the corresponding values in a "column" value.

```
df = DataFrame(region = ["US","US","US","US","EU","EU","EU","EU"],
               product = ["apple","apple","banana","banana","apple","apple","banana","
banana"],
               year = [2010,2011,2010,2011,2010,2011,2010,2011],
               produced = [3.3,3.2,2.3,2.1,2.7,2.8,1.5,1.3],
               consumed = [4.3,7.4,2.5,9.8,3.2,4.3,6.5,3.0])
long1 = stack(df,[:produced,:consumed])
long2 = melt(df,[:region,:product,:year])
long3 = stack(df)
long1 == long2 == long3 # true
```

Unstack

You can specify the dataframe, the column name which content will become the row index (id variable), the column name with content will become the name of the columns (column variable names) and the column name containing the values that will be placed in the new table (column values):

`widedf = unstack(longdf, [:ids], :variable, :value)`

Alternatively you can omit the `:id` parameter and all the existing column except the one defining column names and the one defining column values will be preserved as index (row) variables:

`widedf = unstack(longdf, :variable, :value)`

Sorting

`sort!(df, cols = (:col1, :col2), rev = (false, false))` The (optional) reverse order parameter (rev) must be a turple of the same size as the cols parameter

Use LAJuliaUtils.jl

You can use (my own utility module) `LAJuliaUtils.jl` in order to Pivot and optionally filter and sort in a single function in a spreadsheet-like Pivot Tables fashion. See the relevant section.

Export your data

Export to CSV

`CSV.write("file.csv", df, delim = ';', header = true)` (from package `CSV`)

Export to ods (OpenDocument Spreadsheet file - OpenOffice, LibreOffice, MS Excel and others)

Use the `OdsIO` package:

`ods_write("spreadsheet.ods",Dict(("MyDestSheet",3,2)=>myDf)))`

Export to Dict

This export to a dictionary where the keys are the unique elements of a df column and the values are the splitted dataframes:

```
vars = Dict{String,DataFrame}()
[vars[x] = @where(df, :varName .== x) for x in unique(df[:varName])]
[delete!(vars[k], [:varName]) for k in keys(vars)]
```

Export to hdf5 format

To use hdf5 with the `HDF5` package, some systems may require system-wide hdf5 binaries, e.g. in Ubuntu linux `sudo apt-get install hdf5-tools.`

`h5write("out.h5", "mygroup/myDf", convert(Array, df[:,[list_of_cols]))`

The HDF5 package doesn't yet support directly dataframes, so you need first to export them as Matrix (a further limitation is that it doesn't accept a matrix of Any type, so you may want to export a DataFrame in two pieces, the string and the numeric columns separatly). You can read back the data with `data = h5read("out.h5", "mygroup/myDf")` .

``

JuMP

`` `JuMP` is an algebraic modelling language for mathematical optimisation problems, similar to GAMS, AMPL or Pyomo.

It is solver-independent. It supports also non-linear solvers, providing them with the Gradient and the Hessian.

This notebook provides a commented implementation in JuMP of the classical transport problem found in the GAMS tutorial:

Note: While JuMP seems to work in Julia 0.7/1.0, many solver interfaces (including the ones used in the above notebook) don't jet work with Julia versions above 0.6.

SymPy

SymPy

`` `SymPy` is a wrapper to the Python SymPy library for symbolic computation: solve equations (or system of equations), simplify them, find derivates or integrals...

An overview of its capabilities can be found on the following notebook:

http://nbviewer.jupyter.org/github/sylvaticus/juliatutorial/blob/master/assets/Symbolic computation.ipynb

Some additional notes to that notebook:

- You can plot a function that includes symbols, e.g.: `plot(2x,0,1)` plots `y=2x` in the [0,1] range
- For the infinity symbol use either `oo` or `Inf` (eventually with + or -)

Other Mathematical packages

- Numerical integration of definite integrals (univariate): (QuadGK Package: `quadgk(x->2x,0,2)`)

Weave

`` `Weave` allows to produce dynamic documents where the script that produce the output is embedded directly in the document, with optionally only the output rendered.

Save the document below in a file with extension jmd (e.g. testWeave.jmd)

```
---
title : Test of a document with embedded Julia code and citations
date : 5th September 2018
bibliography: biblio.bib
---


# Section 1 (leave two rows from document headers)

This is a strong affermation that needs a citation [see @Lecocq:2011, pp. 33-35; @Caur
la:2013b, ch. 1].

@Lobianco:2016b [pp. 8] affirms something else.

## Subsection 1.1

This should print a plot. Note that I am not showing the source code in the final PDF:

```{julia;echo=false}
using Plots
pyplot()
plot(sin, -2pi, pi, label="sine function")
```

Here instead I will put in the PDF both the script source code and the output:

```
using DataFrames
df = DataFrame(
         colour = ["green","blue","white","green","green"],
         shape  = ["circle", "triangle", "square","square","circle"],
         border = ["dotted", "line", "line", "line", "dotted"],
         area   = [1.1, 2.3, 3.1, missing, 5.2]
    )
df
```

Note also that I can refer to variables defined in previous chunks (or "cells", following Jupyter terminology):

```
df[:colour]
```
```

Subsubsection

For a much more complete example see the Weave documentation.

References

```

You can then "compile" the document (from within Julia) with:

```
using Weave; weave("testWeave.jmd", out_path = :pwd, doctype = "pandoc2pdf")
```

To obtain the following pdf:

Test of a document with embedded Julia code and citations

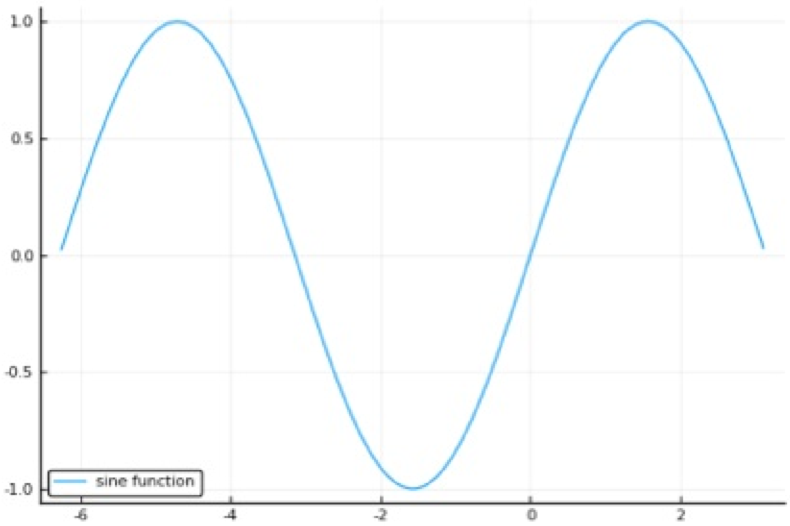
5th September 2018

## Section 1 (leave two rows from document headers)

This is a strong affermation that needs a citation (see Lecocq et al. 2011, 33–35; Caurla et al. 2013, ch. 1). Lobianco et al. (2016, 8) affirms something else.

### Subsection 1.1

This should print a plot. Note that I am not showing the source code in the final PDF:




Here instead I will put in the PDF both the script source code and the output:
```

```
using DataFrames
df = DataFrame(
         colour = ["green","blue","white","green","green"],
         shape  = ["circle", "triangle", "square","square","circle"],
         border = ["dotted", "line", "line", "line", "dotted"],
         area   = [1.1, 2.3, 3.1, missing, 5.2]
    )
df
```

```
5×4 DataFrame
 Row  colour  shape     border  area

 1    green   circle    dotted  1.1
 2    blue    triangle  line    2.3
 3    white   square    line    3.1
 4    green   square    line    missing
 5    green   circle    dotted  5.2
```

Note also that I can refer to variables defined in previous chunks (or "cells", following Jupyter terminology):

```
df[:colour]
```

```
5-element Array{String,1}:
 "green"
 "blue"
 "white"
 "green"
 "green"
```

Subsubsection

For a much more complete example see the Weave documentation.

References

Caurla, S., F. Lecocq, P. Delacote, and A. Barkaoui. 2013. "Stimulating Fuelwood Consumption Through Public Policies: An Assessment of Economic and Resource Impacts Based on the French Forest Sector Model." *Energy Policy* 63: 338–47.

Lecocq, F., S. Caurla, P. Delacote, A. Barkaoui, and A. Sauquet. 2011. "Paying for Forest Carbon or Stimulating Fuelwood Demand? Insights from the French Forest Sector Model." *Journal of Forest Economics* 17 (2): 157–68.

Lobianco, Antonello, Sylvain Caurla, Philippe Delacote, and Ahmed Barkaoui. 2016. "Carbon Mitigation Potential of the French Forest Sector Under Threat of Combined Physical and Market Impacts Due to Climate Change." *Journal of Forest Economics* 23: 4–26. https://doi.org/10.1016/j.jfe.2015.12.003.

In Ubuntu Linux (but most likely also in other systems), weave needs pandora and LaTeX (`texlive-xetex`) already installed in the system.
If you use Ununtu, the version of pandora in the official repositories is too old. Use instead the deb available in https://github.com/jgm/pandoc/releases/latest .

LAJuliaUtils

`` `LAJuliaUtils` is my personal repository for utility functions, mainly for dataframes.

As it is not a registered Julia package, use it with : `add https://github.com/sylvaticus/LAJuliaUtils.jl.git`

It implements the following functions:

- `addCols!(df, colsName, colsType)` - Adds to the DataFrame empty column(s) colsName of type(s) colsType
- `pivot(df::AbstractDataFrame, rowFields, colField, valuesField; <kwd args>)` - Pivot and optionally filter and sort in a single function
- `customSort!(df, sortops)` - Sort a DataFrame by multiple cols, each specifying sort direction and custom sort order
- `toDict(df, dimCols, valueCol)` - Convert a DataFrame in a dictionary, specifying the dimensions to be used as key and the one to be used as value.
- `toDataFrame(t)` - Convert an IndexedTable NDSparse table to a DataFrame, maintaining column types and (eventual) column names.
- `defEmptyIT(dimNames, dimTypes; <kwd args>)` - Define empty IndexedTable(s) with the specific dimension(s) and type(s).
- `defVars(vars, df, dimensions;<kwd args>)` - Create the required IndexedTables from a common DataFrame while specifing the dimensional columns.
- `fillMissings!(vars, value, dimensions)` - For each values in the specified dimensions, fill the values of IndexedTable(s) without a corresponding key.

In particular the pivot() function accepts the following arguments:

- `df::AbstractDataFrame` : the original dataframe, in stacked version (dim1,dim2,dim3... value)
- `rowFields` : the field(s) to be used as row categories (also known as IDs or keys)
- `colField::Symbol` : the field containing the values to be used as column headers
- `valuesField::Symbol` : the column containing the values to reshape
- `ops=sum` : the operation(s) to perform on the data, default on summing them
- `filter::Dict` : an optional filter, in the form of a dictionary of column_to_filter => [list of ammissible values]
- `sort` : optional row field(s) to sort

Note: I didn't yet released `LAJuliaUtils` for Julia 1.0, as some minor functionalities (not actually needed for the `pivot()` function) require `IndexedTables` ported to Julia 1.0. But if you need it, open an issue and I'll release a Julia 1.0 version with the code that doesn't depend to `IndexedTables`.

IndexedTables

`` `IndexedTables` are DataFrame-like data structure that, working with tuples dictionaries, are in my experience much faster to perform select operations.

Unfortunatly, the package is in the process to move from the Named Tuple in NamedTupèles.jl package to the new one in core, and it doesn't yet works in Julia 0.7/1.0.
The following code runs in Julia 0.6.

Create an IndexedTable

The constructor for IndexedTable takes two parts, a Column for the index (dimensions) part and one for the value part. Both can be named or not:

```
 tnamed = Table(
    Columns(
        param  = String["price","price","price","price","waterContent","waterContent"]
,
        item   = String["banana","banana","apple","apple","banana", "apple"],
        region = Union{String,DataArrays.NAtype}["FR","UK","FR","UK",NA,NA]
    ),
    Columns(
       value2000 = Float64[2.8,2.7,1.1,0.8,0.2,0.7],
       value2010 = Float64[3.2,2.9,1.2,0.8,0.2,0.8],
    )
)

tnormal = Table(
    Columns(
        String["price","price","price","price","waterContent","waterContent"],
        String["banana","banana","apple","apple","banana", "apple"],
        Union{String,DataArrays.NAtype}["FR","UK","FR","UK",NA,NA]
    ),
    Columns(
       Float64[2.8,2.7,1.1,0.8,0.2,0.7],
       Float64[3.2,2.9,1.2,0.8,0.2,0.8]
    )
)

tsingle = Table(
    Columns(
        String["price","price","price","price","waterContent","waterContent"],
        String["banana","banana","apple","apple","banana", "apple"],
        Union{String,DataArrays.NAtype}["FR","UK","FR","UK",NA,NA]
    ),
    Float64[2.8,2.7,1.1,0.8,0.2,0.7]
)
```

An alternative way to construct a `Column` is to use a serie of Arrays and the optional `names` paramenter:

```
dimValues = [Array{String,1}(),Array{Int,1}()]
s = Columns(dimValues..., names=[:region,:year])
```

Note that using `Columns()` will always build a tuple, even for a single column. If you want a single column (unnamed!) use directly the `Array` in the constructor, like in the tsingle example.

Edit values

Assign/change values: `t["price","banana","FR] = 2.7, 3.2`

Pipe

The `Pipe` package allows you to improve the Pipe operator `|>` in Julia Base.

Chaining (or "piping") allows to string together multiple function calls in a way that is at the same time compact and readable. It avoids saving intermediate results without having to embed function calls within one another.

With the chain operator `|>` instead, the code to the right of `|>` operates on the result from the code to the left of it. In practice, what is on the left becomes the argument of the function call(s) that is on the right.

Chaining is very useful in data manipulation. Let's assume that you want to use the following (silly) functions operate one after the other on some data and print the final result:

```
add6(a) = a+6; div4(a) = 4/a;
```

You could either introduce temporary variables or embed the function calls:

```
a = 2; b = add6(a); c = div4(b); println(c) # 0.5 println(div4(add6(a)))
```

With piping you can write instead:

```
a |> add6 |> div4 |> println
```

Pipes in Base are very limited, in the sense that support only functions with one argument and only a single function at a time.

Conversely, the `Pipe` package together with the `@pipe` macro hoverrides the |> operator allowing you to use functions with multiple arguments (and there you can use the underscore character " `_` " as placeholder for the value on the LHS) and multiple functions, e.g.:

```
addX(a,x) = a+x; divY(a,y) = a/y @pipe a |> addX(_6) + divY(4,_) |> println # 10.0
```

Note that, as in the basic pipe, functions that require a single argument and this is provided by the piped data, don't need parenthesis.