Julia - first contact

Resources

- **Homepage**
- **Documentation**
- **Cheat Sheet**
- **WikiBook**
- **7 Julia Gotchas and How to Handle Them**

Hint for starting

Use the **Cheat Sheet** to see a compact and rather comprehensive list of basic things Julia. This notebook tries to discuss some concepts behind Julia.

Open Source

- Julia is an Open Source project started at MIT
- Julia itself is distributed under an **MIT license**
  - packages often have different licenses
- Development takes place on **github**
- As of October 2020, more than 1000 contributors to the code
- The Open Source paradigm corresponds well to the fundamental requirement that scientific research should be **transparent and reproducible**

How to install and run Julia

- Installation:
  - **Download** from julialang.org (recommended by Julia creators)
  - Installation via system package manager (yast, apt-get, homebrew...)
- Running:
  - From command line: Edit source code in any editor
  - Julia plugin of Visual Studio code editor
  - Pluto notebooks in the browser
  - Jupyter notebooks in the browser

REPL: Read-Evaluation-Print-Loop

Start REPL by calling `julia` in terminal

REPL modes:

- **Default mode:** `julia>` prompt. Type backspace in other modes to enter default mode.
- **Help mode:** `help?>` prompt. Type `?` to enter help mode. Search via `?search_term`
- **Shell mode:** `shell>` prompt. Type `;` to enter shell mode.
- **Package mode:** `Pkg>` prompt. Type `]` to enter package mode.

Helpful commands in REPL

- `quit()` or `Ctrl+D`: exit Julia.
- `Ctrl+C`: interrupt execution.

- `Ctrl+L` : clear screen.
- Append `;` to suppress displaying output from a command
- `include("filename.jl")` : source a Julia code file.

Package management

- Julia has an evolving package ecosystem developed by a growing community
- Packages provide functionality which is not part of the core Julia installation
- Each package is a git repository
  - Mostly on github as `Package.jl`, e.g. **AbstractTrees**
  - Packages can be added to and removed from Julia installation
  - Any packages can be installed via a git URL
- Packages can be registered in package registries which are themselves git repositories containing metadata of registered packages.
  - By default, the **General Registry** is used
  - Registered packages are added by name

Importing the package manager

In order to install(add) or remove a package, Julia has a package manager which itself is a package installed by default. We nee to import it in order to use it. The import statement makes all functions from the package available via qualified names, i.e. the function names need to be prefixed with the package name like `Pkg.activate`.

```
using Pkg
```

Environments

list of packages currently used is stored in a *package environment*. A particular directory can be activated by the package manager as the current package environment. The file `Project.toml` in that directory contains the list of packages installed, and the file `Manifest.toml` contains the particular versions ot the installed packages and those installed additionally as dependencies.

Here, we activate a temporary directory as package environment:

```
Pkg.activate(mktempdir())
```

- If we skip this step, a global environment stored in `.julia/environments/vX.Y` under the user home directory will be used. All packages from that global environment will be visible in the activated local evironments.
- Environments allow to separate lists of packages with possibly different versions used in different projects
- The pluto notebooks provided during the course always will use this type of temporary environment as created above.

Adding and using packages

Now, we can add packages, possibly downloading them if necessary:

```
Pkg.add("PlutoUI"); using PlutoUI
```

The `using` statements makes all exported functions from a package available without the need to prefix their names with the name of the package:

Add another package: `AbstractTrees`

```
  Resolving package versions...
Updating `/tmp/jl_dxAEn7/Project.toml`
  [1520ce14] + AbstractTrees v0.3.3
```

```
  Updating `/tmp/jl_dxAEn7/Manifest.toml`
    [1520ce14] + AbstractTrees v0.3.3
```

```
with_terminal() do
    Pkg.add("AbstractTrees")
end
```

List installed packages:

```
Status `/tmp/jl_dxAEn7/Project.toml`
  [1520ce14] AbstractTrees v0.3.3
  [7f904dfe] PlutoUI v0.6.9
```

```
with_terminal() do
    Pkg.status()
end
```

Remove package:

```
Status `/tmp/jl_dxAEn7/Project.toml`
  [7f904dfe] PlutoUI v0.6.9


Updating `/tmp/jl_dxAEn7/Project.toml`
  [1520ce14] - AbstractTrees v0.3.3
Updating `/tmp/jl_dxAEn7/Manifest.toml`
  [1520ce14] - AbstractTrees v0.3.3
```

```
with_terminal() do
    Pkg.rm("AbstractTrees")
    Pkg.status()
end
```

Updating packages

Packages can be updated to their newest version:

```
[?25l[2K[?25hStatus `/tmp/jl_dxAEn7/Project.toml`
  [7f904dfe] PlutoUI v0.6.9


   Updating registry at `~/.julia/registries/General`
   Updating registry at `~/.julia/registries/PackageNursery`
   Updating git-repo `git@github.com:j-fu/PackageNursery`
No Changes to `/tmp/jl_dxAEn7/Project.toml`
No Changes to `/tmp/jl_dxAEn7/Manifest.toml`
```

```
with_terminal() do
    Pkg.update()
    Pkg.status()
end
```

Pinning (fixing) package versions

Sometimes, it is a good idea to fix the version of a package installed. This can be achieved by specifying its version during `Pkg.add()`

```
Pkg.add(name="AbstractTrees",version="0.3.2")
```

Local copies of packages

- Upon installation, local copies of the package source code is downloaded from git repository
- By default located in `.julia/packages` subdirectory of your home folder

Standard number types

- Julia is a strongly typed language, so any variable has a type.
- Standard number types allow fast execution because they are supported in the instruction set of the processors
- Default types are autodected from expression

- The `typeof` function allows to detect the type of a variable

Integers

```
i = 1
```
- `i=1`

```
Int64
```
- `typeof(i)`

Floating point

```
x = 10.0
```
- `x=10.0`

```
Float64
```
- `typeof(x)`

Rational

```
r = 3//7
```
- `r=3//7`

```
Rational{Int64}
```
- `typeof(r)`

Irrational

```
p = π = 3.1415926535897...
```
- `p=π`

```
Irrational{:π}
```
- `typeof(p)`

Complex

```
z = 17.5 + 3.0im
```
- `z=17.5+3im`

```
Complex{Float64}
```
- `typeof(z)`

Vectors

- Elements of a given type stored contiguously in memory
- Vectors and 1-dimensional arrays are the same
- Vectors can be created for any element type
- Element type can be determined by `eltype` method

- Construction by explicit list of elements:

```
v1 =  Int64[1, 2, 3, 4]
```
- `v1=[1,2,3,4,]`

```
Int64
```
- `eltype(v1)`

We can create a vector of floats:

```
v1f =  Float64[1.0,  2.0,  3.0,  4.0]
```
```
v1f=Float64[1,2,3,4]
```

- If one element in the initializer is float, the vector becomes float:

```
v2 =  Float64[1.0,  2.0,  3.0,  4.0]
```
```
v2=[1.0,2,3,4,]
```

- Create vector of zeros for given type:

```
v3 =  Float32[0.0,  0.0,  0.0,  0.0,  0.0]
```
```
v3=zeros(Float32,5)
```

- Fill vector with constant data:

```
Float32[17.0,  17.0,  17.0,  17.0,  17.0]
```
```
fill!(v3,17)
```

Ranges

- Ranges describe sequences of numbers and can be used in loops, array constructors etc.
- They contain the recipe for the sequences, not the full data.

```
r1 = 1:10
```
```
r1=1:10
```

```
UnitRange{Int64}
```
```
typeof(r1)
```

- We can collect the sequence from a range into a vector:

```
w1 =  Int64[1,  2,  3,  4,  5,  6,  7,  8,  9,  10]
```
```
w1=collect(r1)
```

```
Array{Int64,1}
```
```
typeof(w1)
```

- We can add a step size to a range:

```
r2 = 1.0:0.8:9.8
```
```
r2=1:0.8:10
```

```
StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}
```

- Create a vector from a list comprehension containing a range:

```
v4 =  Float64[0.841471,  0.909297,  0.14112,  -0.756802,  -0.958924]
```
```
v4=[sin(i) for i=1:5]
```

Create a random vector of given size:

```
v5 =
  Float64[0.242416, 0.0308306, 0.484686, 0.274097, 0.973073, 0.159527, 0.882612, 0.709
```

```
v5=rand(10)
```

Vector dimensions

```
v6 =  Int64[1, 3, 5, 7, 9]
```

```
v6=collect(1:2:10)
```

- `size` is a tuple of dimensions

```
(5)
```

```
size(v6)
```

- `length` describes the overall length:

```
10
```

```
length(v5)
```

Subarrays

- Copies of parts of arrays:

```
v7 =  Int64[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
v7=collect(1:10)
```

```
subv7 =  Int64[2, 3, 4]
```

```
subv7=v7[2:4]
```

```
Int64[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
subv7[1]=17;v7
```

- Views:

```
v8 =  Int64[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
v8=collect(1:10)
```

```
subv8 =  view(::Array{Int64,1}, 2:4): [2, 3, 4]
```

```
subv8=view(v8,2:4)
```

```
Int64[1, 19, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
subv8[1]=19;v8
```

- The `@views` macro can turn a copy statement into a view

```
v9 =  Int64[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
v9=collect(1:10)
```

```
view(::Array{Int64,1}, 2:4): [2, 3, 4]
```

```
@views subv9=v9[2:4]
```

```
Int64[1, 29, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
subv9[1]=29; v9
```

Dot operations

- element-wise operations on arrays

```
v10 =
Float64[0.0, 0.314159, 0.628319, 0.942478, 1.25664, 1.5708, 1.88496, 2.19911, 2.513
```

```
v10=collect(0:0.1π:2π)
```

```
Float64[0.0, 0.309017, 0.587785, 0.809017, 0.951057, 1.0, 0.951057, 0.809017, 0.587
```

```
sin.(v10)
```

```
Float64[100.0, 100.314, 100.628, 100.942, 101.257, 101.571, 101.885, 102.199, 102.5
```

```
v10.+100
```

Matrices

- Elements of a given type stored contiguously in memory, with two-dimensional access
- Matrices and 2-dimensional arrays are the same

- Zero initialization:

```
m1 = 5×6 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0
```

```
m1=zeros(5,6)
```

- `undef` initialization:

```
m2 = 3×3 Array{Float64,2}:
 6.36599e-314  8.48798e-314  5.0e-324
 6.36599e-314  1.061e-313    0.0
 1.9098e-313   2.122e-314    0.0
```

```
m2=Matrix{Float64}(undef,3,3)
```

- list comprehension:

```
m3 = 6×5 Array{Float64,2}:
  0.367879    0.606531    1.0         1.64872    2.71828
 -0.153092   -0.252406   -0.416147   -0.68611   -1.1312
 -0.240462   -0.396455   -0.653644   -1.07768   -1.77679
  0.353227    0.582373    0.96017     1.58305    2.61001
 -0.0535265  -0.0882502  -0.1455     -0.239889  -0.39551
 -0.308677   -0.508923   -0.839072   -1.3834    -2.28083
```

```
m3=[cos(x)*exp(y) for x=0:2:10, y=-1:0.5:1]
```

- The size of a matrix is the tuple of the two matrix dimensions:

```
(6, 5)
```

```
size(m3)
```

The length of a matrix is the length of the contiguous storage array in memory:

```
30
```

```
length(m3)
```

Linear Algebra

```
using LinearAlgebra
```

Let us create some linear algebra objects:

```
n = 10
```

```
n=10
```

```
w =
Float64[0.79359, 0.666671, 0.109393, 0.366506, 0.421056, 0.790425, 0.792178, 0.63792
```

```
w=rand(n)
```

```
u =
Float64[0.699974, 0.778464, 0.798245, 0.79579, 0.361822, 0.315499, 0.105097, 0.08688
```

```
u=rand(n)
```

```
A = 10×10 Array{Float64,2}:
 0.824022   0.399623  0.52761     0.800789   …  0.339729   0.251542  0.787242
 0.950459   0.614287  0.00529413  0.704582      0.520941   0.799017  0.901162
 0.0783334  0.302962  0.0757998   0.341549      0.305228   0.847785  0.442627
 0.185168   0.356941  0.386134    0.359076      0.980641   0.308131  0.385319
 0.865389   0.389219  0.0832434   0.0502272     0.0805102  0.306361  0.183116
 0.117851   0.366842  0.27298     0.0425427  …  0.333443   0.665354  0.447239
 0.416079   0.301092  0.357823    0.864643      0.295109   0.442502  0.343804
 0.398784   0.92582   0.507028    0.844632      0.160503   0.579967  0.367731
 0.193747   0.302114  0.345443    0.393442      0.890489   0.409108  0.0393411
 0.382147   0.272667  0.504316    0.880203      0.196281   0.246422  0.821393
```

```
A=rand(n,n)
```

- Mean square norm $||u||_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$

```
1.684103735769847
```

```
norm(u)
```

- Dot product: $(u, w) = \sum_{i=1}^{n} u_i w_i$

```
2.0555931450006275
```

```
dot(u,w)
```

- Matrix vector product

```
Float64[2.65575, 2.62523, 1.22743, 1.40503, 1.5594, 1.12417, 2.0009, 2.5607, 1.3284
```

```
A*u
```

- Trace (sum of main diagonal elements), determinant, inverse

```
(4.27288,  0.0345759,  10×10 Array{Float64,2}:
  0.932777    0.291554  -0.786702  -0.723059  …  -0.489226    0.35
 -0.521381    0.185718   0.414381   0.861938      1.69629    -0.29
  1.57368    -1.91637    0.350873   0.340511     -0.163444   -0.50
  0.209525   0.387221   0.308942  -0.732407      0.261329    0.73
  0.375511  -0.298364   1.13381   -0.580115      0.173965    0.99
 -1.27153   -0.167275  -0.157481   0.843454  …  -0.142325   -0.71
 -0.473635   0.448125  -0.869938  -0.668267     -0.0543635   0.64
 -0.0418096  0.173461   0.283212   0.808199     -0.216216    0.53
  0.514353  -0.161763   0.462225  -0.787276     -0.465144    0.02
 -0.878337   0.57295   -0.848495   1.42106      -0.380797   -1.91
```

```
tr(A),det(A), inv(A)
```

Control structures

- Conditional execution

```
cond1 = false
```

```
cond1=false
```

```
cond2 = true
```

```
cond2=true
```

```
"cond2"
```

```
if cond1
    "cond1"
elseif cond2
    "cond2"
else
    "nothing"
end
```

- '?' operator for writing shorter code (borrowed from C)

```
"nothing"
```

```
cond1 ? "cond1" : "nothing"
```

- for loop:

```
1
2
3
4
5
```

```
with_terminal() do
    for i in 1:5
        println(i)
    end
end
```

- Preliminary exit of loop

```
1
2
3
4
5
6
```

```
with_terminal() do
    for i in 1:10
        println(i)
        if i>5
            break
        end
```

```
    end
end
```

- Skipping iterations

```
1
2
3
4
6
7
8
9
10
```

```
with_terminal() do
    for i in 1:10
      if i==5
        continue
      end
      println(i)
    end
end
```

Functions

- All arguments to functions are passed by reference
- Function name ending with ! indicates that the function mutates at least one argument, typically the first. This is a convention, not a syntax rule.
- Function objects can be assigned to variables

Structure of function definition

```
function func(req1, req2; key1=dflt1, key2=dflt2)
   # do stuff
    return out1, out2, out3
end
```

- Required arguments are separated with a comma and use the positional notation
- Optional arguments need a default value in the signature
- Return statement is optional, by default, the result of the last statement is returned
- Multiple outputs can be returned as a tuple, e.g., return out1, out2, out3.

```
func0 (generic function with 1 method)
```

```
function func0(x; y=0)
    x+2*y
end
```

```
1
```

```
func0(1)
```

```
201
```

- One line function definition

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

```
g(x)=exp(sin(x))
```

```
1.151562836514535
```

```
g(3)
```

- Nested function definitions

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

```
function outerfunction(n)
    function innerfunction(i)
        println(i)
    end
    for i=1:n
        innerfunction(i)
    end
end
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
with_terminal() do
    outerfunction(13)
end
```

- Functions are variables, too

```
1.151562836514535
```

```
h=g; h(3)
```

- Functions as function parameters

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

```
F(f,x)= f(x)
```

```
1.151562836514535
```

```
F(g,3)
```

- Anonymous functions (convenient in function parameters):

```
0.1411200080598672
```

```
F(x -> sin(x),3)
```

- Do-block syntax: the body of first parameter is in the `do ... end` block:

```
1.151562836514535
```

```
F(3) do x
    exp(sin(x))
end
```

Functions and vectors

- Dot syntax can be used to make any function work on vectors

```
v11 =  Float64[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

```
v11=collect(0:0.1:1)
```

```
Float64[1.0, 1.10499, 1.21978, 1.34383, 1.47612, 1.61515, 1.75882, 1.9045, 2.04901,
```

```
h.(v11)
```

- map function on vector

```
Float64[1.0, 1.10499, 1.21978, 1.34383, 1.47612, 1.61515, 1.75882, 1.9045, 2.04901,
```

```
map(h,v11)
```

- mapreduce: apply operator to each element and collect data

```
0.0
```

```
mapreduce(x->x,*,v11)
```

```
5.5
```

```
mapreduce(x->x,+,v11)
```

```
5.5
```

```
sum(v11)
```

Macros

Julia allows to define macros which allow to modify Julia statements before they are compiled and executed. This capability is similar to the preprocessor in C or C++. Macro names start with `@`. Occasionally we will use predefined macros, e.g. `@elapsed` for returning the time used by some statement.

```
0.000258831
```

```
@elapsed inv(rand(100,100))
```