Getting Started with Julia

Jeff Mills, 8-2019

Workshop material available at:

https://github.com/tszanalytics/Cincinnati_Julia_Workshop_2019

TL; DR Quick Start

Install: https://julialang.org/downloads/platform.html

https://julialang.org/ - scroll to the bottom for installation instructions for editors/IDEs

Learn: https://julialang.org/learning/

Nice intro. video: J. Harriman JuliaCon 2018 Intro workshop video

Concise Intro to Julia as a pdf: https://github.com/bkamins/The-Julia-Express

Go through this and try it out: https://juliadocs.github.io/Julia-Cheat-Sheet/

Starting plots - Get the following to work:
plot(randn(1000), randn(1000), st=:scatter, markercolor=:blue, alpha=0.4, label="")
plot(randn(1000), randn(1000), st=:scatter, markercolor=:1:100, alpha=0.7, label="")

Speeding up workflow (the compilation)

Create a startup.jl file with: `using <list of packages>` to compile on startup

Location of startup.jl file:

C:\Users\~\AppData\Local\Julia-1.1.0\etc\julia

PackageCompiler.jl

```
]dev PackageCompiler
using PackageCompiler
#compile_incremental(:Plots, force=false)
compile_incremental(:Plots, :StatsPlots, :Distributions, :Random, force =
false)
LOOK FOR ERRORS and add package that cause the error (repeat as necessary!)
# Once done.[** find the file sys.dll and replace it with the new one]
# copy from
# C:\Users\UserName\.julia\dev\PackageCompiler\sysimg\sys.dll
# to C:\Users\UserName\AppData\Local\Julia-1.1.0\lib\julia\sys.dll
*] add VisualRegressionTests UnicodePlots LatexStrings Images RDatasets
```

*** Must then run Julia and Atom with Admin privilages (in shortcut properties) ***

Why Julia?

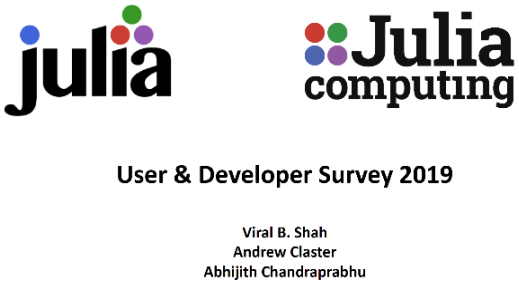
Nature article: https://www.nature.com/articles/d41586-019-02310-3

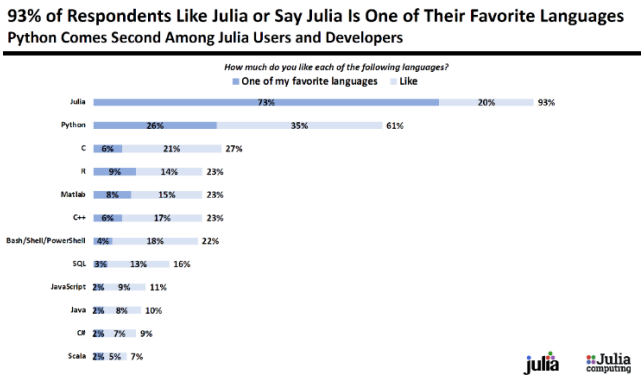
10 Reasons Why You Should Learn Julia article:

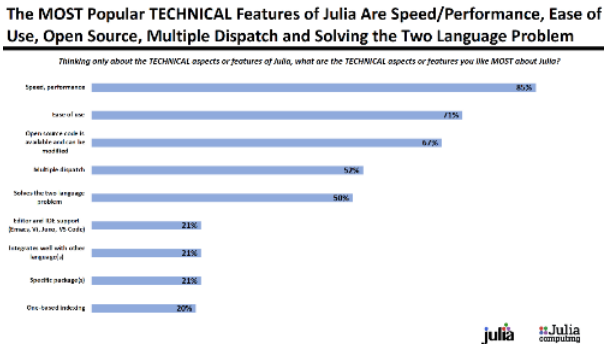
https://blog.goodaudience.com/10-reasons-why-you-should-learn-julia-d786ac29c6ca

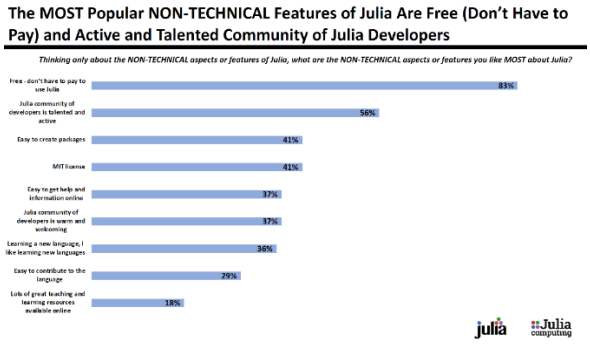
The developers wanted to create a language with all of their favorite things, and they did it!

We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled. — julialang.org

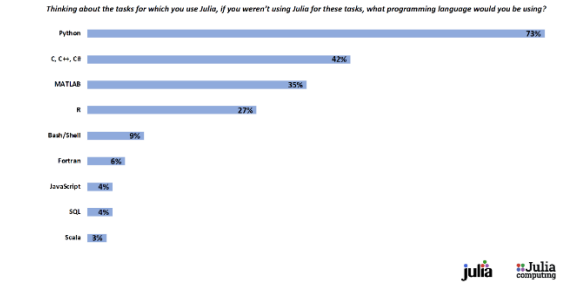




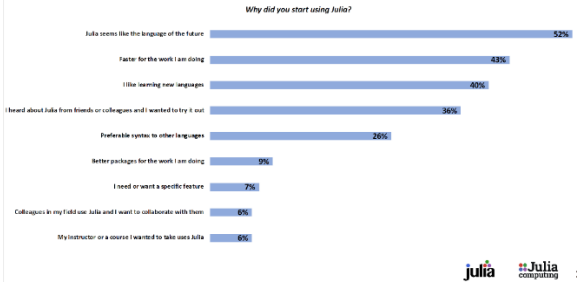




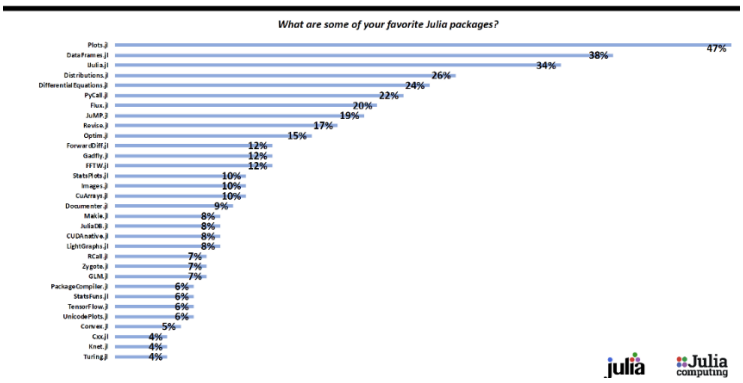
If Not for Julia, Most Would Be Using Python, Followed by C/C++/C#, MATLAB and R



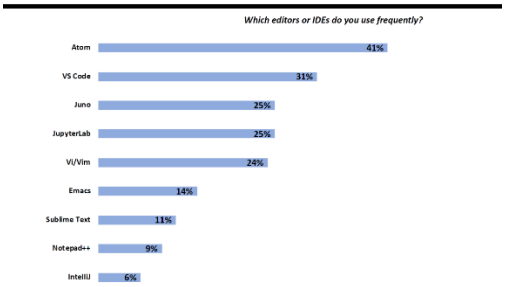
Respondents Started Using Julia Because of Speed and Because Julia Seems Like the Language of the Future



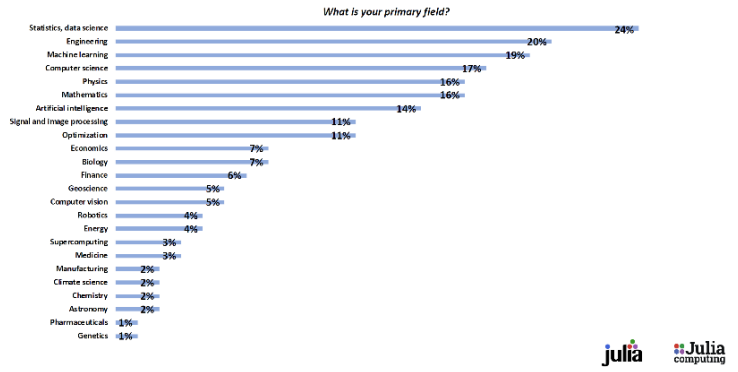
The Most Popular Julia Packages Are Plots.jl, DataFrames.jl and IJulia.jl



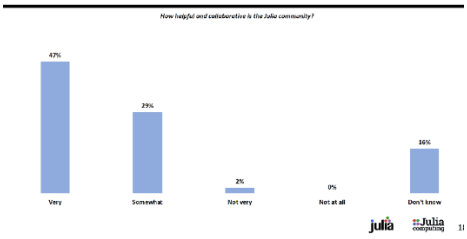
Atom and VS Code Are the Most Popular Editors or IDEs



The Most Popular Fields Are Statistics, Data Science, Engineering, Machine Learning, Computer Science, Physics, Mathematics, Artificial Intelligence, Optimization and Signal and Image Processing



Most Say the Julia Community Is 'Very' or 'Somewhat' Helpful and Collaborative



Installing Julia

Installing Julia and UI/Editors

1. Binaries from Julia computing. Version 1.1 is current most stable version
2. Install packages. IJulia for notebooks, then notebook()
3. Atom/Juno Add package uber-juno in atom, UNCHECK automatic update!

Detailed instructions for installing Julia on Windows:
The most convenient way to install Julia on Windows is by using the binary version installer available from the JuliaLang website.

The following steps are required to install Julia on a Windows system:

1. Download the Windows Self-Extracting Archive (.exe) from https://julialang.org/downloads/. It is recommended to select the 64-bit version.
2. Run the downloaded `*.exe` file to unpack Julia. We recommend extracting Julia into a directory path that does not contain any spaces, for example, `C:\Julia-1.0.1`.
3. After a successful installation, a Julia shortcut will be added to your start menu—at this point, you can select the shortcut to see whether Julia loads correctly.
4. Add `julia.exe` to your system path, as follows:
    1. Open Windows Explorer, right-click on the This PC computer icon, and select Properties.
    2. Click Advanced system settings and go to Environment Variables....
    3. Select the `Path` variable and click Edit....
    4. To the variable value, add `C:\Julia-1.0.1\bin` (in this instruction, we assume that Julia has been installed to `C:\Julia-1.0.1`). Please note that, depending on your Windows version, there is either one `Path` value per line or a semicolon `;` is used to separate values in the `Path` list.
    5. Click OK to confirm. Now, Julia can be run anywhere from the console.

When adding julia.exe to your system path, please note that there are two variable groups on this screen, User variables and System variables. We recommend using User variables. Please note that adding `julia.exe` to the system Path makes it possible for other tools, such as Juno, to automatically locate Julia (Juno also allows for manual Julia path configuration—this can be found in the option marked Packages | Julia client | Settings).

Install some of the packages that we will look at (you will need an internet connection)

```
To install more than one package at a time, first install
the Pkg package, then use:

using Pkg

#### Note: no commas between package names in the following:
packages = split(
    """Plots GR PlotlyJS Interact
    BenchmarkTools
    DataFrames
    Distributions StatPlots
    Optim JuMP
    Random LinearAlgebra
    ForwardDiff
    """)

for package in packages
    Pkg.add(package)
end
```

Using Julia

1. REPL + Notepad, Notepad++, etc.
2. Jupyter notebook
3. Atom/Juno
4. VSCode
5. JuliaBox

Jupyter notebooks

To install Julia and Jupyter locally on your own machine, do the following:
[Note that it is *not* necessary to install Anaconda separately; Julia will do this automatically for you.]
Download and install the stable version of Julia from here for your operating system.
Run the copy of Julia that you just installed. In the Julia REPL, run the following commands:
[* **If** you use Linux, first type: julia> ENV["JUPYTER"] = ""]

Now install the IJulia package, which will automatically install Jupyter (using `miniconda`):

Either

```
]
Add IJulia
```

or

```
using Pkg
Pkg.add("IJulia")
```

[two ways to do the same thing]

Integrated Desktop Environments (IDEs)

There are two IDEs (Integrated Development Environments) available for Julia: Juno, based on the Atom editor, and a Julia plug-in for the Visual Studio Code editor.

Please download Atom and install the uber-juno package; this will give you a Julia development environment. More info is available at the Juno IDE homepage.

Detailed Juno installation instructions

Juno is the recommended IDE for Julia development. The Juno IDE is available at http://junolab.org/. However, Juno runs as a plugin to Atom (https://atom.io/). Hence, in order to install Juno, you need to take the following steps:

1. Make sure that you have installed Julia and added it to the command path (following the instructions given in previous sections).
2. Download and install Atom, available at https://atom.io/.
3. Once the installation is complete, Atom will start automatically.
4. Press *Ctrl* + *,* (*Ctrl* key + *comma* key) to open the Atom settings screen.

5. Select the Install tab.
6. In the Search packages field, type uber-juno and press *Enter*.
7. You will see the uber-juno package developed by JunoLab—click Install to install the package.
8. In order to test your installation, click the Show console tab on the left.

Please note that when being run for the first time from Atom, Julia takes longer to start. This happens because Juno is installed along with several other packages that are being compiled before their first use.

MS VSCode is another IDE that is popular: https://github.com/julia-vscode/julia-vscode - for installation instructions.

JuliaBox is a ready-made pre-installed Julia environment accessible from the web browser. It is available at https://juliabox.com/

Using Julia: Two "cheat sheets" with summaries of basic Julia syntax:

- a one-page summary by Steven Johnson

- a more extensive summary - Let's go through this one [https://juliadocs.github.io/Julia-Cheat-Sheet/].

There is an ever-growing list of resources for learning Julia available on the learning page of the Julia homepage; in particular (especially economics and finance) check out the QuantEcon lectures.

Using Julia: Things to do in Julia:

Write unicode symbols in the code.

Tab completion

Plots

- Plotting in a loop, plotting in a function
- Histogram, density, barplot, animation sliders?
- **Plotting in a function**
- Plot.display or name the plot (plt1 = plot(…)) then return it

- **see mcmc_sample_plot_2019.jl** in C:\Users\millsjf\OneDrive\BayesTesting
- ```
  using Plots, StatsPlots, StatsBase
  # mcs = an MCMC sample (a vector)
  function mcmc_sample_plot(mcs)
      r = autocor(mcs,1:20)
      plot(  plot(mcs,label=""),
      plot(mcs, st=:density, gridcolor=:lightgrey,label="density"),
      plot(r,linewidth=2, st=:bar, bar_width = 0.5,
         label="",xlabel="ACF"),
      layout = (3,1), size=(600,600))
      hline!([mean(mcs) 0], color = [:red :black], label=["mean"
  ""])
      vline!([0 mean(mcs)], color = [:black :red], label=[""
  "mean"])
      # vline!([0 1.0], color = [:black :green],label=["" "true
  mean"])
  end
  ```
- **Plotting in a loop**
- Plot.display or name the plot (as with plotting in a function)

UNICODE characters in strings and formulas

```
s="abπ∑ef\n"
print(s)
ß=2π/3
```

Dice rolls

```
Δ = rand(1:6)

# Complex floating-point numbers
x=2.1+3.2im

# SWAP TWO NUMBERS: Don't need a swap macro
a,b=b,a

# checking approx. equality function:
isapprox(3.0, 3.01, rtol=0.1)

# COMPARISON OPERATORS
a = 2
b = 3
c = 3
@show(a < c && b < c)  # the AND operator
@show(a < c || b < c)  # the OR operator
@show(a != b) ; # NOT equal
```
```

```
# GETTING USER RESPONSE TO TEXT
println("Who are you?")
s=readline()
println("Hello $s and Hello World!")

# Works much better in a function
# as a function:
function whoru()
        println("Who are you?")
        s=readline()
        println("Hello $s and Hello World!")
end
```

Functions

PRO TIP: put EVERYTHING in functions for greatly improved performance

```
function name(args; kwargs)
    body of function
    return objects to return
end
```

E.g.

```
function sum_abs(x, y)
    d = sqrt(x^2) + sqrt(y^2)
    return d
end
```

A simple function can be constructed on one line:

```
sum_abs(x, y) = sqrt(x^2) + sqrt(y^2)
```

Then to use:

```
d = sum_abs(1, 2)
```

Functions of functions:

- Can call functions with function, composable functions:

```
julia> (sqrt ∘ +)(3, 5)   # \circ then tab key to get the operator
(sqrt ∘ +)(3, 5)
```

This adds the numbers then takes the sqrt. E.g.

```
julia> f(x,y) = (sqrt ∘ + )(x, y)
f (generic function with 1 method)

julia> f(2,2)
2.0
```

Alternatively,

```
julia> f1(x) = √x   # \sqrt (then tab key) to get the symbol
g(x,y) = x + y
f1(g(2,2))  # just works!
So does:
g2(x,y,f1,g) = f1(g(x,y))
2(2,2,f1,g)
```

Overloading functions – see functions_examples.ipynb

```
sum_abs(x, y, z) = sqrt(x^2 + sqrt(y^2) + sqrt(z^2)

d2 = sum_abs(1, 2)
d3 = sum_abs(1, 2, 4)
```

Multiple dispatch

```
function sum_abs(x::Int, y::Int)
    d = x^2 + y^2
    return d
end

function sum_abs(x::Float64, y::Float64)
    d = x + y - 10
    return d
end
```

Functions are methods in Julia. Methods are defined outside the type/struct (class). You don't have to go back to the original defn. of the Type (class) to change or define a new method.

**** Multiple dispatch (and function overloading) explained**:

C:\Users\millsjf\OneDrive - University of Cincinnati\Julia_help_notes_textbooks\

multiple_dispatch_explained.jl

Two excellent videos explaining multiple dispatch:

https://www.youtube.com/watch?v=kc9HwsxE1OY

https://www.youtube.com/watch?v=HAEgGFqbVkA

See **julia_helpful_example_snippets.jl**

Loops

If else statements

Comprehension

Distributions

DataFrames

CSV

Explore packages: Plots, StatsPlots, LinearAlgebra, Distributions, StatsFuns, DataFrames, RCall, PyCall, JavaCall, ccall …

Arrays, Tuples, Dictionaries

See – **Julia Express** – pdf in github repository.

Julia 1.0 Programming Cookbook by B. Kaminski and P. Szufel

https://learning.oreilly.com/library/view/julia-10-programming/9781788998369/?ar

The base Julia code is all written in Julia, so you can look up the source code for packages, and take what you want,.

@edit abs(1)

Exercises

- Create/download and read in a CSV file
- Compute the mean, std, 0.95 interval, plot the histogram.
- An experiment with a new treatment results in 6 successes in 30 trials, whereas the placebo results in 2 successes in 28 trials. If all you know before the trial was conducted was that the number of successes is somewhere between 0 and 100%, what is the probability that "chance of success" is greater for the treatment than for placebo? Hint: posterior with a uniform prior is Beta(n+1, n-s+1).

Customize Julia at startup

Create a `hello.jl` file in your working directory, containing the following line:

```
println("Hello " * join(ARGS, ", "))
```

We will later run this file on Julia startup.

There are four methods you can use to parameterize the booting of Julia:

- By setting startup switches
- By passing a startup file
- By defining the `~/.julia/config/startup.jl` file
- By setting environment variables

Put the following statements in the `~/.julia/config/startup.jl` file, using your favorite editor:

- ```
  using Random
  ENV["JULIA_EDITOR"] = "atom"  [or “vim” or “Notepad++” etc.]
  println("Setup successful")
  ```

You can use **the `@edit` macro** to open the location of the definition of the `sin` function in your chosen editor. Julia recognizes the following editors: Vim, Emacs, gedit, textmate, mate, kate, Sublime Text, atom, Notepad++, and Visual Studio Code.

The simplest way to control Julia startup is to pass switches to it. You can get the full list of switches by running the following in the console:

```
$ julia --help
```

In Julia, environment variables can be accessed and changed via the `ENV` dictionary.

Multiprocessing setup

In order to test how multiprocessing works, prepare two simple files that display a text message in the console. When running parallelization tests, we will see messages generated by those scripts appear asynchronously.

Create a `hello.jl` file in your working directory, containing the following code:

```
println("Hello " * join(ARGS, ", "))
```
```

And create `hello2.jl` with the following code:

```
println("Hello " * join(ARGS, ", "))
sleep(1)
```

In order to start several Julia processes, perform the following steps:

1. Specify the number of required worker processes using the `-p` option on Julia startup.
2. Then, check the number of workers in Julia by using the `nworkers()` function from the `Distributed` package.
3. Run the command following `$` in your OS shell, then import the `Distributed` package and write `nworkers()` while in Julia, and then use `exit()` to go back to the shell:

```
$ julia --banner=no -p 2

julia> using Distributed

julia> nworkers()
2

julia> exit()

$
```

If you want to execute some script on every worker on startup, you can do it using the `-L` option.

4. Run the `hello.jl` and `hello2.jl` scripts (the steps to start Julia and exit it are the same as in the preceding steps):

```
$ julia --banner=no -p auto -L hello.jl
Hello !
 From worker 4: Hello !
 From worker 5: Hello !
julia> From worker 2: Hello !
 From worker 3: Hello !
julia> exit()

$ julia --banner=no -p auto -L hello2.jl
Hello !
 From worker 4: Hello !
 From worker 5: Hello !
 From worker 2: Hello !
 From worker 3: Hello !
julia> exit()
```

Multithreading in Julia

Julia can be run in a multithreaded mode. This mode is achieved via the `JULIA_NUM_THREADS` system environment parameter. One should perform the following steps:

1. To start Julia with the number of threads equal to the number of cores in your machine, you have to set the environment variable `JULIA_NUM_THREADS` first
2. Check how many threads Julia is using with the `Threads.nthreads()` function

Running the preceding steps is handled differently on Linux and Windows.

Here is a list of steps to be followed:

1. If you are using bash on Linux, run the following commands:

```
$ export JULIA_NUM_THREADS=`nproc`
$ julia -e "println(Threads.nthreads())"
4
$
```

2. If you are using cmd on Windows, run the following commands:

```
C:\> set JULIA_NUM_THREADS=%NUMBER_OF_PROCESSORS%
C:\> julia -e "println(Threads.nthreads())"
 4
C:\>
```

Observe that we have not used the `-i` option in either case, so the process terminated immediately.

A switch, `-p {N|auto}`, tells Julia to spin up `N` additional worker processes on startup. The `auto` option in the `-p` switch starts as many workers as you have cores on your machine, so `julia -p auto` is equivalent to:

- `julia -p `nproc`` on Linux
- `julia -p %NUMBER_OF_PROCESSORS%` on Windows

It is important to understand that when you start `N` workers, where `N` is greater than `1`, then Julia will spin up `N+1` processes. You can check it using the `nprocs()` function—one master process and `N` worker processes. If `N` is equal to `1`, then only one process is started.

We can see here that `hello.jl` was executed on the master process and on all of the worker processes. Additionally, observe that the execution was asynchronous. In this case, workers `4` and `5` printed their message before the Julia prompt was printed by the master process, but workers `2` and `3` executed their print method after it. By adding a `sleep(1)` statement in `hello2.jl`, we make the master process wait for one second, which is sufficient time for all workers to run their `println` command.

As you have seen, in order to start Julia with multiple threads, you have to set the environment variable `JULIA_NUM_THREADS`. It is used by Julia to determine how many threads it should use. This value—in order to have any effect—must be set before Julia is started. This means that you can access it via the `ENV["JULIA_NUM_THREADS"]` option but changing it when Julia is running

will not add or remove threads. Therefore, before running Julia you have to type the following in a terminal session:

- `export JULIA_NUM_THREADS=[number of threads]` on Linux or if you use bash on Windows
- `set JULIA_NUM_THREADS=[number of threads]` on Windows if you use the standard shell
- You can also add processes after Julia has started using the `addprocs` function. We are running the following code on Windows with two drives, `C:` and `D:`, present. Julia is started in the `D:\` directory:
- ```
  D:\> julia --banner=no -p 2 -L hello2.jl
  Hello
   From worker 3: Hello
   From worker 2: Hello
  julia> pwd()
  "D:\\"

  julia> using Distributed

  julia> pmap(i -> (i, myid(), pwd()), 1:nworkers())
  2-element Array{Tuple{Int64,Int64,String},1}:
   (1, 2, "D:\\")
   (2, 3, "D:\\")

  julia> cd("C:\\")

  julia> pwd()
  "C:\\"

  julia> addprocs(2)
  2-element Array{Int64,1}:
   4
   5

  julia> pmap(i -> (i,myid(),pwd()), 1:nworkers())
  4-element Array{Tuple{Int64,Int64,String},1}:
   (1, 3, "D:\\")
   (2, 2, "D:\\")
   (3, 5, "C:\\")
   (4, 4, "C:\\")
  ```
- In particular, we see that each worker has its own working directory, which is initially set to the working directory of the master Julia process when it is started. Also, `addprocs` does not execute the script that was specified by the `-L` switch on Julia startup.
- Additionally, we can see the simple use of the `pmap` and `myid` functions. The first one is a parallelized version of the `map` function. The second returns the identification number of a process that it is run on.
- As we explained earlier, it is not possible to add threads to a running Julia process. The number of threads has to be specified before Julia is started.
- Deciding between using multiple processes and multiple threads is not a simple decision. A rule of thumb is to use threads if there is a need for data sharing and frequent communication between tasks running in parallel.
  ```