Introduction to Julia Programming Language

Jian Tao

jtao@tamu.edu

HPRC Short Course

11/19/2021

TEXAS A&M UNIVERSITY Visualization

High Performance Research Computing DIVISION OF RESEARCH

TEXAS A&M Institute of Data Science

Part I.
Julia - What and Why?

Julia is a high-level general-purpose dynamic programming language primarily designed for **high-performance numerical analysis and computational science**.

- Born in MIT's Computer Science and Artificial Intelligence Lab in 2009
- Combined the best features of Ruby, MatLab, C, Python, R, and others
- First release in 2012
- Latest stable release v1.6.3 as of Nov 16, 2021
- https://julialang.org/
- customized for "greedy, unreasonable, demanding programmers".
- Julia Computing established in 2015 to provide commercial support.

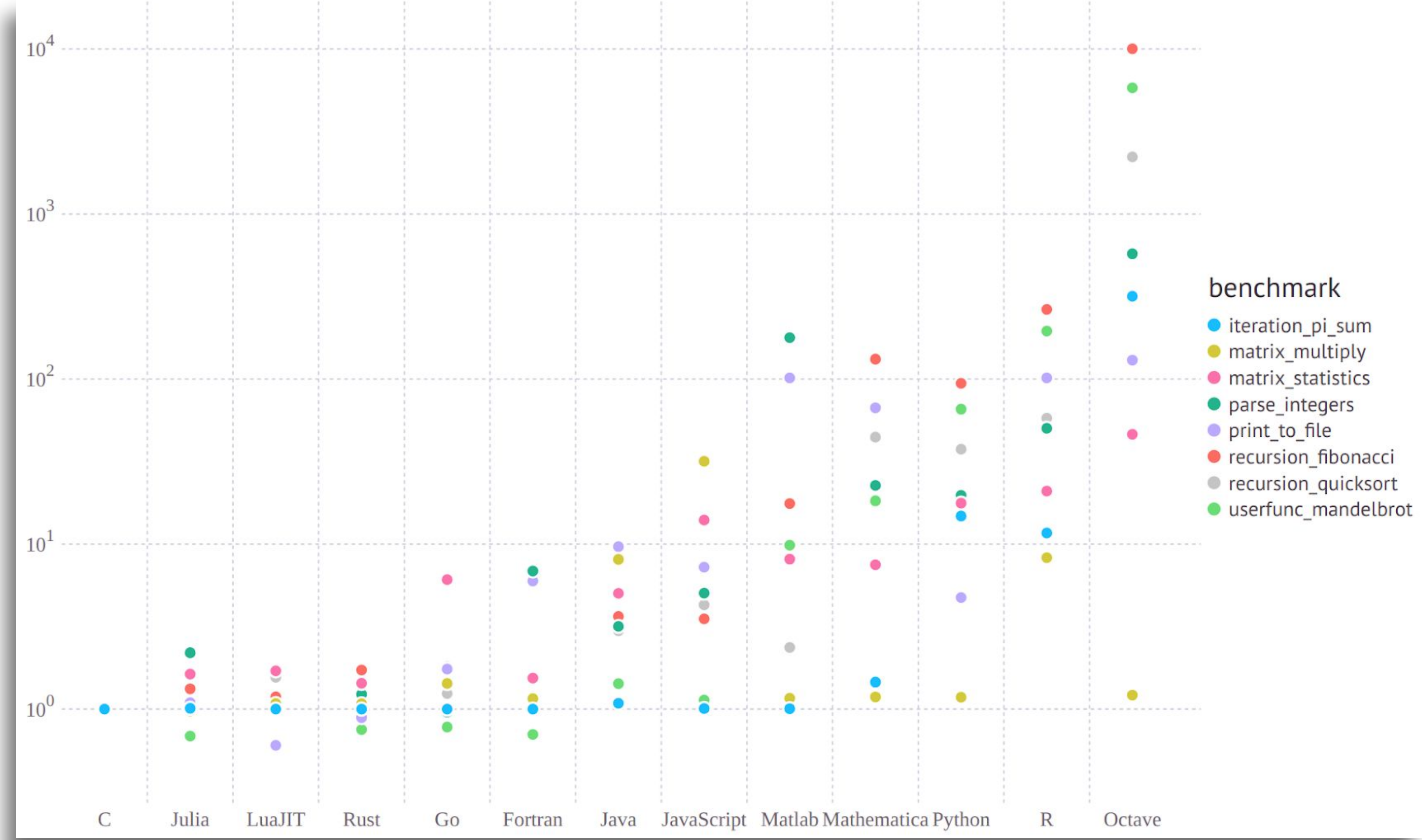


Image Credit: [Julialang.org]

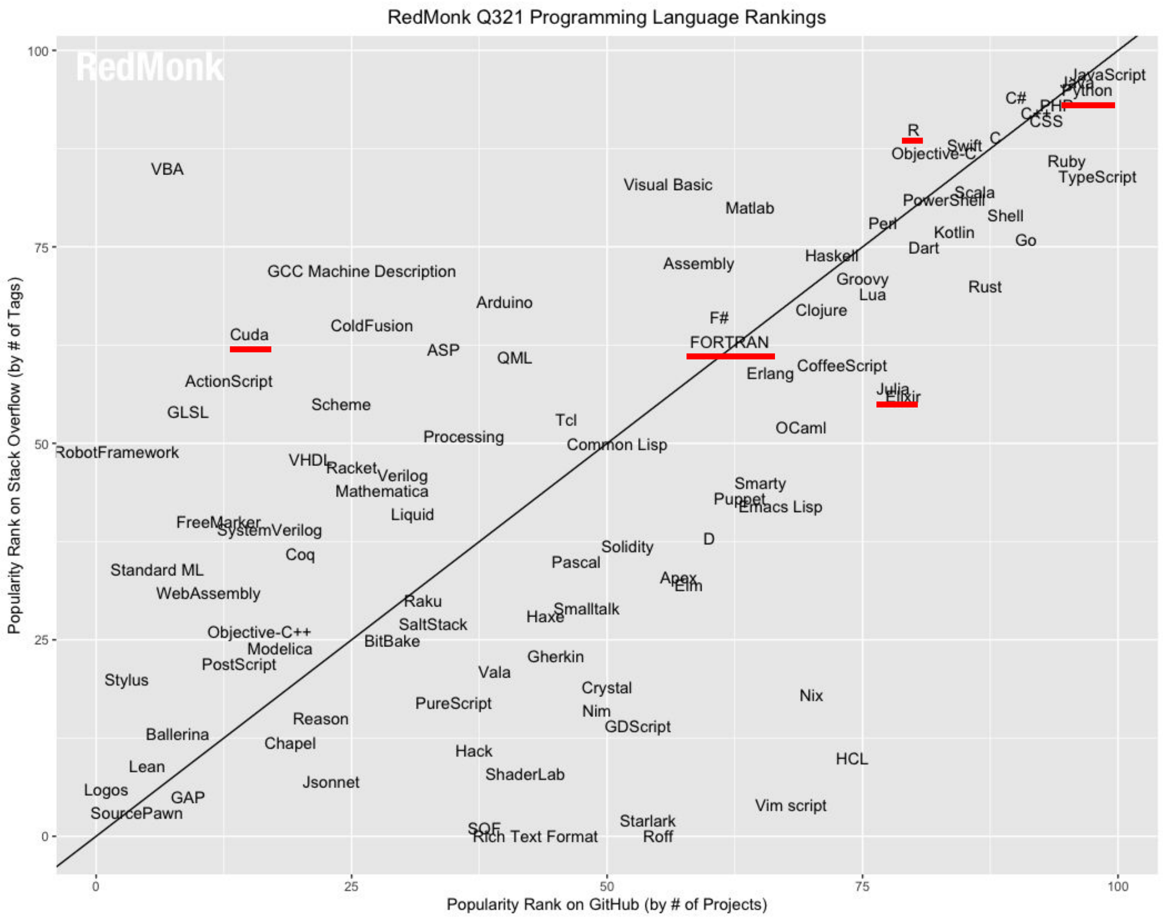


Image Credit: RedMonk (https://redmonk.com/sogrady/2021/08/05/language-rankings-6-21/)

Major features of **Julia:**

- **Fast**: designed for high performance,
- **General**: supporting different programming patterns,
- **Dynamic**: dynamically-typed with good support for interactive use,
- **Technical**: efficient numerical computing with a math-friendly syntax,
- **Optionally typed**: a rich language of descriptive data types,
- **Composable**: Julia’s packages naturally work well together.

*Mostly importantly, for many of us, **Julia** seems to be the language of choice for **Scientific Machine Learning**.*

*"Julia is as programmable as Python while it is as fast as Fortran for number crunching. It is like **Python on steroids**."*
--an anonymous Julia user on the first impression of Julia.

Juno IDE

- Juno is an Integrated Development Environment (IDE) for the Julia language.
- Juno is built on Atom, a text editor provided by Github.

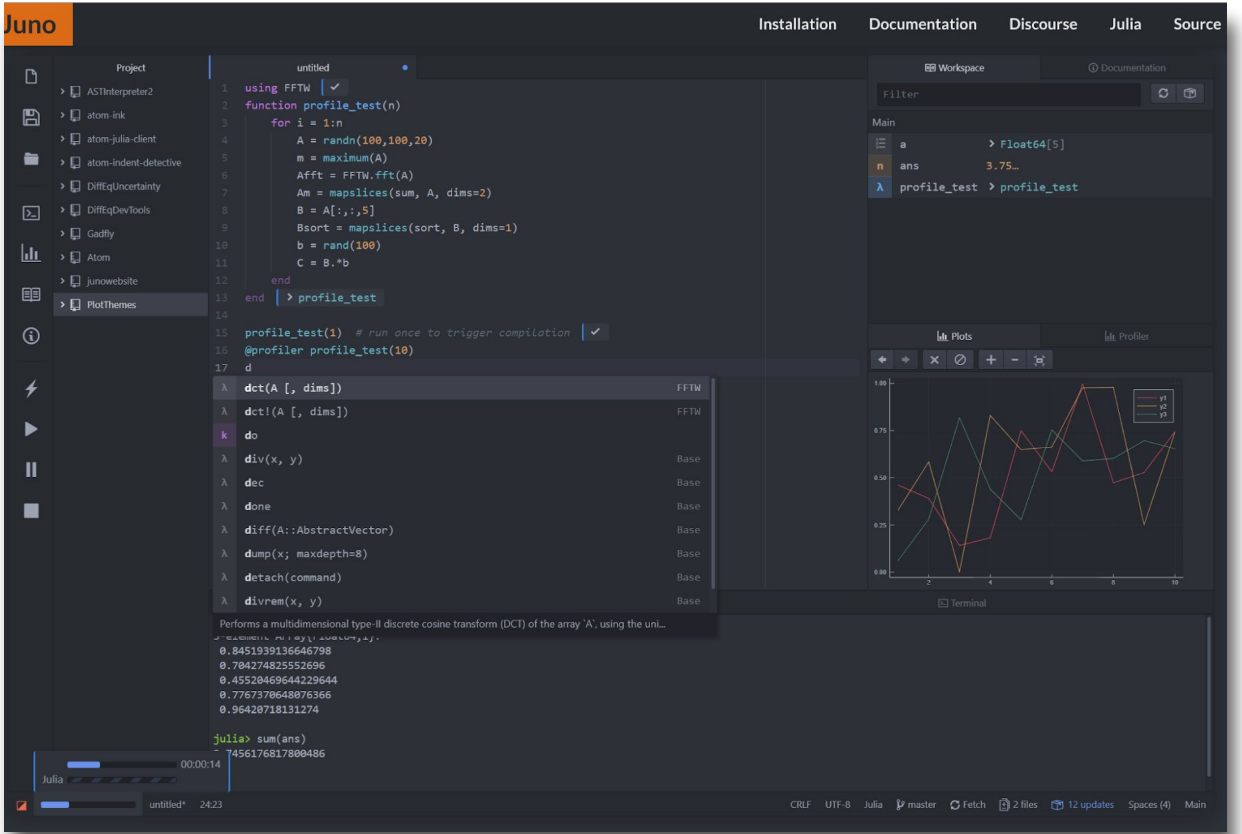


Image Credit: Juno (http://junolab.org/)

Jupyter Notebook

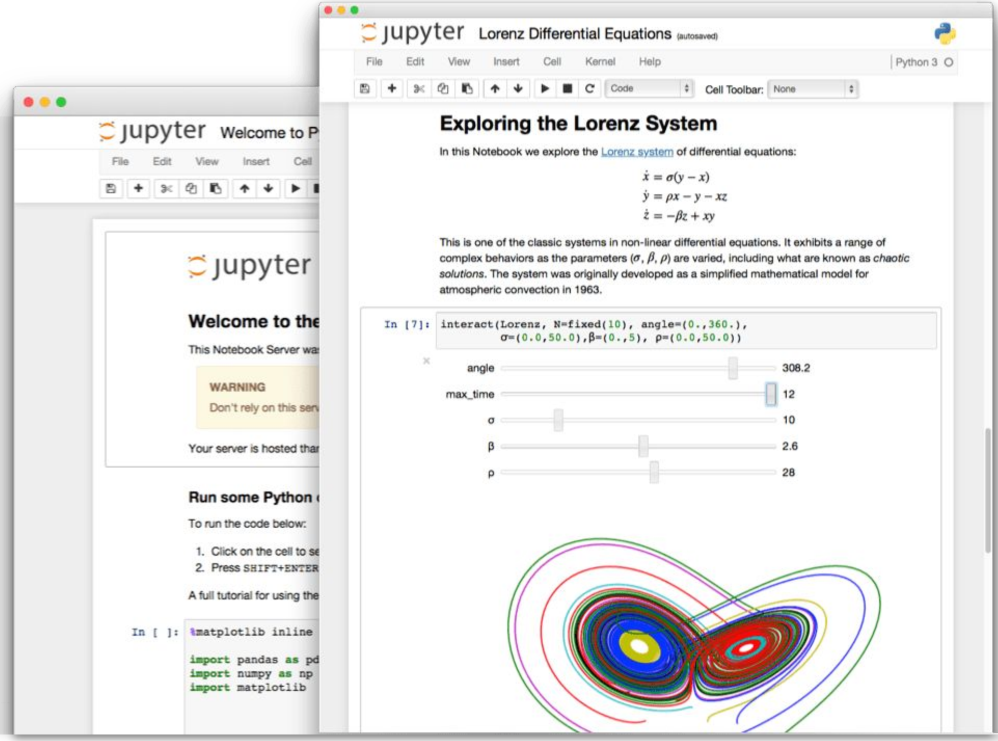
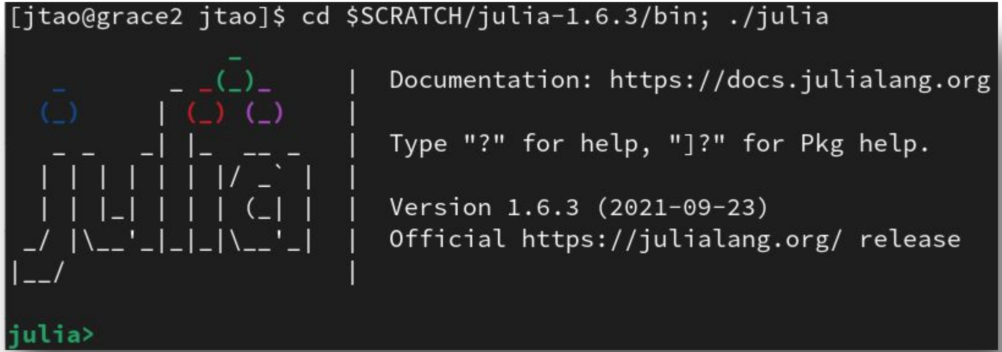


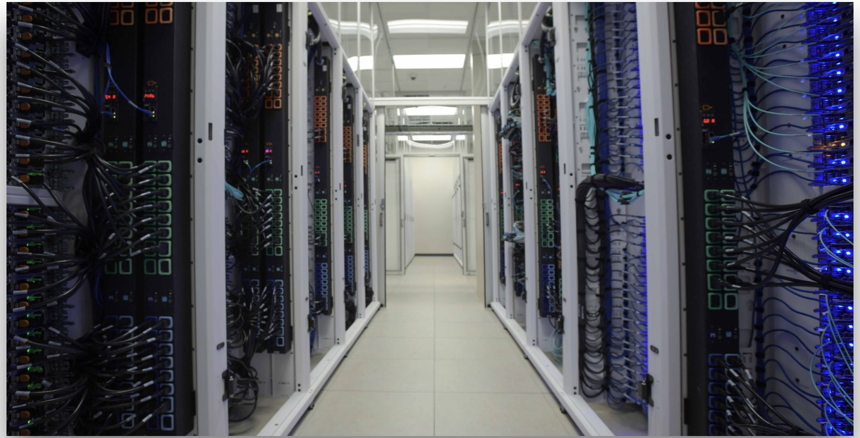
Image Credit: Jupyter (http://jupyter.org/)

Julia REPL



- Julia comes with a full-featured interactive command-line **REPL** (read-eval-print loop) built into the Julia executable.
- In addition to allowing quick and easy evaluation of Julia statements, it has a searchable history, tab-completion, many helpful keybindings, and dedicated help and shell modes.

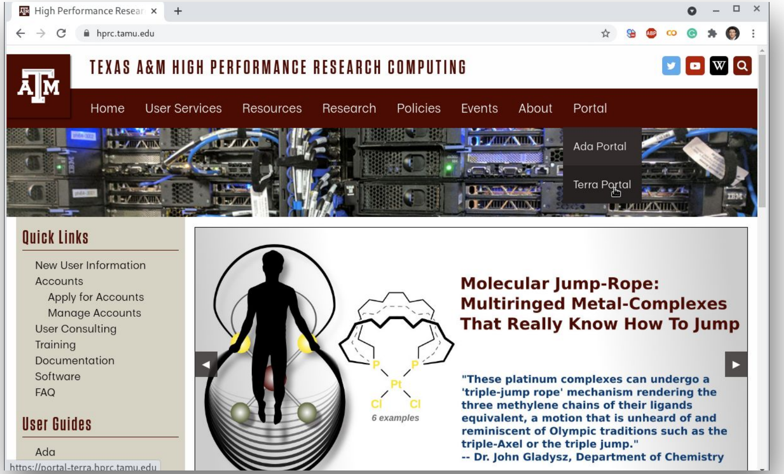
Part II.
Shell Access to Grace @ HPRC



Grace Login Nodes

	NVIDIA A100 GPU	NVIDIA RTX 6000 GPU	NVIDIA T4 GPU	No GPU
Hostnames	grace1.hprc.tamu.edu	grace2.hprc.tamu.edu	grace3.hprc.tamu.edu	grace4.hprc.tamu.edu grace5.hprc.tamu.edu
Processor Type	Intel Xeon 6248R 3.0GHz 24-core			
Memory	384GB DDR4 3200 MHz			
Total Nodes	1	1	1	2
Cores/Node	48			
Interconnect	Mellanox HDR 100 InfiniBand			
Local Disk Space	per node: two 480 GB SSD drives, 1.6 TB NVMe			

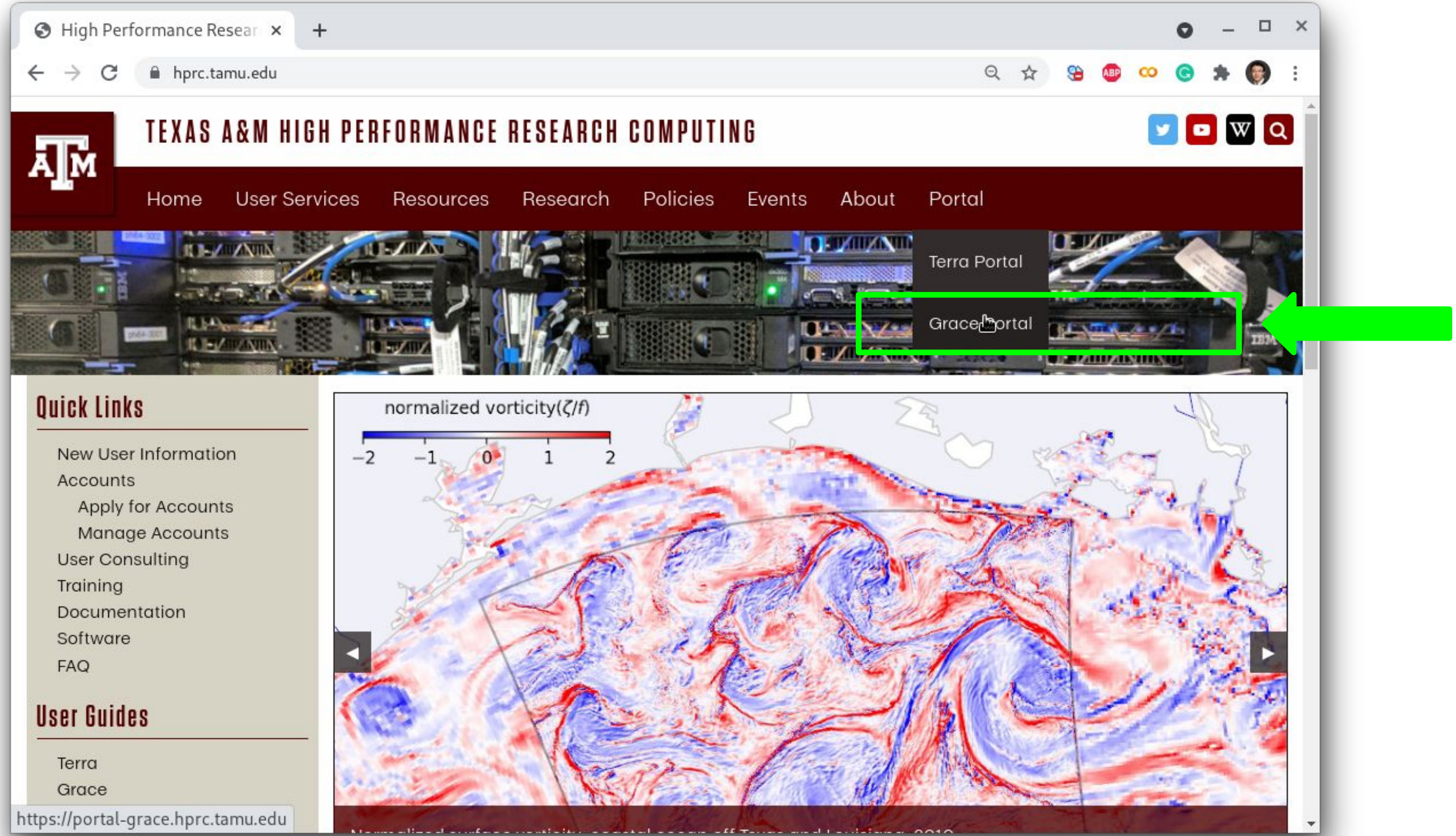
Connecting to HPRC Portal



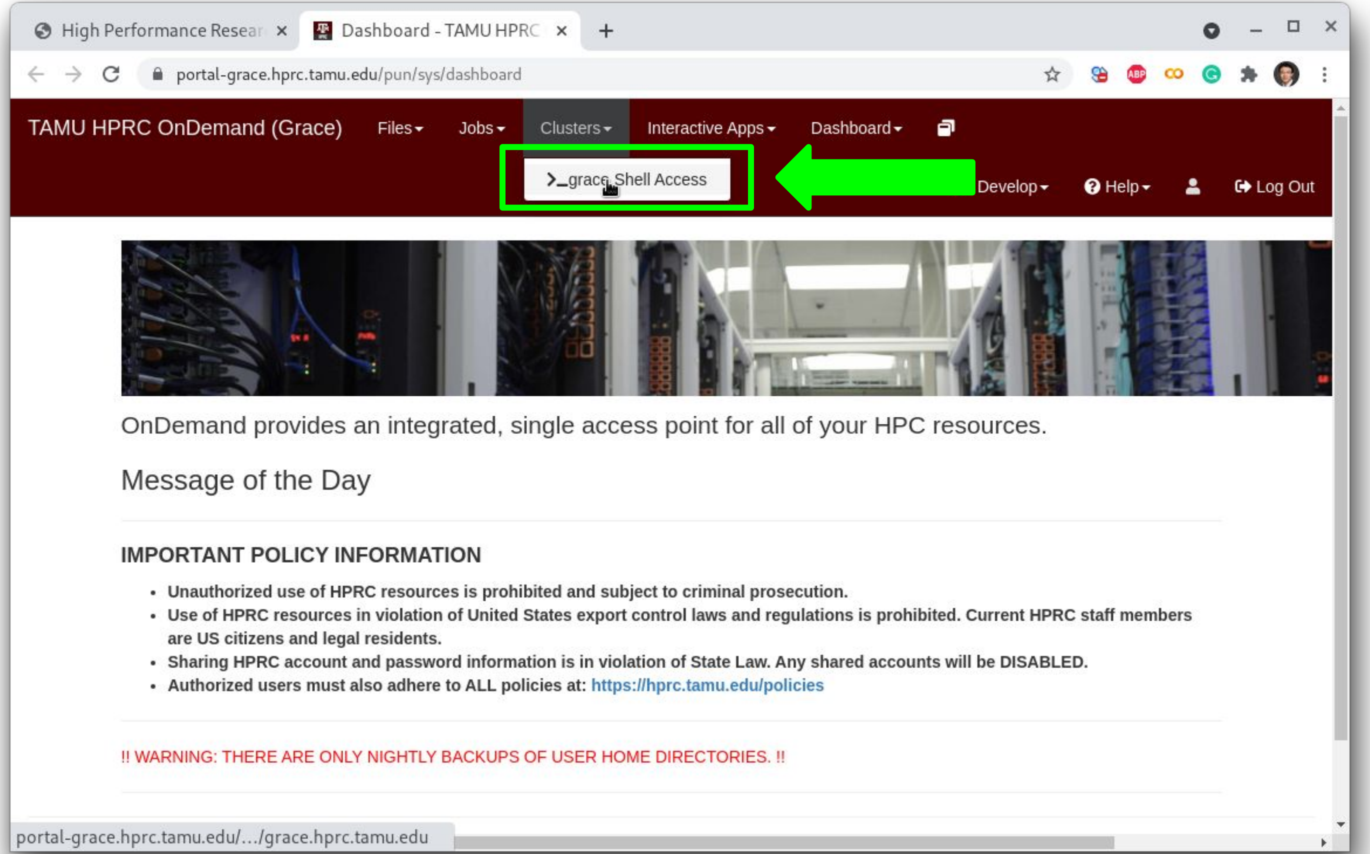
HPRC Portal

*** VPN is required for off-campus users.**

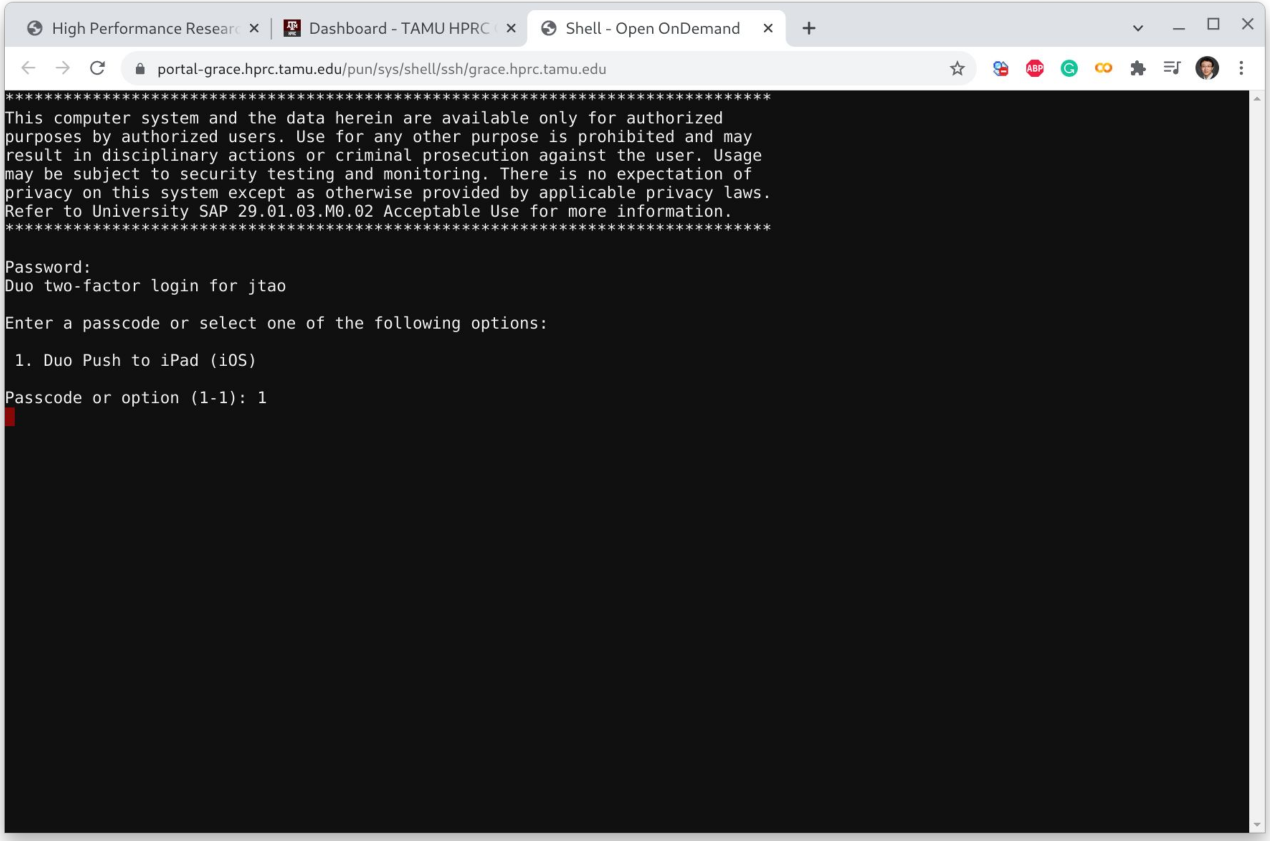
Login HPRC Portal (Grace)



Grace Shell Access - Portal



Grace Shell Access - Shell



Using Pre-installed Julia Module

Step 1. Find the module to be loaded

$ module spider julia
...
Description:
Julia is a high-level, high-performance dynamic programming language
for numerical computing

Versions:
Julia/1.5.3-linux-x86_64
Julia/1.6.1-linux-x86_64
...

(You can also use the [web-based interface](#) to find software modules available on HPRC systems.)

Step 2. Load the module

$ module load Julia/1.6.1-linux-x86_64

Step 3. Start Julia REPL

$ julia

```
[jtao@grace2 ~]$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.1 (2021-04-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

[SW:Julia - TAMU HPRC](#)

Using Your Own Julia Installation

Step 1. Find the version to be installed

Current stable release: v1.6.3 (Sep 23, 2021)

Checksums for this release are available in both MD5 and SHA256 formats.

Windows [help]	64-bit (installer), 64-bit (portable)	32-bit
macOS [help]	64-bit	
Generic Linux on x86 [help]	64-bit (GPG), 64-bit (musl)[1] (GPG)	32-bit
Generic Linux on ARM [help]	64-bit (AArch64) (GPG)	32-bit
Generic Linux on PowerPC [help]	64-bit (little endian) (GPG)	
Generic FreeBSD on x86 [help]	64-bit (GPG)	
Source	Tarball (GPG)	Tarbal

(You can find different versions of Julia at [Download Julia](), The latest stable version of Julia is highly recommended.)

[SW:Julia - TAMU HPRC]()

Step 2. Download & Unzip

$ cd $SCRATCH
$ wget https://.../julia-1.6.3-linux-x86_64.tar.gz
$ tar -zxvf julia-1.6.3-linux-x86_64.tar.gz

Step 3. Start Julia REPL

$ cd $SCRATCH/julia-1.6.3/bin; ./julia

```
[jtao@grace2 jtao]$ cd $SCRATCH/julia-1.6.3/bin; ./julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.3 (2021-09-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

Install Julia Packages

```
# export Julia Depot path (default to ~/.julia)
$export JULIA_DEPOT_PATH=$SCRATCH/.julia

# start Julia
$julia

# type ']' to open Pkg REPL
# press backspace or ^C to quit Pkg REPL.
julia>]
(@v1.6) pkg> add Plots
```

Julia - Quickstart

The julia program starts the interactive **REPL**. You will be immediately switched to the **shell mode** if you type a **semicolon**. A **question mark** will switch you to the **help mode**. The <TAB> key can help with autocompletion.

```
julia> versioninfo()
julia> VERSION
```

Special symbols can be typed with the **escape symbol and** <TAB>, but they might not show properly on the web-based terminal.

```
julia> \sqrt <TAB>
julia> for i ∈ 1:10 println(i) end  #\in <TAB>
```

Julia REPL Keybindings

Keybinding	Descrition
^d	Exit (when buffer is empty)
^c	Interrupt or cancel
^l	Clear console screen
Return/Enter, ^J	New line, executing if it is complete
? or ;	Enter help or shell mode (when at start of a line)
^R, ^S	Incremental history search
]	Enter Pkg REPL
Backspace or ^c	Quit Pkg REPL

Part III.
Julia as an Advanced Calculator

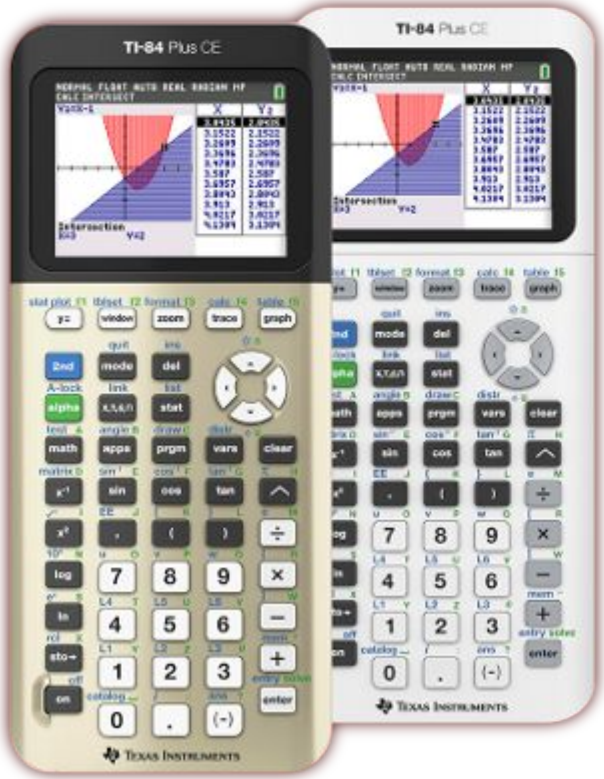


Image Credit: http://www.ti.com/

Arithmetic Operators

+	Addition (also unary plus)
-	Subtraction (also unary minus)
*	multiplication
/	division
\	inverse division
%	mod
^	to the power of

More about Arithmetic Operators

1. The **order of operations** follows the math rules.
2. The **updating version** of the operators is formed by placing a "**=**" immediately after the operator. For instance, **x+=3** is equivalent to **x=x+3**.
3. **Unicode** could be defined as operator.
4. **A "dot" operation** is automatically defined to perform the operation element-by-element on arrays in every binary operation.
5. **Numeric Literal Coefficients**: Julia allows variables to be immediately preceded by a numeric literal, implying multiplication.

Arithmetic Expressions

Some examples:

```
julia> 10/5*2
julia> 5*2^3+4\2
julia> -2^4
julia> 8^1/3
julia> pi*ℯ #\euler <TAB>
julia> x=1; x+=3.1
julia> x=[1,2]; x = x.^(-2)
```

Relational Operators

```
==       True, if it is equal
!=,≠     True, if not equal to #\ne <TAB>
<        less than
>        greater than
<=,≤     less than or equal to #\le <TAB>
>=,≥     greater than or equal to #\ge <TAB>
```

** try ≠(4,5), what does this mean? How about !=(4,5)*

Boolean and Bitwise Operators

```
&&              Logical and
||              Logical or
!               Not
⊻, xor()        Exclusive OR
|               Bitwise OR
~               Negate
&               Bitwise And
>>              Right shift
<<              Left shift
```

NaN and Inf

NaN is a not-a-number value of type Float64.

Inf is positive infinity of type Float64.

-Inf is negative infinity of type Float64.

- **Inf** is equal to itself and greater than everything else except **NaN**.
- **-Inf** is equal to itself and less then everything else except **NaN**.
- **NaN** is not equal to, not less than, and not greater than anything, including itself.

```
julia> NaN == NaN  #false

julia> NaN != NaN
true

julia> NaN < NaN
false

julia> NaN > NaN
false

julia> isequal(NaN, NaN)
true

julia> isnan(1/0)
false
```

Variables

The basic types of Julia include **float**, **int**, **char, string**, and **bool**. A global variable can not be deleted, but its content could be cleared with the keyword **nothing.** Unicode can be used as variable names!

```
julia> b = true; typeof(b)
julia> varinfo()
julia> x = "Hi"; x > "He"        # x='Hi' is wrong. why?
julia> y = 10
julia> z = complex(1, y)
julia> println(b, x, y, z)
julia> b = nothing; show(b)
julia> 🏈=2; 🏃=1              # \:football <TAB> \:runner: <TAB>
```

Naming Rules for Variables

Variable names must begin with a letter or underscore

```
julia> 4c = 12
```

Names can include any combinations of letters, numbers, underscores, and exclamation symbol. Some unicode characters could be used as well

```
julia> c_4 = 12; δ = 2
```

Maximum length for a variable name is not limited

Julia is case sensitive. The variable name **A** is different than the variable name **a**.

Displaying Variables

We can display a variable (i.e., show its value) by simply typing the name of the variable at the command prompt (leaving off the semicolon).

We can also use **`print`** or **`println`** (print plus a new line) to display variables.

```
julia> print("The value of x is:"); print(x)
julia> println("The value of x is:"); print(x)
```

Exercise

Create two variables: **`a = 4`** and **`b = 17.2`**

Now use Julia to perform the following set of calculations:

$(b+5.4)^{1/3}$

$b^2-4b+5a$

`a>b && a>1.0`

`a!=b`

Basic Syntax for Statements (I)

1. Comments start with **'#'**

2. Compound expressions with **`begin`** blocks and **`(;)`** chains

```
julia>  z = begin
               x = 1
               y = 2
               x + y
           end
julia>  z = (x = 1; y = 2; x + y)
```

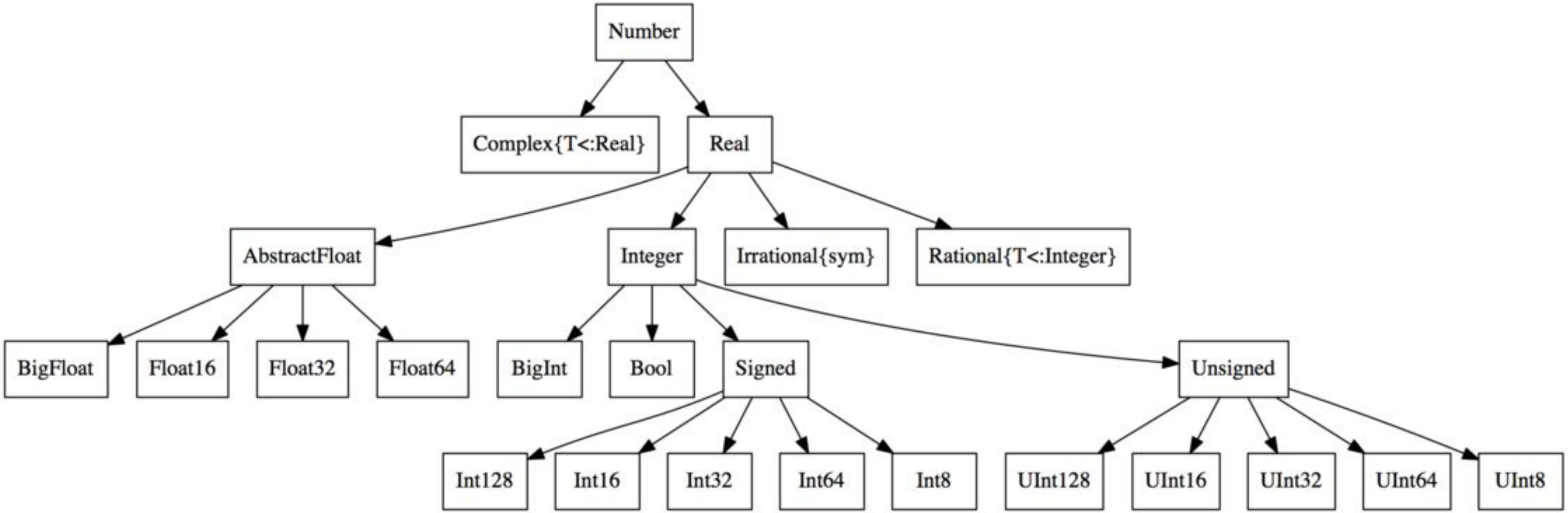
Basic Syntax for Statements (II)

The statements could be freely arranged with an optional `;` if a new line is used to separate statements.

```
julia> begin x = 1; y = 2; x + y end

julia> (x = 1;
        y = 2;
        x + y)
```

Numerical Data Types



Integer Data Types

Type	Signed?	Number of bits	Smallest value	Largest value
Int8	✓	8	-2^7	2^7 - 1
UInt8		8	0	2^8 - 1
Int16	✓	16	-2^15	2^15 - 1
UInt16		16	0	2^16 - 1
Int32	✓	32	-2^31	2^31 - 1
UInt32		32	0	2^32 - 1
Int64	✓	64	-2^63	2^63 - 1
UInt64		64	0	2^64 - 1
Int128	✓	128	-2^127	2^127 - 1
UInt128		128	0	2^128 - 1
Bool	N/A	8	false (0)	true (1)

Handling Big Integers

An overflow happens when a number goes beyond the representable range of a given type. Juliat provides **BigInt** type to handle big integers.

```
julia> x = typemax(Int64)
julia> x + 1
julia> x + 1 == typemin(Int64)
julia> x = big(typemax(Int64))^100
```

Floating Point Data Types

Type	Precision	Number of bits	Range
Float16	half	16	-65504 to -6.1035e-05 6.1035e-05 to 65504
Float32	single	32	−3.402823E38 to −1.401298E-45 1.401298E-45 to 3.402823E38
Float64	double	64	-1.79769313486232E308 to -4.94065645841247E-324 4.94065645841247E-324 to 1.79769313486232E308

- Additionally, full support for **Complex** and **Rational Numbers** is built on top of these primitive numeric types.
- All numeric types interoperate naturally without explicit casting thanks to a user-extensible **type promotion system**.

Handling Floating-point Types (I)

Perform each of the following calculations in your head.

```
julia> a = 4/3
julia> b = a - 1
julia> c = 3*b
julia> e = 1 - c
```

What does Julia get?

Handling Floating-point Types (II)

What does Julia get?

```
julia> a = 4/3   #1.3333333333333333

julia> b = a - 1 #0.33333333333333326
julia> c = 3*b   #0.9999999999999998
julia> e = 1 - c #2.220446049250313e-16
```

It is impossible to perfectly represent all real numbers using a finite string of 1's and 0's.

Handling Floating-point Types (III)

Now try the following with BigFloat

```
julia> a = big(4)/3
julia> b = a - 1
julia> c = 3*b
julia> e = 1 - c #-1.7272337110188...e-77
```

Next, set the precision and repeat the above

```
julia> setprecision(4096)
```

BigFloat variables can store floating point data with arbitrary precision with a performance cost.

Complex and Rational Numbers

The global constant **im** is bound to the complex number **i**, representing the principal square root of **-1**.

```
julia> 2(1 - 1im)

julia> sqrt(complex(-1, 0))
```

Note that **`3/4im == 3/(4*im) == -(3/4*im)`**, since a literal coefficient binds more tightly than division. **`3/(4*im)!=(3/4*im)`**

Julia has a **rational number** type to represent exact ratios of integers. Rationals are constructed using the `//` operator, e.g., **`9//27`**

Some Useful Math Functions

Rounding and division functions

Function	Descrition
round(x)	round x to the nearest integer
floor(x)	round x towards -Inf
ceil(x)	round x towards +Inf
trunc(x)	round x towards zero
div(x,y)	truncated division; quotient rounded towards zero
fld(x,y)	floored division; quotient rounded towards -Inf
cld(x,y)	ceiling division; quotient rounded towards +Inf
rem(x,y)	remainder; satisfies x == div(x,y)*y + rem(x,y); sign matches x
gcd(x,y...)	greatest positive common divisor of x, y,...
lcm(x,y...)	least positive common multiple of x, y,...

Sign and absolute value functions

Function	Description
abs(x)	a positive value with the magnitude of x
abs2(x)	the squared magnitude of x
sign(x)	indicates the sign of x, returning -1, 0, or +1
signbit(x)	indicates whether the sign bit is on (true) or off (false)
copysign(x,y)	a value with the magnitude of x and the sign of y
flipsign(x,y)	a value with the magnitude of x and the sign of x*y

* *The built-in math functions in Julia are implemented in C(openlibm).*

Chars and Strings

Julia has a first-class type representing a single character, called **Char.** Single quotes are & double quotes are used different in Julia.

```
julia> a = 'H'  #a is a character object
julia> b = "H"  #a is a string with length 1
```

Strings and Chars can be easily manipulated with built-in functions.

```
julia> c = string('s') * string('d')
julia> length(c); d = c^10*"4"; split(d,"s")
```

Handling Strings (I)

1. The built-in type used for strings in Julia is **String**. This supports the full range of Unicode characters via the UTF-8 encoding.
2. Strings are **immutable.**
3. A **Char** value represents a single character.
4. One can do comparisons and a limited amount of arithmetic with Char.
5. All indexing in Julia is **1-based**: the first element of any integer-indexed object is found at index 1.

```
julia> str = "Hello, world!"
julia> c = str[1]       #c = 'H'
julia> c = str[end]     #c = '!'
julia> c = str[2:8]     #c = "ello, w"
```

Handling Strings (II)

Interpolation: Julia allows interpolation into string literals using **$**, as in Perl. To include a literal **$** in a string literal, escape it with a backslash:

```
julia> "1 + 2 = $(1 + 2)"  #"1 + 2 = 3"
julia> print("\$100 dollars!\n")
```

Triple-Quoted String Literals: no need to escape for special symbols and trailing whitespace is left unaltered.

Handling Strings (III)

Julia comes with a collection of tools to handle strings.

```
julia> str="Julia"
julia> occursin("lia", str)
julia> z = repeat(str, 10)
julia> firstindex(str)
julia> lastindex(str)
julia> length(str)
```

Julia also supports Perl-compatible regular expressions (regexes).

```
julia> ismatch(r"^\s*(?:#|$)", "# a comment")
```

Help

- For help on a specific function or macro, type **?** followed by its name, and press enter. This only works if you know the name of the function you want help with. With **^S** and **^R** you can also do historical search.

```
Julia> ?cos
```

- Type **`?help`** to get more information about help

```
Julia> ?help
```

Part IV. Functions

```
function mandelbrot(a)
   z = 0
   for i=1:50
       z = z^2 + a
   end
   return z
end
for y=1.0:-0.05:-1.0
   for x=-2.0:0.0315:0.5
       abs(mandelbrot(complex(x, y))) < 2 ?
print("*") : print(" ")
   end
   println()
end
```

Definition of Functions

Two equivalent ways to define a function

```
julia> function func(x,y)
           return x + y, x
       end

julia> Σ(x,y) = x + y, x
```

Operators are functions

```
julia> +(1,2); plusfunc=+
Julia> plusfunc(2,3)
```

Recommended style for function definition: **append ! to names of functions that modify their arguments**

Functions with Optional Arguments

You can define functions with optional arguments with default values.

```
julia> function point(x, y, z=0)
           println("$x, $y, $z")
       end
julia> point(1,2); point(1,2,3)
```

Keywords and Positional Arguments

Keywords can be used to label arguments. Use a **semicolon** after the function's unlabelled arguments, and follow it with one or more **keyword=value** pairs

```
julia> function func(a, b, c="one"; d="two")
          println("$a, $b, $c, $d")
       end
julia> func(1,2); func(d="four", 1, 2, "three")
```

Anonymous Functions

As functions in Julia are first-class objects, they can be created anonymously without a name.

```
julia> x -> 2x - 1
julia> function (x)
           2x - 1
          end
```

An anonymous function is primarily used to feed in other functions.

```
julia> map((x,y,z) -> x + y + z,
           [1,2,3], [4, 5, 6], [7, 8, 9])
```

"Dotted" Function

Dot syntax can be used to vectorize functions, i.e., applying functions **elementwise** to arrays.

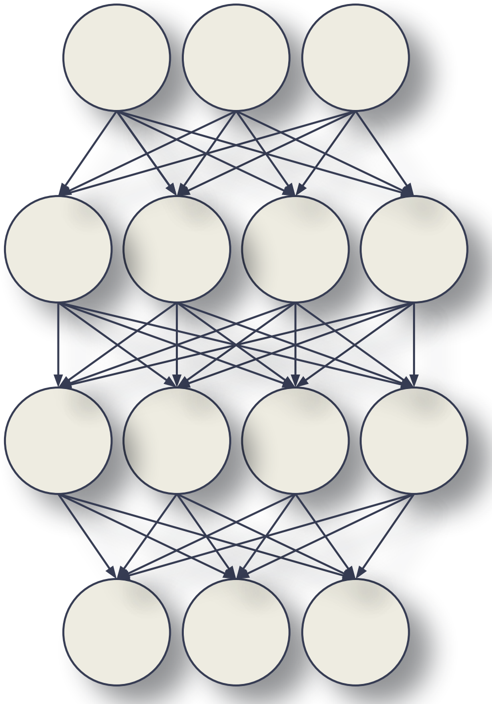
```
julia> func(a, b) = a * b
julia> func(1, 2)
julia> func.([1,2], 3)
julia> sin.(func.([1,2],[3,4]))
```

Function of Function

Julia functions can be treated the same as other Julia objects. You can return a function within a function.

```
julia> function my_exp_func(x)
           f = function (y) return y^x end
           return f
       end
julia> squarer=my_exp_func(2); quader=my_exp_func(3)
julia> squarer(3)
julia> quader(3)
```

Part V.
Data Structures: Tuples, Arrays, Sets, and Dictionaries



Tuples

A tuple is an ordered sequence of elements. Tuples are good for small fixed-length collections. Tuples are **immutable**.

```
julia> t = (1, 2, 3)
julia> t = ((1, 2), (3, 4))
julia> t[1][2]
```

Arrays

An array is an ordered collection of elements. In Julia, arrays are used for lists, vectors, tables, and matrices. Arrays are **mutable**.

```
julia> a = [1, 2, 3]      # column vecor
julia> b = [1 2 3]        # row vector
julia> c = [1 2 3; 4 5 6] # 2x3 vector
julia> d = [n^2 for n in 1:5]
julia> f = zeros(2,3); g = rand(2,3)
julia> h = ones(2,3); j = fill("A",9)
julia> k = reshape(rand(5,6),10,3)
julia> [a a]                  # hcat
julia> [b;b]                  # vcat
```

Array & Matrix Operations

Many Julia operators and functions can be used preceded with a dot. These versions are the same as their non-dotted versions, and work on the arrays element by element.

```
julia> b = [1 2 3; 4 5 7; 7 8 9]
julia> b .+ 10     # each element + 10
julia> sin.(b)     # sin function
julia> b'              # transpose (transpose(b))
julia> inv(b)          # inverse
julia> b * b           # matrix multiplication
julia> b .* b          # element-wise multiplication
julia> b .^ 2          # element-wise square
```

Sets

Sets are mainly used to eliminate repeated numbers in a sequence/list.
It is also used to perform some standard set operations.
A could be created with the Set constructor function

Examples:

```
julia> months=Set(["Nov","Dec","Dec"])
julia> typeof(months)
julia> push!(months,"Sept")
julia> pop!(months,"Sept")
julia> in("Dec", months)
julia> m=Set(["Dec","Mar","Feb"])
julia> union(m,months)
julia> intersect(m,months)
julia> setdiff(m,months)
```

Dictionaries

Dictionaries are mappings between keys and items stored in the dictionaries. Alternatively one can think of dictionaries as sets in which something stored against every element of the set. To define a dictionary, use **`Dict()`**

Examples:

```
julia> m=Dict("Oct"=>"October",
              "Nov"=>"November",
                 "Dec"=>"December")
julia> m["Oct"]
julia> get(m, "Jan", "N/A")
julia> haskey(m, "Jan")
julia> m["Jan"]="January"
julia> delete!(m, "Jan")
julia> keys(m)
julia> values(m)
julia> map(uppercase, collect(keys(m)))
```

Part VI.
Conditional Statements & Loops

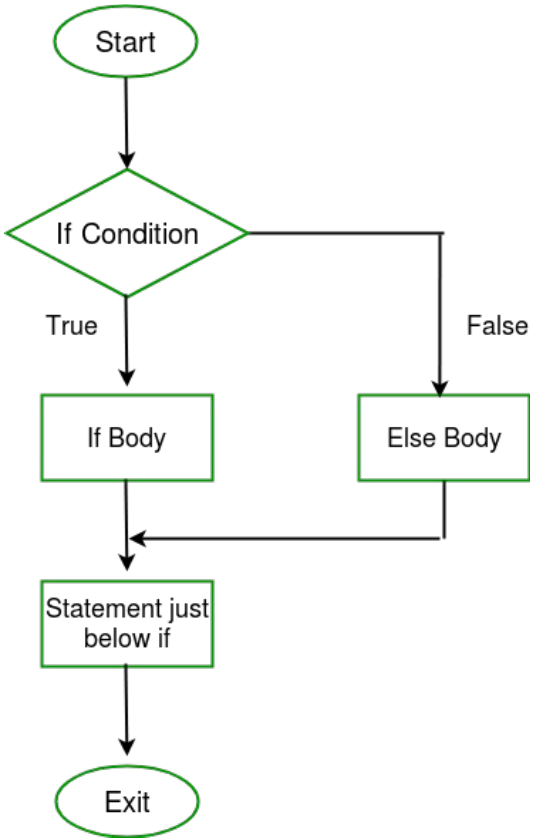


Image Credit: https://www.geeksforgeeks.org

Controlling Blocks

Julia has the following controlling constructs

- **ternary** expressions
- **boolean switching** expressions
- **if elseif else end** - conditional evaluation
- **for end** - iterative evaluation
- **while end** - iterative conditional evaluation
- **try catch error throw** exception handling

Ternary and Boolean Expressions

A ternary expression can be constructed with the ternary operator **"?"** and **":"**,

```
julia> x = 1
julia> x > 0 ? sin(x) : cos(x)
```

You can combine the boolean condition and any expression using **&&** or **||**,

```
julia> isodd(42) && println("That's even!")
```

Conditional Statements

Execute statements if condition is true.

There is no "**switch**" and "**case**" statement in Julia.

There is an "**ifelse**" statement.

```
julia> a = 8
julia> if a>10
           println("a > 10")
       elseif a<10
           println("a < 10")
       else
            println("a = 10")
       end
```

```
julia> s = ifelse(false, "hello", "goodbye") * " world"
```

Loop Control Statements - *for*

for statements help repeatedly execute a block of code for a certain number of iterations. Loop variables are local.

```
julia> for i in 0:1:10
           if i % 3 == 0
               continue
           end
           println(i)
       end
julia> for l in "julia"
           print(l, "-^-")
       end
```

Other Usage of *for* Loops

Array comprehension:

```
julia> [n for n in 1:10]
```

Array enumeration:

```
julia> [i for i in enumerate(rand(3))]
```

Generator expressions:

```
julia> sum(x for x in 1:10)
```

Nested loop:

```
for x in 1:10, y in 1:10
    @show (x, y)
    if y % 3 == 0
        break
    end
end
```

Loop Control Statements - *while*

while statements repeatedly execute a block of code as long as a condition is satisfied.

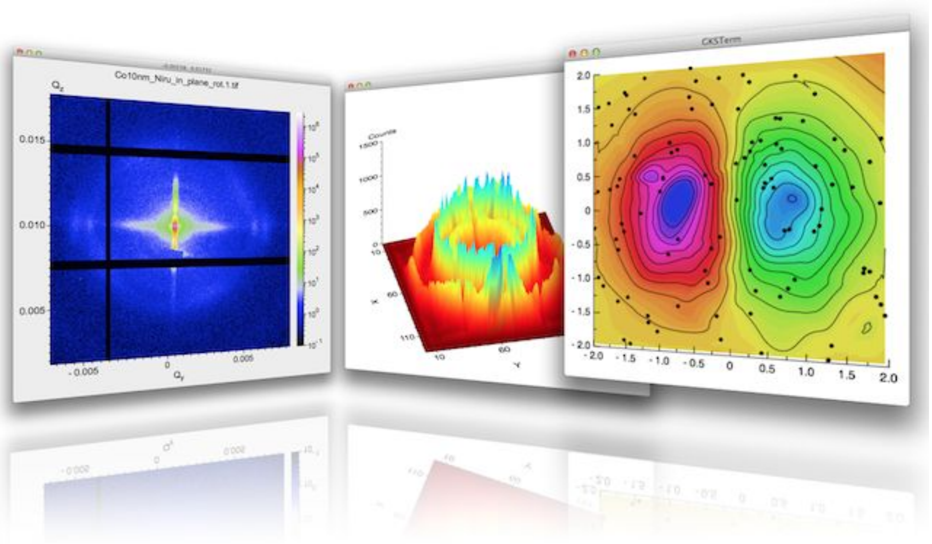
```
julia> n = 1
julia> s = 0
julia> while n <= 100
           s = s + n
           n = n + 1
       end
julia> println(s)
```

Exception Handling Blocks

try ... catch construction checks for errors and handles them gracefully,

```
julia> s = "test"
julia> try
          s[1] = "p"
       catch
          println("caught an error: $e")
          println("continue with execution!")
       end
```

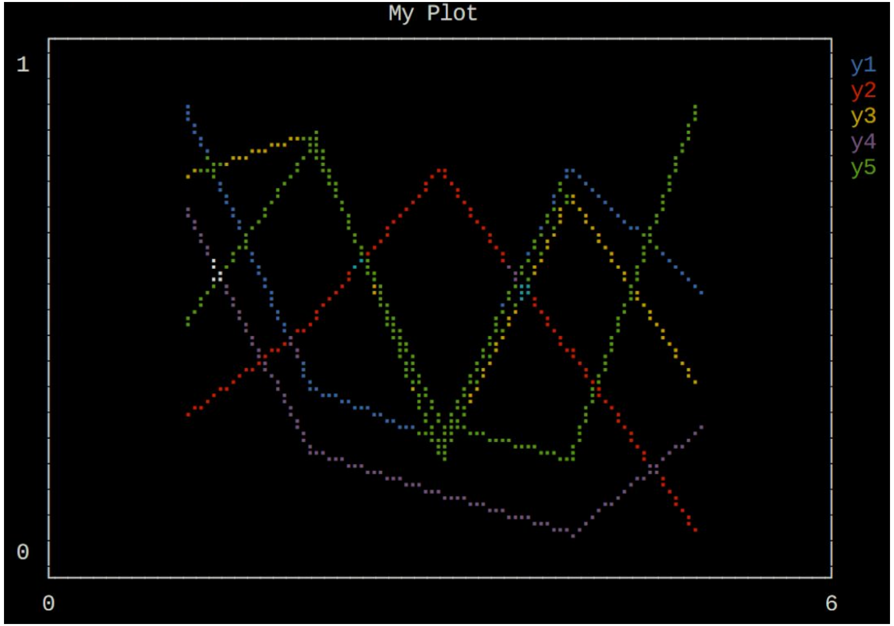
Part VII.
Plot with Julia
UnicodePlots



UnicodePlots

UnicodePlots is simple and lightweight and it plots directly in your terminal.

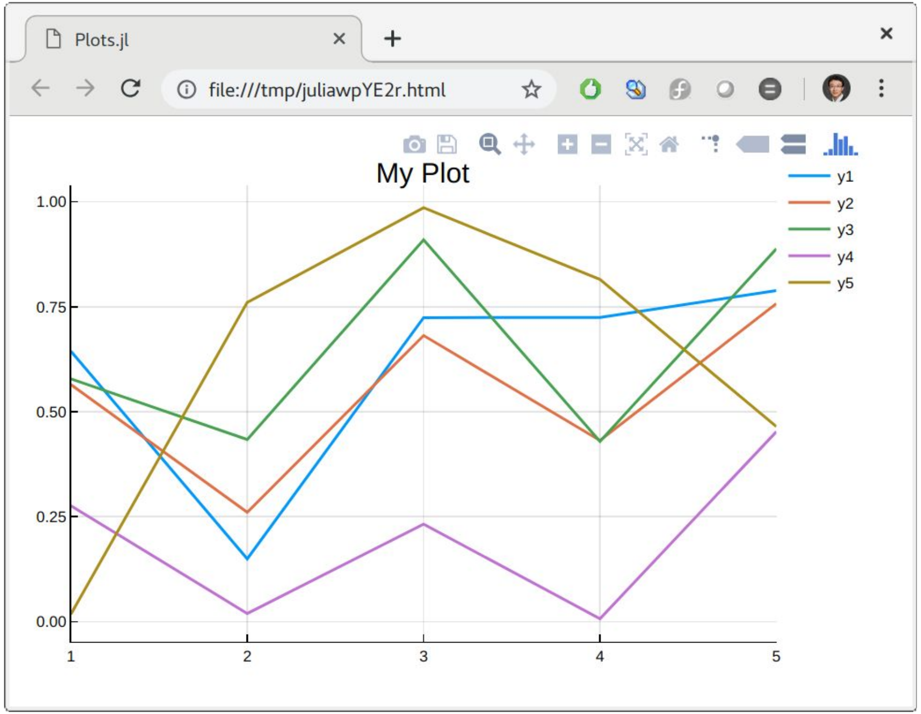
```
julia> using Plots
julia> unicodeplots()
julia> plot(rand(5,5),
linewidth=2, title="My
Plot")
```



Plotly Julia Library

Plotly creates leading open source software for Web-based data visualization and analytical apps. Plotly Julia Library makes interactive, publication-quality graphs online.

```
julia> using Plots
julia> plotly()
julia> plot(rand(5,5), linewidth=2, title="My Plot")
```



GR Framework

GR framework is a universal framework for cross-platform visualization applications.

```
julia> using Plots
julia> gr()
julia> plot(rand(5,5), linewidth=4, title="My Plot", size=(1024,1024))
```

Online Resources

Official Julia Document

https://docs.julialang.org/en/v1/

Julia Online Tutorials

https://julialang.org/learning/

Introducing Julia (Wikibooks.org)

https://en.wikibooks.org/wiki/Introducing_Julia

MATLAB–Python–Julia cheatsheet

https://cheatsheets.quantecon.org/

The Fast Track to Julia

https://juliadocs.github.io/Julia-Cheat-Sheet/

Acknowledgements

- The slides are created based on the materials from Julia official website and the Wikibook *Introducing Julia* at wikibooks.org.
- Supports from Texas A&M Engineering Experiment Station (TEES) and High Performance Research Computing (HPRC).

Appendix

Modules and Packages

Julia code is organized into **files**, **modules**, and **packages**. Files containing Julia code use the **.jl** file extension. Modules can be defined as

```
module MyModule
    ...
end
```

Julia manages its packages with **Pkg**

```
julia> Pkg.add("MyPackage")
julia> Pkg.status()
julia> Pkg.update()
julia> Pkg.rm("MyPackage")
```

ASCII Code

When you press a key on your computer keyboard, the key that you press is translated to a binary code.

```
A = 1000001        (Decimal = 65)
a = 1100001        (Decimal = 97)
0 = 0110000        (Decimal = 48)
```

ASCII Code

ASCII stands for **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Terminology

A **bit** is short for **bi**nary digi**t.** It has only two possible values: On (1) or Off (0).

A **byte** is simply a string of 8 bits.

A **kilobyte** (KB) is 1,024 (2^10) bytes.

A **megabyte** (MB) is 1,024 KB or 1,024^2 bytes.

A **gigabyte** (GB) is 1,024 MB or 1,024^3 bytes.

How Computers Store Variables

Computers store all data (numbers, letters, instructions, …) as strings of 1s and 0s (bits).

A **bit** is short for **bi**nary digi**t.** It has only two possible values: On (1) or Off (0).