Introduction to Julia Programming Language

Jian Tao
jtao@tamu.edu
Fall 2018 HPRC Short Course
09/21/2018
# Upcoming Short Courses

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Course</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sept 21</td>
<td>2:30 to 5:00 PM</td>
<td>Introduction to Deep Learning with TensorFlow</td>
<td>Jian Tao</td>
</tr>
<tr>
<td></td>
<td>6:00 to 8:45 PM</td>
<td>Linux Workshop</td>
<td>Lisa Perez</td>
</tr>
<tr>
<td>Sep 25</td>
<td>10 AM to 12:30 PM</td>
<td>Introduction to Next Generation Sequencing</td>
<td>Michael Dickens</td>
</tr>
<tr>
<td>Sep 28</td>
<td>11:30 to 2:00 PM</td>
<td>Introduction to Matlab</td>
<td>Jian Tao</td>
</tr>
<tr>
<td></td>
<td>2:30 to 5:00 PM</td>
<td>Python for Matlab Users</td>
<td>Jian Tao</td>
</tr>
</tbody>
</table>

[https://hprc.tamu.edu/training/](https://hprc.tamu.edu/training/)
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 v 1.0 in Aug 2018
- [https://julialang.org/](https://julialang.org/)
- customized for "greedy, unreasonable, demanding programmers".
The image shows a graph comparing the performance of various programming languages, with benchmarks on the y-axis and different programming languages on the x-axis. The y-axis represents the performance in 

Image Credit: Julialang.org
**Julia** is a free and open source project, with

- more than 700 active open source contributors,
- 1,900 registered packages,
- 41,000 GitHub stars,
- 2 million downloads,
- used at more than 700 universities and research institutions,
- used at companies such as Aviva, BlackRock, Capital One, and Netflix.
Image Credit: RedMonk (https://redmonk.com/sogrady/2018/08/10/language-rankings-6-18/)
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.

"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.
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.
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>

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

<table>
<thead>
<tr>
<th>Keybinding</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^D</td>
<td>Exit (when buffer is empty)</td>
</tr>
<tr>
<td>^C</td>
<td>Interrupt or cancel</td>
</tr>
<tr>
<td>^L</td>
<td>Clear console screen</td>
</tr>
<tr>
<td>Return/Enter, ^J</td>
<td>New line, executing if it is complete</td>
</tr>
<tr>
<td>? or ;</td>
<td>Enter help or shell mode (when at start of a line)</td>
</tr>
<tr>
<td>^R, ^S</td>
<td>Incremental history search</td>
</tr>
</tbody>
</table>
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

Exploring the Lorenz System

In this Notebook we explore the Lorenz system of differential equations:

\[\begin{align*}
\dot{x} &= \sigma(y - x) \\
\dot{y} &= x(\rho - z) - y \\
\dot{z} &= xy - \beta z
\end{align*}\]

This is one of the classic systems in non-linear differential equations. It exhibits a range of complex behaviors as the parameters (\(\sigma\), \(\rho\), \(\beta\)) are varied, including what are known as chaotic solutions. The system was originally developed as a simplified mathematical model for atmospheric convection in 1963.

Image Credit: Jupyter (http://jupyter.org/)
Julia as an Advanced Calculator
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. E.g., \( ÷= \) is equivalent to \( /= \).
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
 julia> x=1; x+=3.1
 julia> x=[1,2]; x.^=-2
```
Relational Operators

==    True, if it is equal
!=,≠    True, if not equal to
<        less than
>        greater than
<=,≤    less than or equal to
>=,≥    greater than or equal to
Boolean and Bitwise Operators

&& Logical and
|| Logical or
! Not
prov, 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 than everything else except **NaN**.
- **NaN** is not equal to, not less than, and not greater than anything, including itself.

```julia
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`.

```julia
julia> b = true; typeof(b)
julia> whos()
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; println(b)
```
Naming Rules for Variables

- Variable names must begin with a letter or underscore
  \[\text{julia}> \text{4c} = 12\]
- Names can include any combinations of letters, numbers, underscores, and exclamation symbol. Some unicode characters could be used as well
  \[\text{julia}> \text{c_4} = 12; \text{δ} = 2\]
- Maximum length for a variable name is not limited
- Python is case sensitive. The variable name A is different than the variable name a.
Exercise

Create two variables: \( a = 4 \) and \( b = 17.2 \)

Now use Julia to perform the following set of calculations:

\[
(b+5.4)^{1/3} \quad \quad b^2-4b+5a \\
 a>b \quad \text{and} \quad a>1.0 \quad \quad a\neq b
\]
Basic Syntax for Statements (I)

1. Comments start with '#'
2. No spaces or tabs are allowed between function name
3. Compound expressions with **begin** blocks and (;) chains

```julia
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)
```
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:"); println(x)
```
Numerical Data Types

- **Number**
  - **Complex\{T<:Real\}
  - **Real**
    - **AbstractFloat**
      - **BigFloat**
      - **Float16**
      - **Float32**
      - **Float64**
    - **Integer**
    - **Irrational\{sym\}
    - **Rational\{T<:Integer\}
  - **Signed**
    - **Int128**
    - **Int16**
    - **Int32**
    - **Int64**
    - **Int8**
  - **Unsigned**
    - **UInt128**
    - **UInt16**
    - **UInt32**
    - **UInt64**
    - **UInt8**
# Integer Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Signed?</th>
<th>Number of bits</th>
<th>Smallest value</th>
<th>Largest value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int8</td>
<td>✓</td>
<td>8</td>
<td>-2^7</td>
<td>2^7 - 1</td>
</tr>
<tr>
<td>UInt8</td>
<td></td>
<td>8</td>
<td>0</td>
<td>2^8 - 1</td>
</tr>
<tr>
<td>Int16</td>
<td>✓</td>
<td>16</td>
<td>-2^15</td>
<td>2^15 - 1</td>
</tr>
<tr>
<td>UInt16</td>
<td></td>
<td>16</td>
<td>0</td>
<td>2^16 - 1</td>
</tr>
<tr>
<td>Int32</td>
<td>✓</td>
<td>32</td>
<td>-2^31</td>
<td>2^31 - 1</td>
</tr>
<tr>
<td>UInt32</td>
<td></td>
<td>32</td>
<td>0</td>
<td>2^32 - 1</td>
</tr>
<tr>
<td>Int64</td>
<td>✓</td>
<td>64</td>
<td>-2^63</td>
<td>2^63 - 1</td>
</tr>
<tr>
<td>UInt64</td>
<td></td>
<td>64</td>
<td>0</td>
<td>2^64 - 1</td>
</tr>
<tr>
<td>Int128</td>
<td>✓</td>
<td>128</td>
<td>-2^127</td>
<td>2^127 - 1</td>
</tr>
<tr>
<td>UInt128</td>
<td></td>
<td>128</td>
<td>0</td>
<td>2^128 - 1</td>
</tr>
<tr>
<td>Bool</td>
<td>N/A</td>
<td>8</td>
<td>false (0)</td>
<td>true (1)</td>
</tr>
</tbody>
</table>
Handling Big Integers

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

```julia
julia> x = typemax(Int64)
julia> x + 1
julia> x + 1 == typemin(Int64)
julia> x = big(typemax(Int64))^100
```
### Floating Point Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Precision</th>
<th>Number of bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float16</td>
<td>half</td>
<td>16</td>
<td>-65504 to -6.1035e-05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6.1035e-05 to 65504</td>
</tr>
<tr>
<td>Float32</td>
<td>single</td>
<td>32</td>
<td>-3.402823E38 to -1.401298E-45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.401298E-45 to 3.402823E38</td>
</tr>
<tr>
<td>Float64</td>
<td>double</td>
<td>64</td>
<td>-1.79769313486232E308 to -4.94065645841247E-324</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.94065645841247E-324 to 1.79769313486232E308</td>
</tr>
</tbody>
</table>

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

What does Python 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   #0.99999999999999998
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
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
julia> setprecision(4096)
```

BigFloat variables can store floating point data with arbitrary precision with a performance cost.
Complex and Rational Numbers

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

\begin{verbatim}
    julia> 2(1 - 1im)
    julia> sqrt(complex(-1, 0))
\end{verbatim}

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

Julia has a \textbf{rational number} type to represent exact ratios of integers. Rationals are constructed using the \texttt{//} operator, e.g., \texttt{9//27}
# Some Useful Math Functions (I)

## Rounding and division functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>round(x)</td>
<td>round x to the nearest integer</td>
</tr>
<tr>
<td>floor(x)</td>
<td>round x towards -( \infty )</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>round x towards +( \infty )</td>
</tr>
<tr>
<td>trunc(x)</td>
<td>round x towards zero</td>
</tr>
<tr>
<td>div(x,y)</td>
<td>truncated division; quotient rounded towards zero</td>
</tr>
<tr>
<td>fld(x,y)</td>
<td>floored division; quotient rounded towards -( \infty )</td>
</tr>
<tr>
<td>cld(x,y)</td>
<td>ceiling division; quotient rounded towards +( \infty )</td>
</tr>
<tr>
<td>rem(x,y)</td>
<td>remainder; satisfies ( x = \text{div}(x,y) \cdot y + \text{rem}(x,y) ); sign matches x</td>
</tr>
</tbody>
</table>

## Sign and absolute value functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>a positive value with the magnitude of x</td>
</tr>
<tr>
<td>abs2(x)</td>
<td>the squared magnitude of x</td>
</tr>
<tr>
<td>sign(x)</td>
<td>indicates the sign of x, returning -1, 0, or +1</td>
</tr>
<tr>
<td>signbit(x)</td>
<td>indicates whether the sign bit is on (true) or off (false)</td>
</tr>
<tr>
<td>copysign(x,y)</td>
<td>a value with the magnitude of x and the sign of y</td>
</tr>
<tr>
<td>flipsign(x,y)</td>
<td>a value with the magnitude of x and the sign of x*y</td>
</tr>
</tbody>
</table>
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
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
julia> str="Julia"; search(str, 'a')
julia> contains(str, "lia")
julia> z = repeat(str, 10)
julia> start(str); endof(str)
julia> length(str)
```

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

```julia
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
Functions
Definition of Functions

Two equivalent ways to define a function

```julia
julia> function f(x,y)
            return x + y, x
        end
julia> Σ(x,y) = x + y, x
```

Operators are functions

```julia
julia> +(1,2); f=+
Julia> f(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
julia> function f(a, b, c="one"; d="two")
    println("$a, $b, $c, $d")
    end
julia> f(1,2); f(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> f(a, b) = a * b
 julia> f(1, 2)
 julia> f.([1,2], 3)
 julia> sin.(f.([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
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)
```
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
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. Tuples are mutable.

````julia
julia> a = [1, 2, 3]    # column vector
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
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
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> b\I          # inverse (inv(b) or I/b)
julia> det(b)      # determinant
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
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)))
```
Conditional Statements & Loops
Controlling Blocks

Julia has the following 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
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"
```
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
```
while statements repeatedly execute a block of code as long as a condition is satisfied.

```julia
julia> n = 1
julia> sum = 0
julia> while n <= 100
    sum = sum + n
    n = n + 1
julia> print(sum)
```
Exception Handling Blocks

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

julia> s = "test"
julia> try
       s[1] = "p"
catch e
       println("caught an error: $e")
       println("continue with execution!")
end
Plots with Julia
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 is a universal framework for cross-platform visualization applications.

```
Julia> using Plots
Julia> gr()
Julia> plot(rand(5,5), linewidth=2, title="My Plot")
```
UnicodePlots is simple and lightweight and it plots directly in your terminal.

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

Official Julia Document
https://docs.julialang.org/en/v1/

Julia Channel on Youtube
https://www.youtube.com/user/JuliaLanguage

Julia Online Tutorials
https://julialang.org/learning/

Introducing Julia (Wikibooks.org)
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.

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

Julia manages its packages through the package `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.

\[ A = 1000001 \quad (\text{Decimal} = 65) \]
\[ a = 1100001 \quad (\text{Decimal} = 97) \]
\[ 0 = 0110000 \quad (\text{Decimal} = 48) \]
# ASCII Code

ASCII stands for American Standard Code for Information Interchange

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>@</td>
<td>34</td>
<td>22</td>
<td>#</td>
<td>66</td>
<td>42</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>#</td>
<td>35</td>
<td>23</td>
<td>$</td>
<td>67</td>
<td>43</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>$</td>
<td>36</td>
<td>24</td>
<td>%</td>
<td>68</td>
<td>44</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>%</td>
<td>37</td>
<td>25</td>
<td>^</td>
<td>69</td>
<td>45</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>^</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>G</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>&amp;</td>
<td>39</td>
<td>27</td>
<td>*</td>
<td>71</td>
<td>47</td>
<td>H</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>*</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>I</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>(</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>J</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>)</td>
<td>42</td>
<td>2A</td>
<td>-</td>
<td>74</td>
<td>4A</td>
<td>K</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>_</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>L</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>+</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>M</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>,</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>N</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>-</td>
<td>46</td>
<td>2E</td>
<td>0</td>
<td>78</td>
<td>4E</td>
<td>O</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>0</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>P</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>/</td>
<td>48</td>
<td>30</td>
<td>1</td>
<td>80</td>
<td>50</td>
<td>Q</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>1</td>
<td>49</td>
<td>31</td>
<td>2</td>
<td>81</td>
<td>51</td>
<td>R</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>2</td>
<td>50</td>
<td>32</td>
<td>3</td>
<td>82</td>
<td>52</td>
<td>S</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>3</td>
<td>51</td>
<td>33</td>
<td>4</td>
<td>83</td>
<td>53</td>
<td>T</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>4</td>
<td>52</td>
<td>34</td>
<td>5</td>
<td>84</td>
<td>54</td>
<td>U</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>5</td>
<td>53</td>
<td>35</td>
<td>6</td>
<td>85</td>
<td>55</td>
<td>V</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>6</td>
<td>54</td>
<td>36</td>
<td>7</td>
<td>86</td>
<td>56</td>
<td>W</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>7</td>
<td>55</td>
<td>37</td>
<td>8</td>
<td>87</td>
<td>57</td>
<td>X</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>8</td>
<td>56</td>
<td>38</td>
<td>9</td>
<td>88</td>
<td>58</td>
<td>Y</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>9</td>
<td>57</td>
<td>39</td>
<td>0</td>
<td>89</td>
<td>59</td>
<td>z</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>0</td>
<td>58</td>
<td>3A</td>
<td></td>
<td>90</td>
<td>5A</td>
<td>A</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td></td>
<td>59</td>
<td>3B</td>
<td></td>
<td>91</td>
<td>5B</td>
<td>B</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td></td>
<td>60</td>
<td>3C</td>
<td></td>
<td>92</td>
<td>5C</td>
<td>C</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td></td>
<td>61</td>
<td>3D</td>
<td></td>
<td>93</td>
<td>5D</td>
<td>D</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td></td>
<td>62</td>
<td>3E</td>
<td></td>
<td>94</td>
<td>5E</td>
<td>E</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td></td>
<td>63</td>
<td>3F</td>
<td></td>
<td>95</td>
<td>5F</td>
<td>F</td>
</tr>
</tbody>
</table>
Terminology

A **bit** is short for **binary digit**. 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 binary digit. It has only two possible values: On (1) or Off (0).