Introduction to the Julia Programming Language

NSF ACES

Jian Tao and Wes Brashear
20 July 2024
Introduction to Julia: Outline

Part I. A brief overview of Julia (~10 mins)

Part II. Getting Started with Julia on ACES (~10 mins)

Part III. Mathematical Operations in Julia (~10 mins)

Part IV. Variables, Data Types and Structures, Functions and Flow Control (~35 mins)

Part V. Plotting with Julia (~10 mins)
Part I: A brief overview of Julia

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 v1.10.2 as of Mar 31, 2024
- [https://julialang.org/](https://julialang.org/)
- customized for "greedy, unreasonable, demanding programmers".
- [Julia Computing](https://julia.computing) established in 2015 to provide commercial support.
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.
Where to Run Julia

- Juno IDE - developed for the Julia language (no longer under development)
- VSCode - extensions for Julia are actively being managed
- Jupyter Notebook
- Julia REPL
  - Run, Evaluate, Print, Loop
  - Interactive
  - Searchable history, tab-completion, keybindings, dedicated help and shell modes
Part II: Getting started with Julia on ACES
Accessing the HPRC ACES Portal
Accessing the HPRC ACES Portal

Select the Identity Provider appropriate for your account
Accessing the ACES Shell

![ACES Shell Access](image-url)
IMPORTANT POLICY INFORMATION

- Unauthorized use of HPRC resources is prohibited and subject to criminal prosecution.
- Use of HPRC resources in violation of United States export control laws and regulations is prohibited. Current HPRC staff members are US citizens and legal residents.
- Sharing HPRC account and password information is in violation of Texas State Law. Any shared accounts will be DISABLED.
- Authorized users must also adhere to ALL policies at:
  https://hprc.tamu.edu/policies/

ACES Status, June 28

Most H100s are available again. We plan to return the remaining 4 H100s to service early next week after a hardware component is replaced.

!! WARNING: THERE ARE ONLY NIGHTLY BACKUPS OF USER HOME DIRECTORIES. !!

Please restrict usage to 8 CORES across ALL login nodes. Users found in violation of this policy will be SUSPENDED.

To see these messages again, run the `motd` command.

Your current disk quotas are:

<table>
<thead>
<tr>
<th>Disk</th>
<th>Disk Usage</th>
<th>Limit</th>
<th>File Usage</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/u.eu100000</td>
<td>2.7G</td>
<td>10.0G</td>
<td>8593</td>
<td>10000</td>
</tr>
<tr>
<td>/scratch/user/u.eu100000</td>
<td>1.6T</td>
<td>5.0T</td>
<td>80960</td>
<td>25000</td>
</tr>
<tr>
<td>/scratch/group/p.sta22004.000</td>
<td>4K</td>
<td>1.0T</td>
<td>1</td>
<td>50000</td>
</tr>
<tr>
<td>/scratch/group/p.tra23003.000</td>
<td>4K</td>
<td>1.0T</td>
<td>1</td>
<td>50000</td>
</tr>
<tr>
<td>/scratch/group/p.tra220029.000</td>
<td>4K</td>
<td>1.0T</td>
<td>1</td>
<td>50000</td>
</tr>
</tbody>
</table>

Type 'showquota' to view these quotas again.

[u.eu100000@aces.login1 ~]$

Using Pre-installed Julia Modules

**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/1.8.5-linux-x86_64
- Julia/1.9.3-linux-x86_64
- Julia/1.10.0-musl-x86_64
- Julia/1.10.2-linux-x86_64

...

**Step 2. Load the module**

$ module load Julia/1.10.2-linux-x86_64

**Step 3. Start Julia REPL**

$ julia

You can also use the **web-based interface** to find software modules available on HPRC systems.
Installing Your Own Copy of Julia

**Step 1. Find the version to be installed at Download Julia**

Current stable release: v1.10.4 (June 4, 2024)

*You can install the latest Julia version (v1.10.4 June 4, 2024) directly by running this in your terminal:*

```
$curl -fsSL https://install.julialang.org | sh
```

<table>
<thead>
<tr>
<th>Platform</th>
<th>64-bit</th>
<th>32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows [help]</td>
<td>installer, portable</td>
<td>installer, portable</td>
</tr>
<tr>
<td>macOS x86 (Intel or Rosetta) [help]</td>
<td>.dmg, .tar.gz</td>
<td>.dmg, .tar.gz</td>
</tr>
<tr>
<td>macOS (Apple Silicon) [help]</td>
<td>.dmg, .tar.gz</td>
<td>.dmg, .tar.gz</td>
</tr>
</tbody>
</table>

**Step 2. Download & Unzip**

```
$ cd $SCRATCH
$ wget https://.../julia-1.10.4-linux-x86_64.tar.gz
$ tar -zxvf julia-1.10.4-linux-x86_64.tar.gz
```

**Step 3. Start Julia REPL**

```
$ module purge
$ cd $SCRATCH/julia-1.10.4/bin; ./julia
```

Documentation: https://docs.julialang.org
Type "?" for help, "?" for Pkg help.
Version 1.10.4 (2024-06-04)
Official https://julialang.org/ release
# 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>
 (@v1.9) pkg> add Plots UnicodePlots Plotly
Copying Course Examples

- Navigate to your personal scratch directory
  
  $ cd $SCRATCH

- Files for this course are located at

  /scratch/training/julia_examples.tgz

  Make a copy in your personal scratch directory

  $ cp /scratch/training/julia/julia_examples.tgz $SCRATCH/

- Extract the files

  $ tar -zxvf julia_examples.tgz

- Enter this directory (your local copy)

  $ cd julia_examples

  $ julia helloworld.jl
Julia - Quickstart

- The julia program starts the interactive REPL.
- You can switch 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.
- To get version information:
  
  ```julia
  julia> versioninfo()
  ```

- Special symbols can be typed with a backslash and <TAB>, but they might not show properly on the web-based terminal.
  
  ```julia
  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>
<tr>
<td>]</td>
<td>Enter Pkg REPL</td>
</tr>
<tr>
<td>Backspace or ^c</td>
<td>Quit Pkg REPL</td>
</tr>
</tbody>
</table>
Part III: Mathematical Operations in Julia
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Expression</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+x</td>
<td>unary plus</td>
<td>the identity operation</td>
</tr>
<tr>
<td>-x</td>
<td>unary minus</td>
<td>maps values to their additive inverses</td>
</tr>
<tr>
<td>x + y</td>
<td>binary plus</td>
<td>performs addition</td>
</tr>
<tr>
<td>x - y</td>
<td>binary minus</td>
<td>performs subtraction</td>
</tr>
<tr>
<td>x * y</td>
<td>times</td>
<td>performs multiplication</td>
</tr>
<tr>
<td>x / y</td>
<td>divide</td>
<td>performs division</td>
</tr>
<tr>
<td>x ÷ y</td>
<td>integer divide</td>
<td>x / y, truncated to an integer</td>
</tr>
<tr>
<td>x ^ y</td>
<td>power</td>
<td>raises x to the yth power</td>
</tr>
<tr>
<td>x % y</td>
<td>remainder</td>
<td>equivalent to rem(x,y)</td>
</tr>
</tbody>
</table>
More about Arithmetic Operators

- The **order of operations** follows the math rules.
- The **updating version** of the operators is formed by placing a "=" immediately after the operator. For instance, \( x+=3 \) is equivalent to \( x=x+3 \).
- **Unicode** could be defined as operator.
- A "dot" **operation** is automatically defined to perform the operation element-by-element on arrays in every binary operation.
- **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*\euler  #\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 $\neq$
<  less than
>  greater than
<=, ≤ less than or equal to $\leq$
>=, ≥ greater than or equal to $\geq$

* 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
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
```
Part IV: Variables, Data Types and Structures, Functions and Flow Control
Variables

Examples:

```
 julia> b = true; typeof(b)
 julia> x = "Hi"
 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
  
  \texttt{ julia> 4c = 12 }

- Names can include any combinations of letters, numbers, underscores, and exclamation symbol. Some unicode characters could be used as well
  
  \texttt{ julia> c_4 = 12; \delta = 2 }

- Maximum length for a variable name is not limited
- Julia is case sensitive. The variable name \texttt{A} is different than the variable name \texttt{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
julia> print("The value of x is:"); print(x)
27
julia> println("The value of x is:"); print(x)
27
```
Exercise

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

Now use Julia to perform the following set of calculations:

\[
\begin{align*}
(b+5.4)^{1/3} & \quad b^2 - 4b + 5a \\
a > b \land a > 1.0 & \quad a \neq b
\end{align*}
\]
Data Types

- Data types in Julia are polymorphic, they can be
  - dynamic: determined at runtime (e.g. Python, R)
  - static: defined explicitly (e.g. Java, C++)

*more information on declaring types here: https://docs.julialang.org/en/v1/manual/types/

- Data types include:
  - char
  - string
  - float
  - int
  - bool
Chars and Strings

**Char:** represent a single character
- Denoted with single quotations ‘ ‘

**String:** can represent an object of one or more characters
- Denoted with double quotations “ “

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

Strings can be easily manipulated with built-in functions:

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

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. One can do comparisons and a limited amount of arithmetic with **Char**.
4. 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"
```
Working with Strings

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")
```

Working with Strings

**Julia** comes with a collection of tools to handle strings.

```julia
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.

```julia
julia> occursin(r"^\s*(?:#|$)", "# a comment")
```
Numerical Data Types
# 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 **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

<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**.
Working with Floating Points

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?
Working with Floating Points

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.
Working with Floating Points

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 // operator, e.g., \texttt{9//27}
### Some Useful Math Functions

#### 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 -Inf</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>round x towards +Inf</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 -Inf</td>
</tr>
<tr>
<td>cld(x,y)</td>
<td>ceiling division; quotient rounded towards +Inf</td>
</tr>
<tr>
<td>rem(x,y)</td>
<td>remainder; satisfies x == div(x,y)*y + 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>

* The built-in math functions in Julia are implemented in C([openlibm](https://github.com/xorg/libm)).
Getting Help with Functions

- 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
  ```
Code Elements and Syntax

- Comments start with ‘#’
- Compound expressions
  - Can be created with “;” chains
    
    ```julia
    julia> z = (x = 1, y = 2, x + y)
    ```
  - or with blocks: start with “begin” and finish with “end”
    
    ```julia
    julia> z = begin
        x = 1
        y = 2
        x + y
    end
    ```
Data Structures

**Tuples:** ordered sequence of elements.

- Good for small fixed-length collections
- Immutable

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

**Arrays:** ordered collection of elements.

- In Julia, **arrays** are used for lists, vectors, tables, and matrices

  ```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> [a a]
  julia> [b;b]
  ```

- **Mutable**
Array and 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
```
Data Structures

**Sets:** mainly used to eliminate repeated numbers in a sequence/list and are also used to perform some standard set operations.

```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)
```
Data Structures

Dictionaries: mappings between keys and items stored in the dictionaries.

- To define a dictionary, use `Dict()`

```julia
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)))
```
Flow Control

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
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
```
Loop Control Statements - **for**

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

```julia
julia> for i in 0:1:10
         if i % 3 == 0
             continue
         end
         println(i)
     end

julia> for l in "julia"
         print(l, "-^-")
     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
julia> s = "test"
julia> try
       s[1] = "p"
    catch err
       println("caught an error: $err")
       println("continue with execution!"")
    end
```
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
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)
```

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

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

```julia
julia> func(a, b) = a * b
julia> func(1, 2)
julia> func.([1,2], 3)
julia> sin.(func.([1,2],[3,4]))
```
Part V: Plotting with Julia
UnicodePlots is simple and lightweight and it plots directly in your terminal (might not work with web-based shell).

```julia
julia> using Plots
julia> unicodeplots()
julia> plot(rand(5,5), linewidth=2, title="My Plot")
```
Plotly creates leading open source software for Web-based data visualization and analytical apps. Plotly Julia Library makes interactive, publication-quality graphs online (not working with web-based shell).

```julia
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 (not working with web-based shell).

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

- Fractals refer to geometric shapes containing detailed structures at arbitrarily small scales
- Fractals appear similar at various scales

Credit: [Fractal - Wikipedia](https://en.wikipedia.org/wiki/Fractal)
Benoit Mandelbrot Set

\[ z_{n+1} = z_n^2 + c \]

- \( z \) and \( c \) are complex numbers.
- Starting with \( z_0 = 0 \).
- Mandelbrot set is the set of values of \( c \) when \( z_n \) remains bounded for a relatively large \( n \).
Mandelbrot - Julia Version

```
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(" ") # in one line
        println()
    end
end
```

The first published picture of the Mandelbrot set, by Robert W. Brooks and Peter Matelski in 1978, reproduced with the code to the left.
Online Resources

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

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

Introducing Julia (Wikibooks.org)

MATLAB–Python–Julia cheatsheet
https://cheatsheets.quantecon.org/

The Fast Track to Julia
https://juliadocs.github.io/Julia-Cheat-Sheet/
Acknowledgments

● The slides are created based on the materials from Julia official website and the Wikibook Introducing Julia at wikibooks.org.

● Support from Texas A&M Engineering Experiment Station (TEES), Texas A&M Institute of Data Science (TAMIDS), and Texas A&M High Performance Research Computing (HPRC).

● Support from NSF OAC Award #2019129 - MRI: Acquisition of FASTER - Fostering Accelerated Sciences Transformation Education and Research

● Support from NSF OAC Award #2112356 - Category II: ACES - Accelerating Computing for Emerging Sciences
<table>
<thead>
<tr>
<th>Talk/Event</th>
<th>Date/Time</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial: Hands-on exercises on the Intel Data Center GPU Max 1100 (PVC-GPU) for AI/ML and Molecular Dynamics Workflows on the ACES Testbed</td>
<td>Mon, July 22, 2024 9:00 AM-12:30 PM ET</td>
<td>Room 553B</td>
</tr>
<tr>
<td>Seventh Workshop on Strategies for Enhancing HPC Education and Training (SEHET24)</td>
<td>Mon, July 22, 2024 9:00 AM-12:30 PM ET</td>
<td>Room 557</td>
</tr>
<tr>
<td>Workshop: Providing cutting-edge computing testbeds to the science and engineering community</td>
<td>Mon, July 22, 2024 1:30 PM-5:00 PM ET</td>
<td>Room 554A</td>
</tr>
<tr>
<td>Workshop: Engaging Secondary Students in Computing: K12 Outreach</td>
<td>Mon, July 22, 2024 1:30 PM-5:00 PM ET</td>
<td>Room 553A</td>
</tr>
<tr>
<td>Cultivating Cyberinfrastructure Careers through Student Engagement at Texas A&amp;M University High Performance Research Computing</td>
<td>Tue, July 23, 2024 11:00 AM-11:25 AM ET</td>
<td>Junior Ballroom</td>
</tr>
<tr>
<td>Insight Gained from Migrating a Machine Learning Model to Intelligence Processing Units</td>
<td>Tue, July 23, 2024 11:00 AM-11:25 AM ET</td>
<td>Room 551 A&amp;B</td>
</tr>
<tr>
<td>BOF 4: What's in it for me? How can we truly democratize the research computing and data community?</td>
<td>Tue, July 23, 2024 1:30 PM-2:30 PM ET</td>
<td>Room 551 A&amp;B</td>
</tr>
<tr>
<td>Talk/Event</td>
<td>Date/Time</td>
<td>Room</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>BRICCs: Building Pathways to Research Cyberinfrastructure at Under</td>
<td>Tue, July 23, 2024 3:25 PM-3:50 PM ET</td>
<td>Junior Ballroom</td>
</tr>
<tr>
<td>Resourced Institutions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Bandwidth Performance across Accelerators</td>
<td>Tue, July 23, 2024 3:25 PM-3:50 PM ET</td>
<td>Ballroom B</td>
</tr>
<tr>
<td>Container Adoption in Campus High Performance Computing</td>
<td>Wed, July 24, 2024 11:00 AM-11:25 AM ET</td>
<td>Ballroom B</td>
</tr>
<tr>
<td>Engaging Secondary Students in Computing and Cybersecurity</td>
<td>Wed, July 24, 2024 3:15 PM-3:30 PM ET</td>
<td>Room 557</td>
</tr>
<tr>
<td>Exploring the Viability of Composable Architectures to Overcome Memory</td>
<td>Wed, July 24, 2024 3:45 PM-4:00 PM ET</td>
<td>Room 553 A&amp;B</td>
</tr>
<tr>
<td>Limitations in High Performance Computing Workflows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance of Molecular Dynamics Acceleration Strategies on Composable</td>
<td>Wed, July 24, 2024 4:15 PM-4:30 PM ET</td>
<td>Room 551 A&amp;B</td>
</tr>
<tr>
<td>Cyberinfrastructure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOF 17: Fantastic ACCESS Cyberinfrastructure Resources and Where to Find</td>
<td>Wed, July 24, 2024 4:45 PM-5:45 PM ET</td>
<td>Room 553 A&amp;B</td>
</tr>
<tr>
<td>Them</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOF 18: Recipes to build successful cross-institutional collaborative</td>
<td>Wed, July 24, 2024 4:45 PM-5:45 PM ET</td>
<td>Junior Ballroom</td>
</tr>
<tr>
<td>computing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thank you

- We gratefully acknowledge support from National Science Foundation awards #2112356 (ACES), #2019129 (FASTER) and #19257614 (SWEETER)
- Please visit our talks and BoFs at PEARC24
- Helpdesk: help@hprc.tamu.edu