Introduction to Julia

Part I. Getting Started with FASTER (~20 mins)
Part II. Julia - What and Why? (~20 mins)
Part III. Julia as an Advanced Calculator (~30 mins)
Part IV. Basics of Julia (~60 mins)
Part V. Plotting with Julia (~10 mins)
Q&A and Break (10 mins)
Part I. Getting Started with FASTER

TAMU HPRC Short Course: [Getting Started with FASTER and ACES]
# FASTER Cluster

hprc.tamu.edu/wiki/FASTER:Intro

<table>
<thead>
<tr>
<th>Resources</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-core login nodes</td>
<td>4 (3 for TAMU, 1 for ACCESS)</td>
</tr>
<tr>
<td>64-core compute nodes (256GB RAM each)</td>
<td>180 (11,520 cores)</td>
</tr>
<tr>
<td>Composable GPUs</td>
<td>200 T4 16GB</td>
</tr>
<tr>
<td></td>
<td>40 A100 40GB</td>
</tr>
<tr>
<td></td>
<td>10 A10 24GB</td>
</tr>
<tr>
<td></td>
<td>4 A30 24GB</td>
</tr>
<tr>
<td></td>
<td>8 A40 48GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Mellanox HDR100 InfiniBand</td>
</tr>
<tr>
<td></td>
<td>(MPI and storage)</td>
</tr>
<tr>
<td></td>
<td>Liqid PCIe Gen4 (GPU composability)</td>
</tr>
<tr>
<td>Global Disk</td>
<td>5PB DDN Lustre appliances</td>
</tr>
</tbody>
</table>

FASTER (Fostering Accelerated Sciences Transformation Education and Research) is a 180-node Intel cluster from Dell featuring the Intel Ice Lake processor.
Composability at Hardware Level

**Traditional Server Configuration**

- Server pool
- Accelerator pool (GPUs, FPGA, etc.)
- Storage pool (SSDs)

**Composable Resources**

- Liquid Fabric
  - Composable Server Configuration
  - server
  - Accelerator
  - SSDs

**Composable Server Configuration**

[Image of server pool, accelerator pool, and storage pool]

[Image of composable resources and configuration]

[Link to webpage: hprc.tamu.edu/wiki/FASTER:Intro]
### ACES - Accelerating Computing for Emerging Sciences (Phase I)

<table>
<thead>
<tr>
<th>Component</th>
<th>Quantity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphcore IPU</td>
<td>16</td>
<td>16 Colossus GC200 IPUs and dual AMD Rome CPU server on a 100 GbE RoCE fabric</td>
</tr>
<tr>
<td>Intel FPGA PAC D5005</td>
<td>2</td>
<td>FPGA SOC with Intel Stratix 10 SX FPGAs, 64 bit quad-core Arm Cortex-A53 processors, and 32GB DDR4</td>
</tr>
<tr>
<td>Intel Optane SSDs</td>
<td>8</td>
<td>3 TB of Intel Optane SSDs addressable as memory using MemVerge Memory Machine.</td>
</tr>
</tbody>
</table>

ACES Phase I components are available through [FASTER](#)
Accessing the HPRC Portal

- HPRC webpage: [hprc.tamu.edu](http://hprc.tamu.edu), Portal dropdown menu
Accessing FASTER via the HPRC Portal (TAMU)

Log-in using your TAMU NetID credentials.
Accessing FASTER via the HPRC Portal (ACCESS)

Log-in using your ACCESS credentials.

Select the Identity Provider appropriate for your account.
Login HPRC Portal - FASTER/FASTER(ACCESS)
FASTER Shell Access - Portal

OnDemand provides an integrated, single access point for all of your HPC resources.

Message of the Day

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 State Law. Any shared accounts will be DISABLED.
- Authorized users must also adhere to ALL policies at: https://hprc.tamu.edu/policies

!! WARNING: THERE ARE ONLY NIGHTLY BACKUPS OF USER HOME DIRECTORIES. !!
FASTER Shell Access - Shell

This computer system and the data herein are available only for authorized purposes by authorized users. Use for any other purpose is prohibited and may result in disciplinary actions or criminal prosecution against the user. Usage may be subject to security testing and monitoring. There is no expectation of privacy on this system except as otherwise provided by applicable privacy laws. Refer to University SAP 29.01.03.M0.02 Acceptable Use for more information.

Password: 

UPDATE (12:11a 02/20/2023): There was another storage incident between 11:10-11:50p. We are still investigating this new incident since the indicators observed so far were not related to the previous user's jobs.

UPDATE (10:16p 02/19/2023): We may have isolated and removed the batch jobs that were impacting the Grace and FASTER shared storage. We are continuing to monitor the storage for any further issues.

Original announcement (7:13p 02/19/2023): Both the FASTER and Grace clusters are currently having issues with their shared storage since about 5:15p February 19th. The root cause and recovery options are under investigation.

[02feb2023] FASTER hardware update: DS085 FPGAs are currently unavailable.

To see these messages again, run the motd command.

Your current disk quotas are:

<table>
<thead>
<tr>
<th>Disk</th>
<th>Usage</th>
<th>Limit</th>
<th>File Usage</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>/home/jtao</td>
<td>4K</td>
<td>10.0G</td>
<td>5</td>
<td>10000</td>
</tr>
<tr>
<td>/scratch/user/jtao</td>
<td>21.5G</td>
<td>1.0T</td>
<td>156632</td>
<td>250000</td>
</tr>
<tr>
<td>/scratch/group/hprc</td>
<td>4.0T</td>
<td>31.0T</td>
<td>628476</td>
<td>1000000</td>
</tr>
</tbody>
</table>

* Quota increase for /scratch/group/hprc will expire on Dec 31, 2026

Type `showquota` to view these quotas again.

[jtao@fastr2 ~]$
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/1.6.5-linux-x86_64
  Julia/1.7.0-linux-x86_64
  Julia/1.7.1-linux-x86_64
  Julia/1.7.2-linux-x86_64
  Julia/1.8.0-linux-x86_64
...
```

You can also use the [web-based interface](https://software.tamu.edu) to find software modules available on HPRC systems. [SW:Julia - TAMU HPRC](https://software.tamu.edu)

**Step 2. Load the module**

```
$ module load Julia/1.8.0-linux-x86_64
```

**Step 3. Start Julia REPL**

```
$ julia
[...]  
```

Documentation: [https://docs.julialang.org](https://docs.julialang.org)
Type "?" for help, ":?" for Pkg help.
Version 1.8.0 (2022-08-17)
Official [https://julialang.org](https://julialang.org)/ release
Using Your Own Julia Installation

**Step 1. Find the version to be installed**

Current stable release: v1.8.5 (January 8, 2023)

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Size (GB)</th>
<th>Format</th>
<th>Architecture</th>
<th>32-bit ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows [help]</td>
<td>3</td>
<td>64-bit (installer), 64-bit (portable)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>macOS x86 (Intel or Rosetta) [help]</td>
<td>4.2</td>
<td>64-bit (.dmg), 64-bit (.tar.gz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>macOS ARM (M-series Processor) [help]</td>
<td>4.2</td>
<td>64-bit (.dmg), 64-bit (.tar.gz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic Linux on x86 [help]</td>
<td>4.2</td>
<td>64-bit (glibc) (GPG), 64-bit (musl)[1] (GPG)</td>
<td>32-bit (GPG)</td>
<td></td>
</tr>
<tr>
<td>Generic Linux on ARM [help]</td>
<td>4.2</td>
<td>64-bit (AArch64) (GPG)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic Linux on PowerPC [help]</td>
<td>4.2</td>
<td>64-bit (little endian) (GPG)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic FreeBSD on x86 [help]</td>
<td>4.2</td>
<td>64-bit (GPG)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>4.2</td>
<td>Tarball (GPG)</td>
<td></td>
<td>Tarball</td>
</tr>
</tbody>
</table>

**Step 2. Download & Unzip**

$ cd $SCRATCH
$ wget https://.../julia-1.8.5-linux-x86_64.tar.gz
$ tar -zxvf julia-1.8.5-linux-x86_64.tar.gz

**Step 3. Start Julia REPL**

$ module purge
$ cd $SCRATCH/julia-1.8.5/bin; ./julia

[jtac@faster2 bin]$ ./julia

Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.8.5 (2023-01-08)
Official https://julialang.org/ release
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>

(@v1.8) pkg> add Plots UnicodePlots Plotly
Commands to Copy 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_examples.tgz $SCRATCH/
  ```

- Extract the files
  
  ```
  $ tar -zxvf julia_examples.tgz
  ```

- Enter this directory (your local copy)
  
  ```
  $ cd julia_examples
  $ julia helloworld.jl
  ```
Load Julia Module, Compile, and Run
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

<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 II.
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 v1.8.5 as of Mar 14, 2023
- [https://julialang.org/](https://julialang.org/)
- customized for "greedy, unreasonable, demanding programmers".
- [Julia Computing](https://julialang.org/) established in 2015 to provide commercial support.
Image Credit: [Julia Micro-Benchmarks](#)
RedMonk Q322 Programming Language Rankings

Image Credit: RedMonk (The RedMonk Programming Language Rankings: June 2022 – tecosystems)
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.

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

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

Exploring the Lorenz System

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

\[
\begin{align*}
\frac{dx}{dt} &= \alpha (y - x) \\
\frac{dy}{dt} &= \beta x - y - xz \\
\frac{dz}{dt} &= xy - \gamma z
\end{align*}
\]

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

In [7]: interact(Lorenz, x0=x0, y0=y0, z0=z0, \\
\quad \alpha=\text{slider}(0.1, 10), \\
\quad \beta=\text{slider}(0.1, 10), \\
\quad \gamma=\text{slider}(0.1, 10))

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

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 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
<=,\le less than or equal to \le <TAB>
>=,\ge 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 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
```
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)
```
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> 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:

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

1. Comments start with '#'

2. 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)
```
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 `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, 6.1035e-05 to 65504</td>
</tr>
<tr>
<td>Float32</td>
<td>single</td>
<td>32</td>
<td>-3.402823E38 to -1.401298E-45, 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, 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 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.
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.
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>round(x)</code></td>
<td>round x to the nearest integer</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>round x towards -Inf</td>
</tr>
<tr>
<td><code>ceil(x)</code></td>
<td>round x towards +Inf</td>
</tr>
<tr>
<td><code>trunc(x)</code></td>
<td>round x towards zero</td>
</tr>
<tr>
<td><code>div(x,y)</code></td>
<td>truncated division; quotient rounded towards zero</td>
</tr>
<tr>
<td><code>fld(x,y)</code></td>
<td>floored division; quotient rounded towards -Inf</td>
</tr>
<tr>
<td><code>cld(x,y)</code></td>
<td>ceiling division; quotient rounded towards +Inf</td>
</tr>
<tr>
<td><code>rem(x,y)</code></td>
<td>remainder; satisfies x == div(x,y)*y + rem(x,y); sign matches x</td>
</tr>
<tr>
<td><code>gcd(x,y...)</code></td>
<td>greatest positive common divisor of x, y,...</td>
</tr>
<tr>
<td><code>lcm(x,y...)</code></td>
<td>least positive common multiple of x, y,...</td>
</tr>
</tbody>
</table>

## Sign and absolute value functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs(x)</code></td>
<td>a positive value with the magnitude of x</td>
</tr>
<tr>
<td><code>abs2(x)</code></td>
<td>the squared magnitude of x</td>
</tr>
<tr>
<td><code>sign(x)</code></td>
<td>indicates the sign of x, returning -1, 0, or +1</td>
</tr>
<tr>
<td><code>signbit(x)</code></td>
<td>indicates whether the sign bit is on (true) or off (false)</td>
</tr>
<tr>
<td><code>copysign(x,y)</code></td>
<td>a value with the magnitude of x and the sign of y</td>
</tr>
<tr>
<td><code>flipsign(x,y)</code></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://openlibm.sgi.com)).
Chars and Strings

Julia has a first-class type representing a single character, called \texttt{Char}. Single quotes are `&` double quotes are used different in Julia.

\begin{verbatim}
julia> a = 'H'  \# a is a character object
julia> b = "H"  \# a is a string with length 1
\end{verbatim}

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

\begin{verbatim}
julia> c = string('s') * string('d')
julia> length(c); d = c^10*'4'; split(d,"s")
\end{verbatim}
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.
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> occursin(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. Basics of Julia

1. Functions - Building Blocks of Julia

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

```julia
julia> Σ(x,y) = x + y, x
```

Operators are functions

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

```julia
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
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 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 \textbf{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 IV. Basics of Julia

2. 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 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]             # 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** 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
julia> m=Dict("Oct"=>"October",
               "Nov"=>"November",
               "Dec"=>"December")
```

```julia
julia> m["Oct"]
```

```julia
julia> get(m, "Jan", "N/A")
```

```julia
julia> haskey(m, "Jan")
```

```julia
julia> m["Jan"]="January"
```

```julia
julia> delete!(m, "Jan")
```

```julia
julia> keys(m)
```

```julia
julia> values(m)
```

```julia
julia> map(uppercase, collect(keys(m)))
```
Part IV. Basics of Julia

3. 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
julia> [n for n in 1:10]
```

Array enumeration:

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

Generator expressions:

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

Nested loop:

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

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

---

[Image] GKS QtTerm

**My Plot**

- y1
- y2
- y3
- y4
- y5
Fractal

- Fractal refers to geometric shapes containing detailed structure at arbitrarily small scales.
- Fractals appear similar at various scales.

Credit: Fractal - Wikipedia
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 \( n \).
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
    end
    println()
end

Mandelbrot - Julia Version

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
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

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

Julia manages its packages with Pkg

```julia
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 American Standard Code for Information Interchange.
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.