

Programming in Julia

Nathan Warner



**Northern Illinois
University**

Computer Science
Northern Illinois University
United States

Contents

1	Creating and running Julia scripts, and outputting	8
1.1	Outputting data	8
1.1.1	println	8
1.1.2	Print	8
1.1.3	Formatted output: @printf from the Printf module	8
1.1.4	Error and debugging output	9
1.1.5	Display	9
2	Preliminary	10
2.1	Typing style	10
2.1.1	Dynamic Typing	10
2.1.2	Optional Type Annotations	10
2.2	Just in time	10
2.3	Language style	11
2.3.1	Block structure	11
2.4	Paradigms	11
2.5	Memory	12
2.5.1	The garbage collector	12
2.6	Importing	13
2.6.1	Using	13
2.6.2	Import	13
2.6.3	Selective imports with using	13
2.6.4	Selective imports with import	14
2.6.5	Extending functions	14
3	The basics	15
3.1	Data types	15

3.1.1	Numerical Types	15
3.1.2	Boolean Type	15
3.1.3	Characters and Strings	15
3.1.4	Abstract Data Types	16
3.1.5	Composite Types	16
3.1.6	Collection Types	16
3.1.7	Nothing and Missing:	16
3.1.8	User-defined Types	16
3.1.9	typeof()	16
3.2	Type conversions	17
3.2.1	Using Constructors	17
3.2.2	Using convert Function	17
3.2.3	Parsing Strings	17
3.2.4	Automatic Conversion	18
3.3	Type coercion / Type promotion	18
3.4	Variables	19
3.4.1	Type annotations	19
3.4.2	Constant Declaration (Immutable Binding / mutable content)	19
3.4.3	Local / global keywords	20
3.5	Operators	21
3.6	Operator precedence	23
3.7	Ranges	23
3.8	Concatenation and Splitting Operators	23
3.9	Broadcasting	24
3.10	Comments	25
3.11	The in operator	25
3.12	Symbols	25
4	Functions	26
4.1	Returning	26
4.2	Short form (single-line) functions	27
4.2.1	Lambdas	27
4.3	Multiple Dispatch	27

4.4	Default argument	28
4.5	Variadic functions and kwargs	28
4.5.1	Variadic functions	28
4.5.2	Kwarg functions	28
4.5.3	Variadic kwargs	29
5	Control flow	30
5.1	The let block	30
5.2	Conditional blocks (if-elseif-else)	30
5.3	For loops	31
5.4	While loops	31
5.5	Short-Circuit Control	31
5.6	Begin blocks	31
5.7	Anonymous Blocks	32
5.8	Functions as Control Blocks (do blocks)	32
6	Strings	33
6.1	Characters	33
6.2	String Operations	33
6.3	Converting to strings	34
6.4	Indexing and Slicing	34
6.5	Into to string Functions	34
6.6	Unicode Support	35
6.7	Mutability	35
6.8	Multiline Strings	35
6.9	Raw Strings	35
6.10	String Types	35
6.11	String functions	36
6.11.1	Basic String Information	36
6.11.2	String Manipulation	36
6.11.3	Searching and Matching	36
6.11.4	Splitting and Joining	36
6.11.5	String Transformation	37

6.11.6	Character Inspection	37
7	Splatting and slurping	38
7.1	Splatting	38
7.2	Slurping	38
8	Arrays	39
8.1	Creating Arrays	39
8.2	Accessing elements	39
8.3	Modifying arrays	40
8.4	Properties of arrays	40
8.5	Broadcasting	40
8.6	Key features	40
8.7	Traversing an array	40
8.8	Array slicing	41
8.9	Arrays and strings	41
8.10	Objects and values	41
8.11	Aliasing	42
8.12	Array functions	42
8.12.1	Array Creation	43
8.12.2	Array Properties	43
8.12.3	Array Indexing and Slicing	43
8.12.4	Array Modification	44
8.12.5	Combining and Splitting Arrays	44
8.12.6	Mapping and Broadcasting	44
8.12.7	Statistical Functions	44
8.12.8	Array Manipulation	45
8.12.9	Matrix-Specific Functions	45
8.12.10	Linear Algebra Functions	45
9	Other containers	46
9.1	Tuples	46
9.2	Named Tuples	46
9.3	Dictionaries	46

9.4	Sets	46
9.5	Ranges	46
9.6	Sparse Arrays	47
9.7	Deque (from the DataStructures package)	47
9.8	Stores elements with associated priorities.	47
10	Dictionaries	48
10.0.1	Getting the length	48
10.1	Getting the keys	48
10.1.1	Checking if a key exists	48
10.2	Getting the values	49
10.3	Looping through dictionaries	49
11	Pairs	50
11.1	Key features	50
11.2	Accessing elements	50
12	Sets	51
12.1	Creating sets	51
12.2	Basic Operations	51
13	Comprehensions	53
13.1	Set Comprehensions	53
13.2	Dictionary Comprehensions	53
14	Iterators	55
15	Structs	56
15.1	Key features	56
15.2	Creating structs	56
15.3	Mutable Structs	57
15.4	Structs Without Field Type Annotations	57
15.5	Default Values	57
15.6	Parametric Structs	57
15.7	Abstract types and Subtypes	58

15.7.1	Abstract types	58
15.7.2	Concrete types	58
15.8	Usage of Abstract Types	59
15.9	Abstract Types and Parametric Types	60
15.10	Constructors	60
16	Memory	62
16.1	Stack Allocation	62
16.2	Heap Allocation	62
16.3	Garbage collection	62
16.4	Memory Views and Sharing	62
16.5	Static arrays	63
16.6	Advanced: Memory Buffers and Pointers	63
16.7	References	63
16.7.1	Immutable types	63
16.7.2	Mutable Types	63
16.7.3	Copying Mutable Objects	64
16.7.4	References in Functions	64
16.7.5	Ref for Explicit References	65
17	Working with files	66
17.1	The basics	66
17.2	Reading Files	66
17.3	Writing Files	66
17.4	Checking File Properties	67
17.5	Directory Operations	67
17.6	Example: Copying a File	68
18	Error handling	69
18.1	Throwing errors	69
18.2	Handling Errors	69
18.3	Finally Block	70
18.4	Custom Errors	70

19	Regular expressions	71
19.1	Basic Syntax	71
19.2	Key Functions for Regex in Julia	71
19.3	Flags	72
19.4	RegexMatch Object	72
20	Math	73
20.1	Plots.jl	73
20.1.1	Installation	73
20.1.2	Create and save plots	73
20.1.3	Titles and Labels	74
20.1.4	Line and Marker Styles	74
20.1.5	Legends	75
20.1.6	Plot Types	75
20.1.7	Create multiple subplots in a grid:	76
21	Packages	77
21.1	Installing a Package	77
21.2	Updating Installed Packages	77
21.3	Removing a Package:	77
21.4	Listing Installed Packages	77
21.5	To use a package in your project	77
21.6	Build or rebuild a package	77

Creating and running Julia scripts, and outputting

First, make sure you have Julia (juliaup) downloaded on your machine. Create a .jl file and run it with

```
1 julia script.jl
```

Here is a simple Hello World! script written in julia

```
0 # helloworld.jl
1 println("Hello World!")
```

1.1 Outputting data

1.1.1 println

Prints the value followed by a newline. Suitable for general-purpose printing

```
0 println("Hello world")
```

1.1.2 Print

Similar to `println`, but does not append a newline

1.1.3 Formatted output: `@printf` from the `Printf` module

Prints the value followed by a newline. Suitable for general-purpose printing

```
0 using Printf
1 @printf("Pi to 3 decimal places: %.3f\n", pi) # Output: Pi to 3
  ↪ decimal places: 3.142
```

1.1.4 Error and debugging output

- **@warn:** Logs a warning message.

```
0 @warn "This is a warning message."
```

- **@info:** Logs an informational message.

```
0 @info "This is an informational message."
```

- **@error:** Logs an error message.

```
0 @error "This is an error message."
```

- **@show:** Prints the expression and its value. Useful for debugging.

```
0 x = 42
1 @show x # Output: x = 42
```

1.1.5 Display

Displays a value using a richer representation (e.g., for plots or tables in Jupyter).

```
0 display("Hello, World!") # Output: "Hello, World!"
```

Preliminary

2.1 Typing style

Julia has a dynamic and strong typing system with optional type annotations.

2.1.1 Dynamic Typing

Variables don't need explicit type declarations, but their types are checked at runtime.

```
0  x = 42          # Type inferred as Int
1  y = 3.14        # Type inferred as Float64
```

2.1.2 Optional Type Annotations

Types can be explicitly specified for variables, function arguments, and return values to enforce constraints or improve readability.

```
0  function add(a::Int, b::Int)::Int
1      return a + b
2  end
```

2.2 Just in time

Julia is a Just-In-Time (JIT) compiled language. It uses the LLVM compiler framework to generate efficient machine code at runtime.

Julia infers types of variables and function arguments dynamically when a function is called.

For each unique combination of input types, Julia generates specialized machine code. This allows functions to be optimized for the specific types of data they operate on.

The first time a function is called with a specific set of argument types, Julia compiles it to machine code. Subsequent calls with the same types execute the precompiled code, leading to improved performance.

The JIT compilation provides the interactivity of an interpreted language (like Python) with the performance of a compiled language (like C or Fortran).

However, the initial compilation can introduce a "time-to-first-call" latency, which is a common trade-off in JIT-compiled languages

2.3 Language style

Julia's language style is explicit, concise, and block-delimited, resembling languages like Python and MATLAB

2.3.1 Block structure

Block Structure: Julia uses explicit keywords to start and end code blocks, ensuring clarity and readability.

```
0  function add(a, b) # Function block starts with `function`  
1      return a + b  
2  end               # Block ends with `end`
```

Note: Julia supports optional semicolons (;) for separating statements on the same line.

2.4 Paradigms

Julia is a multi-paradigm programming language that supports a variety of programming styles, allowing flexibility and adaptability for different use cases. The key paradigms Julia offers include

1. **Procedural Programming:** Follows a structured approach with sequences of instructions (procedures or functions).

Uses variables, loops, and conditionals to control the flow of execution.

2. **Object-Oriented Programming (OOP):** While Julia doesn't have traditional classes and inheritance, it uses structures (struct) and multiple dispatch to model OOP-like behavior.

Supports encapsulation and polymorphism using types and methods.

3. **Functional Programming:** Treats functions as first-class citizens (can be passed as arguments, returned as results)

Supports immutability, higher-order functions, and function composition.

4. **Meta-programming:** Julia has powerful capabilities for generating and transforming code at runtime using macros and the Expr type.

Enables code introspection and metaprogramming.

5. **Dynamic Programming:** Julia supports dynamic typing and runtime type flexibility, making it suitable for exploratory programming.

6. **Concurrent and Parallel Programming:** Julia provides built-in support for concurrent and parallel computing with features like coroutines (@async), threads, and distributed computing

7. **Scientific and Numerical Programming:** Julia's syntax and performance are well-suited for numerical computation, similar to MATLAB or NumPy.

Provides rich libraries for linear algebra, optimization, and differential equations.

8. **Multiple Dispatch (Core Paradigm of Julia):** At its core, Julia leverages multiple dispatch, allowing method selection based on the types of all function arguments, not just the first one (as in single dispatch).

This provides flexibility and is fundamental to Julia’s design.

9. **Array-Oriented Programming:** Julia is designed for efficient array and matrix operations, similar to MATLAB or R.

Supports broadcasting and element-wise operations with the `.` operator.

2.5 Memory

In Julia, memory management is automatic, and it uses references under the hood for many operations. While Julia doesn’t expose raw pointers in the same way as C or C++, it provides mechanisms to work with references and low-level memory when needed.

Julia has automatic garbage collection to manage memory, reclaiming unused objects when they are no longer referenced.

Users generally don’t need to manually free memory, but they can force garbage collection using `GC.gc()` if necessary.

2.5.1 The garbage collector

A garbage collector is a system for automatic memory management in programming languages. Its purpose is to reclaim memory occupied by objects no longer accessible or needed, thus preventing memory leaks and reducing the need for manual memory management by the programmer.

The garbage collector periodically identifies and frees unused memory by determining which objects are no longer reachable from the program. The process typically involves the following steps:

When objects or variables are created, memory is allocated from the heap (a memory area reserved for dynamic allocation). The garbage collector manages this heap and tracks references to objects.

The GC determines which objects are “reachable” and should not be collected. Reachable objects are those that can be accessed directly or indirectly by the program.

The starting points for determining reachability, such as

- Global variables
- Local variables in the current call stack
- CPU registers

Unreachable objects are those that:

- Are not referenced by any active part of the program.
- Cannot be accessed directly or indirectly from the “roots.”

Once the GC identifies unreachable objects, it frees the memory they occupy, making it available for new allocations.

Garbage collectors use a technique called "*reference counting*". Reference counting tracks the number of references to each object. When an object's reference count drops to zero, it is immediately deallocated.

A limitation of garbage collectors are *circular references*. A circular reference occurs when two or more objects reference each other in a way that creates a loop. This can prevent proper memory deallocation in systems that rely solely on reference counting for garbage collection.

2.6 Importing

In Julia, imports are managed using the `using` and `import` keywords to access modules and their functionality. Modules are collections of functions, types, and constants that help organize code into namespaces.

2.6.1 Using

Makes all exported symbols of a module available. Symbols (functions, types, etc.) are accessed directly without the module prefix.

```
0 using Random # Load the Random module
1
2 x = rand(1:10, 5) # Use `rand` directly (exported by Random)
```

2.6.2 Import

Imports the module but does not bring its symbols into the current namespace.

You must qualify all functions/types with the module name unless explicitly brought into scope.

2.6.3 Selective imports with using

Allows you to bring only specific exported symbols into the current namespace.

```
0 using Random: rand # Import only `rand`
1
2 x = rand(1:10, 5) # Use `rand` directly
```

Makes the specific symbol (`rand` in this case) from the `Random` module accessible in the current scope. Allows you to use `rand` directly in your code without the `Random.` prefix.

However, you cannot extend (add methods to) the `rand` function using this approach.

2.6.4 Selective imports with import

Allows you to import specific symbols, even if they are not exported by the module.

```
0  import Random: rand, shuffle! # Import `rand` and `shuffle!`  
1  
2  x = rand(1:10, 5)
```

Imports the specific symbol (`rand`) from the `Random` module into the current scope and allows extending it.

`rand` is directly accessible in your code.

You can extend the `rand` function by adding new methods.

2.6.5 Extending functions

If you want to extend a function from a module (e.g., add a new method), you must use `import`.

```
0  import Base: +  
1  
2  # Extend the `+` operator for a custom type  
3  struct MyType  
4    x::Int  
5  end  
6  
7  +(a::MyType, b::MyType) = MyType(a.x + b.x)
```

The basics

3.1 Data types

Julia has the following data types

3.1.1 Numerical Types

- **Integers:**
 - `Int8`, `Int16`, `Int32`, `Int64`, `Int128`: Signed integers with various bit sizes.
 - `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`: Unsigned integers.
 - `Int`: Default signed integer type (dependent on the platform, typically `Int64` or `Int32`).
- **Floating-point numbers:** `Float16`, `Float32`, `Float64`: IEEE 754 floating-point numbers.
- **Big numbers:**
 - `BigInt`: Arbitrary precision integers.
 - `BigFloat`: Arbitrary precision floating-point numbers.
- **Complex numbers:** `Complex{T}`: Complex numbers with real and imaginary parts of type `T`.
- **Rational numbers:** `Rational{T}`: Fractions represented as numerator//denominator.

In Julia, the default type for a literal integer like 15 is `Int` (platform-dependent, typically `Int64` on 64-bit systems). However, you can explicitly create integers of specific types (`Int8`, `Int16`, etc.) using constructors. For example,

```
o x = Int8(15) # Creates an Int8 with value 15
```

3.1.2 Boolean Type

- **Bool**: true or false

3.1.3 Characters and Strings

- **Char**: Single Unicode character (e.g., `'a'`).
- **String**: A sequence of characters (e.g., `"Hello, world!"`).

3.1.4 Abstract Data Types

- **Number:** Abstract type for all numbers.
- **Real:** Abstract type for real numbers (Int, Float64, etc.).
- **AbstractString:** Abstract type for string-like objects.

3.1.5 Composite Types

- **Tuples:** Fixed-size collections of values, e.g., (1, "hello", true).
- **NamedTuples:** Tuples with named fields, e.g., (a=1, b=2).

3.1.6 Collection Types

- **Arrays:**
 - **1D arrays (vectors):** Vector{T} (e.g., [1, 2, 3]).
 - **2D arrays (matrices):** Matrix{T} (e.g., [1 2; 3 4]).
 - **Higher-dimensional arrays:** Array{T, N}.
- **Ranges:**
 - **1:10:** A range from 1 to 10.
 - **1:2:10:** A range with a step of 2.
- **Dictionaries:**
 - **Dict{K, V}:** A collection of key-value pairs, e.g., Dict{"a" => 1, "b" => 2}.
- **Sets:**
 - **Set{T}:** An unordered collection of unique elements, e.g., Set([1, 2, 3])

3.1.7 Nothing and Missing:

- **Nothing:** Represents the absence of a value (similar to null in other languages).
- **Missing:** Represents missing data (useful in data analysis).

3.1.8 User-defined Types

- **struct:** Immutable composite types.
- **mutable struct:** Mutable composite types.

3.1.9 typeof()

We can use the `typeof()` function to retrieve the type of a variable

```
0 x = 10
1 println(typeof(x))
```

3.2 Type conversions

3.2.1 Using Constructors

Julia uses constructors to convert a value to a specific type. This is the most common way to perform type casting.

```
0 # Convert to Integer Types
1 x = Int8(42)           # Converts 42 to an Int8
2 y = UInt16(300)        # Converts 300 to a UInt16
```

3.2.2 Using convert Function

The convert function explicitly converts a value to the desired type.

```
0 convert{Type}(value)
```

For example

```
0 # Convert to Int
1 x = convert{Int}(42.5)      # Converts 42.5 to 42 (truncates
   ↳ the decimal)
2
3 # Convert to Float64
4 y = convert{Float64}(42)    # Converts 42 to 42.0
5
6 # Convert to String
7 s = convert{String}(123)    # Converts 123 to "123"
```

3.2.3 Parsing Strings

To convert a string to a numerical type, use the parse function.

```
0 parse{Type}(string)
```

For example,

```
0 # Parse to Integer
1 x = parse{Int}("123")      # Converts "123" to 123
2
3 # Parse to Float64
4 y = parse{Float64}("3.14") # Converts "3.14" to 3.14
```

3.2.4 Automatic Conversion

Some operations perform automatic type promotion or conversion.

```
0  a = 3          # Int
1  b = 2.5        # Float64
2  c = a + b      # Automatically promotes `a` to Float64
3  println(c)     # Output: 5.5
4  println(typeof(c)) # Output: Float64
```

3.3 Type coercion / Type promotion

Type coercion in Julia refers to the process of converting a value from one type to another. This can happen explicitly when you manually convert types or implicitly when Julia promotes values to a common type to ensure compatibility in operations.

Explicit type coercion is performed using the `convert` function or type constructors. These methods create a new value of the desired type from an existing value.

Implicit type coercion occurs automatically when Julia promotes values to a common type during operations. This is part of Julia's type promotion system.

When two values of different types are used in an operation, Julia promotes them to a common type

```
0  x = 3          # Int
1  y = 3.5        # Float64
2
3  z = x + y      # `x` is promoted to Float64
4  println(typeof(z)) # Output: Float64
```

Julia's promotion rules are designed for precision and performance:

- Integers are promoted to floats when combined with floating-point numbers.
- Smaller numeric types (e.g., `Int8`) are promoted to larger types (e.g., `Int` or `Float64`).
- Complex numbers are preserved when combined with real numbers.

The `promote` function explicitly promotes multiple values to a common type without performing operations

```
0  a, b = promote(2, 3.5)
1  println(typeof(a)) # Float64
2  println(typeof(b)) # Float64
```

We can easily convert numeric types to strings, but errors will occur when you try to do the opposite

```

0  x = 1
1  x = string(x)
2  println(typeof(x)) # string
3
4  # ERROR!
5  y = "1"
6  y = Int(y) # Cannot do

```

3.4 Variables

In Julia, variables are declared simply by assigning a value to a name. Julia is dynamically typed, so variables don't need an explicit type declaration, but you can optionally annotate types.

```

0  x = 10
1  name = "Julia"

```

3.4.1 Type annotations

You can explicitly specify the type of a variable using a `::` type annotation:

```

0  y::Int = 42    # y must always hold an Int
1  z::Float64 = 3.14

```

If the value assigned to the variable doesn't match the specified type, Julia throws a `TypeError`

3.4.2 Constant Declaration (Immutable Binding / mutable content)

Use the `const` keyword to declare constants. The type of the constant is inferred from its initial value, and while the value itself can be mutable, the binding cannot be changed

```

0  const pi = 3.14159

```

(Immutable binding): "the binding cannot be changed" means that the name of the constant is permanently bound to the initial object it was assigned to, and you cannot reassign the constant to a new value or object. However, if the value itself is a mutable object, you can modify the content of that object.

```

0  const MY_CONSTANT = 42
1
2  MY_CONSTANT = 100 # ERROR: invalid redefinition of constant
   ↳ MY_CONSTANT

```

(**Mutable content**): If the constant is a mutable object (e.g., an array or dictionary), the content of the object can still be modified, even though the binding to the name is fixed

```
0  const MY_ARRAY = [1, 2, 3]
1
2  # Modify the content of the array (allowed)
3  push!(MY_ARRAY, 4)
4  println(MY_ARRAY) # Output: [1, 2, 3, 4]
5
6  # Reassign the constant to a new object (not allowed)
7  MY_ARRAY = [5, 6, 7] # ERROR: invalid redefinition of constant
   ↳ MY_ARRAY
```

3.4.3 Local / global keywords

The local and global keywords control the scope of variables and determine whether a variable exists within a local scope (e.g., inside a function or loop) or the global scope (outside all functions, at the top level of a module or script).

The global scope refers to variables that are accessible throughout the entire module or script. Variables in the global scope are defined at the top level and are not limited to specific blocks or functions.

By default, variables assigned outside functions or blocks are global.

You can explicitly declare a variable as global inside a function if you want to modify a variable from the global scope.

```
0  x = 10 # Global variable
1
2  function modify_global()
3  global x # Declare x as global to modify it
4  x += 5
5  end
6
7  modify_global()
8  println(x) # Output: 15
```

Variables in the global scope can be read within functions without using the global keyword. To modify a global variable inside a function, you must explicitly declare it as global.

By default, variables declared inside loops are local to the loop. To modify a global variable inside a loop, you must use global.

```
0  x = 0 # Global variable
1
2  for i = 1:5
3      global x += i # Explicitly declare x as global to modify it
4  end
5
6  println(x) # Output: 15
```

The local scope refers to variables that are confined to a specific block, such as a function, loop, or let block. These variables cannot be accessed outside the block where they are defined.

Variables declared inside a function, loop, or block are local by default. You can explicitly use the `local` keyword to restrict a variable to the current block.

```
0  function example()
1      local_var = 42 # Local variable
2      println(local_var)
3  end
4
5  example()
6  # println(local_var) # ERROR: local_var is not defined outside
   ↪ the function
```

```
0  x = 100 # Global variable
1
2  function nested_scope()
3      local x = 50 # Create a local variable named x
4      println(x)   # Output: 50 (local variable)
5  end
6
7  nested_scope()
8  println(x)       # Output: 100 (global variable remains
   ↪ unchanged)
```

Use the `local` keyword for clarity.

3.5 Operators

Arithmetic:

- **Addition:** `+` (e.g., `a + b`)
- **Subtraction:** `-` (e.g., `a - b`)
- **Multiplication:** `*` (e.g., `a * b`)
- **Division:** `/` (e.g., `a / b`)
- **Integer Division:** `div` (e.g., `div(a, b)`)
- **Modulo (Remainder):** `%` (e.g., `a % b`)
- **Floor Division:** `//` (e.g., `a // b`)
- **Power:** `^` (e.g., `a^b`)
- **Negation:** `-` (e.g., `-a`)

Comparison

- **Equality:** `==`
- **Inequality:** `!=` or `≠`
- **Less than:** `<`
- **Greater than:** `>`
- **Less than or equal to:** `<=` or `≤`
- **Greater than or equal to:** `>=` or `≥`

Logical Operators

- **Logical AND:** `&&`
- **Logical OR:** `||`
- **Logical NOT:** `!`

Bitwise Operators

- **Bitwise AND:** `&`
- **Bitwise OR:** `|`
- **Bitwise XOR:** `⊔` (veebar in REPL with \LaTeX)
- **Bitwise NOT:** `~` (Keyboard tilde)
- **Left Shift:** `«`
- **Right Shift:** `»`

Assignment Operators

- **Basic assignment:** `=`
- **Compound assignments:** `+=`, `-=`, `*=`, `/=`, `÷=`, `%=`, `^=`, `&=`, `|=`, `⊔=`, `«=`, `»=`

Inclusive range:

- **Inclusive range:** `:`
- **Exclusive range:** `start:step:end`

Element-wise Operators

- **Dot syntax for element-wise operations:** `.+`, `.-`, `.*`, `./`, `.^`, `./.` (add a `.` before an operator to make it element-wise)

String Operators

- **String concatenation:** `*`
- **String interpolation:** `$`

Ternary

- **Ternary conditional:** `a?b:c` (c-style)

Concatenation and Splitting Operators

- **Horizontal concatenation:** `hcat`
- **Vertical concatenation:** `vcat`
- **Range concatenation:** `...` (splatting)

Broadcasting Operator

- **Dot syntax for broadcasting:** `.`

3.6 Operator precedence

Julia follows PEMDAS

3.7 Ranges

Ranges are compact representations of sequences with a defined step size. They are memory-efficient as they do not explicitly store each element.

```
0  # Default step of 1
1  r1 = 1:5
2  println(collect(r1)) # Output: [1, 2, 3, 4, 5]
3
4  # Custom step size
5  r2 = 1:2:10
6  println(collect(r2)) # Output: [1, 3, 5, 7, 9]
```

3.8 Concatenation and Splitting Operators

Julia provides functions and operators to concatenate or split arrays, matrices, and strings.

Consider the two arrays

```
0  a = [1, 2]
1  b = [3, 4]
```

- **Horizontal Concatenation (`hcat`):** Combines arrays horizontally (columns).

```
0  result = hcat(a, b)
1  println(result) # Output: [1 3; 2 4]
```

- **Vertical Concatenation (`vcat`):** Combines arrays vertically (rows).


```

0 result = vcat(a, b)
1 println(result) # Output: [1; 2; 3; 4]

```

- Concatenation Operator ([...]): Combines arrays inline.

```

0 result = [a; b] # Vertical concatenation
1 println(result) # Output: [1; 2; 3; 4]
2
3 result = [a b] # Horizontal concatenation
4 println(result) # Output: [1 3; 2 4]

```

- Strings can be split using split:

```

0 str = "Hello,World,Julia"
1 result = split(str, ",")
2 println(result) # Output: ["Hello", "World", "Julia"]

```

- Arrays can be split manually with slicing:

```

0 arr = [1, 2, 3, 4, 5]
1 part1 = arr[1:3]
2 part2 = arr[4:end]
3 println(part1) # Output: [1, 2, 3]
4 println(part2) # Output: [4, 5]

```

3.9 Broadcasting

Broadcasting allows you to apply a function or operator element-wise to arrays or other collections. This is achieved using the dot operator (.).

```

0 a = [1, 2, 3]
1 b = [4, 5, 6]
2
3 # Element-wise addition
4 c = a .+ b
5 println(c) # Output: [5, 7, 9]
6
7 # Broadcasting with scalar
8 d = a .* 2
9 println(d) # Output: [2, 4, 6]

```

You can also broadcast custom functions:

```

0  f(x, y) = x + y^2
1  result = f.(a, b)
2  println(result)  # Output: [17, 27, 39]

```

3.10 Comments

Julia uses the pound sign for comments

```

0  # Comment

```

3.11 The in operator

The in operator works similar to Python

```

0  a = [1,2,3]
1  println(1 in a) # True
2  println(5 in a) # False
3
4  if 2 in a
5      println("2 is in a")
6  end
7
8  b = "Julia"
9  println('J' in b) # True
10 println('x' in b) # False

```

3.12 Symbols

A symbol is a unique, immutable identifier often used to represent names, labels, or keys efficiently. Symbols are lightweight and generally faster to compare or store than strings.

- **Creation:** A symbol is created by prefixing an identifier with a colon (:), such as `:symbol_name`. **Immutability:** *Symbols are immutable, meaning they cannot be modified after creation.*
- **Uniqueness:** Two symbols with the same name always refer to the same object in memory:
- **Efficient Comparison:** Comparing symbols is faster than comparing strings because Julia only compares their internal representations.

```

0  a = :hello
1  println(a) # :hello
2  println(typeof(a)) # Symbol

```

Functions

Functions in Julia are first-class, and can therefore be higher order. In programming, first-class functions refer to functions that are treated as first-class citizens. This means they are treated like any other value or variable in the language

- **Assignment:** Functions can be assigned to variables.
- **Passing as Arguments:** Functions can be passed as arguments to other functions.
- **Returning from Other Functions:** Functions can be returned as the result of other functions.
- **Storage in Data Structures:** Functions can be stored in lists, dictionaries, or other data structures.

If functions in a language lack any of these properties, they are not first-class. Such functions are sometimes referred to as second-class functions or non-first-class functions

A higher-order function is a function that satisfies at least one of the following criteria:

- Takes another function as an argument
- Returns a function as its result

A function that neither takes a function as an argument nor returns a function is not a higher-order function. These are sometimes referred to as first-order functions or regular functions

Functions in Julia are created with the keyword *function* and ended with the keyword *end*

```
0  function f()  
1      ...  
2  end
```

Functions can also take arguments and access globals

```
0  x = 5  
1  
2  function f(arg1)  
3      println(x)  
4      println(arg1)  
5  end
```

4.1 Returning

We can specify a return value with the keyword *return*, but if it is omitted, Julia will return the last evaluated expression. Observe the following function

```

0  function f()
1      5 + 10
2  end
3
4  x = f()
5  println(x) # 15

```

4.2 Short form (single-line) functions

For simple functions, you can define them in a single line.

```

0  fn(x,y) = x + y

```

4.2.1 Lambdas

Anonymous functions are unnamed functions and are often used as arguments to other functions.

```

0  (x,y) -> x+y

```

4.3 Multiple Dispatch

Multiple dispatch is a core feature of Julia that allows the selection of a method to execute based on the types of all arguments passed to a function, not just the first argument (as in single dispatch). This makes Julia highly flexible and efficient for polymorphism.

```

0  # Define methods for different type combinations
1  function calculate(a::Int, b::Int)
2      a + b
3  end
4
5  function calculate(a::Float64, b::Float64)
6      a * b
7  end
8
9  function calculate(a::Int, b::Float64)
10     a - b
11 end
12
13 # Call the function
14 println(calculate(3, 4))           # Output: 7 (Int + Int)
15 println(calculate(3.0, 4.0))      # Output: 12.0 (Float64 * Float64)
16 println(calculate(3, 4.0))        # Output: -1.0 (Int - Float64)

```

4.4 Default argument

Default arguments work the way you would expect

```
0 function f(name="Julia")
1     println(name)
2 end
3 f() # Julia
4 f("Nate") # Nate
```

4.5 Variadic functions and kwargs

4.5.1 Variadic functions

We use the ellipsis (...) notation to create variadic functions

```
0 function f(args...)
1     println(args) # (1,2,3)
2     println(typeof(args)) # Tuple{Int64, Int64, Int64}
3 end
4
5 f(1,2,3)
```

4.5.2 Kwargs functions

kwargs functions are functions that accept keyword arguments. Keyword arguments are optional parameters specified by name when calling a function. They are particularly useful for setting default values and making functions more readable and flexible.

Keyword arguments are defined using a semicolon ; instead of a comma in the function definition:

```
0 function f(firstname; lastname="Doe")
1     println("$firstname, $lastname")
2 end
3
4 f("Julia", lastname="Coder") # Julia Coder
5 f("Julia") # Julia Doe
```

Suppose we only had one keyword argument, then we could write

```
0  function f(; arg="Nothing")
1      println(arg)
2  end
3
4  f()
5  f(arg="Something")
```

4.5.3 Variadic kwargs

We use the syntax

```
0  function f(; arg...)
1      for a in arg
2          println(a)
3      end
4  end
5
6  f(arg1="a1", arg2="a2")
```

Control flow

5.1 The let block

In Julia, the `let` block is used to create a new local scope. This allows you to define variables whose lifetime is limited to the block, even if there are variables with the same name in the surrounding scope.

```
0  let [variables = initial_values]
1      ...
2  end
```

For example,

```
0  let a=5,b=10
1      println(a,"\n", b)
2  end
3
4  println(a, "\n", b) # Error
```

If no variable is declared in the `let` block, it still serves as a local scope:

```
0  let
1      y = 5 # Only exists within this block
2      println("y inside let: $y")
3  end
4
5  # println(y) # This would cause an error since y is not defined
   ↪ outside
```

5.2 Conditional blocks (if-elseif-else)

Used for conditional execution of code.

```
0  x = 10
1  if x > 0
2      println("Positive")
3      elseif x == 0
4          println("Zero")
5      else
6          println("Negative")
7      end
```

5.3 For loops

Used for iterating over collections, ranges, or custom iterables.

```
0  for i in 1:5
1      println(i)  # Output: 1, 2, 3, 4, 5
2  end
```

Iterating Over Collections

```
0  arr = ["apple", "banana", "cherry"]
1  for fruit in arr
2      println(fruit)
3  end
```

5.4 While loops

Repeats a block of code as long as a condition is true.

```
0  x = 5
1  while x > 0
2      println(x)
3      x -= 1
4  end
```

5.5 Short-Circuit Control

```
0  x = 5
1  x > 0 && println("Positive")  # Executes if x > 0
2  x < 0 && println("Not Negative")  # Executes if x >= 0
```

5.6 Begin blocks

Groups multiple statements into a single block, useful for one-liners or compound expressions.

```
0  begin
1      x = 1
2      y = 2
3      println(x + y)  # Output: 3
4  end
```


5.7 Anonymous Blocks

You can use anonymous blocks for scoping purposes without naming them explicitly.

```
0  x = 10
1  println("Before block: $x")
2  # New block
3  x = begin
4      y = 20
5      z = y + 10
6      z # Return value
7  end
8  println("After block: $x") # Output: 30
```

5.8 Functions as Control Blocks (do blocks)

```
0  map(x -> x^2, [1, 2, 3]) # Single-line lambda
1  # Equivalent with a do block
2  map([1, 2, 3]) do x
3      x^2
4  end
```

Strings

6.1 Characters

Characters in Julia use the single quote syntax, similar to C.

```
0  ch = 'a'
1
2  str = 'Julia' # ERROR
3  str = "j" # Acceptable
```

In Julia, strings are sequences of characters, and they are used to represent text. Strings are immutable, meaning once a string is created, it cannot be altered directly

6.2 String Operations

In general, you can't perform mathematical operations on strings, even if the strings look like numbers. But there are two exceptions, `*` and `^`.

The `*` operator performs string concatenation, which means it joins the strings by linking them end-to-end. For example:

The `^` operator also works on strings; it performs repetition. For example, `"Spam"^3` is `"SpamSpamSpam"`. If one of the values is a string, the other has to be an integer.

Use `$` to insert variables or expressions into strings.

```
0  name = "Julia"
1  println("Welcome to $name programming!") # Output: "Welcome to
   ↪ Julia programming!"
```

Note: We can also concatenate strings using the string constructor

```
0  a = string("hello", " ", "world")
1  print(a) # Hello world
```

Further, we can use the `repeat` function instead of the string exponentiation operator, which uses `*` under the hood. Observe

```
0  println(repeat("Hello", 2)) # HelloHello
```

6.3 Converting to strings

To convert say, an int, we can use the string constructor.

```
0 a = 10
1 println(typeof(string(a))) # String
```

6.4 Indexing and Slicing

Julia strings are 1-indexed, and you can access individual characters using square brackets.

```
0 str = "Julia"
1 first_char = str[1] # 'J'
2 substring = str[2:4] # "uli"
```

6.5 Into to string Functions

Julia provides a variety of built-in functions to work with strings:

- Length:

```
0 length("hello")
```

- Find and replace:

```
0 str = "apple pie"
1 new_str = replace(str, "apple" => "cherry") # "cherry pie"
```

- Splitting and joining:

```
0 split_str = split("a,b,c", ",") # ["a", "b", "c"]
1 joined_str = join(["a", "b", "c"], "-") # "a-b-c"
```

- Searching:

```
0 occursin("Julia", "Hello, Julia!") # true
1 findfirst("Julia", "Hello, Julia!") # 8 (start index of the
  ↪ match)
```

6.6 Unicode Support

Julia strings are fully Unicode-compliant, meaning they support characters from virtually all languages.

6.7 Mutability

Strings in Julia are immutable, so modifying an existing string directly is not allowed. However, you can create new strings based on the original.

```
0 str = "Hello"
1 str[1] = 'J' # Error: Strings are immutable
```

Instead, create a new string:

```
0 new_str = "J" * str[2:end] # "Jello"
```

6.8 Multiline Strings

For strings spanning multiple lines, use triple double quotes ("""").

```
0 multiline_str = """
1 This is a
2 multiline string.
3 """
```

6.9 Raw Strings

Use raw"""...""" to create raw strings where escape sequences like `\n` are not processed.

```
0 raw_str = raw"Line 1\nLine 2" # "Line 1\\nLine 2"
```

6.10 String Types

- **String:** The default string type in Julia.
- **SubString:** A view of a part of a string, created using slicing. More efficient for substring operations.

```
0 s = "Hello, Julia!"
1 sub_s = SubString(s, 8, 12) # "Julia"
```

Use `String` for general-purpose text and `SubString` for efficient substring manipulation

6.11 String functions

Julia provides a rich set of functions for working with strings. Below is a comprehensive list categorized for easier reference.

6.11.1 Basic String Information

- `length(s)` - Returns the number of characters in the string `s`.
- `sizeof(s)` - Returns the number of bytes in the string `s`.
- `isequal(s1, s2)` - Checks if two strings are exactly the same.
- `isempty(s)` - Returns true if the string `s` is empty

6.11.2 String Manipulation

- `join(strings, delim)` - Joins strings in an array with `delim` as the separator.
- `replace(s, pattern => replacement)` - Replaces all occurrences of `pattern` with `replacement`.
- `strip(s)` - Removes leading and trailing whitespace.
- `lstrip(s)` - Removes leading whitespace.
- `rstrip(s)` - Removes trailing whitespace.
- `chomp(s)` - Removes the trailing newline character.
- `pad(s, n; lpad=n1, rpad=n2)` - Pads a string to a specified length.

6.11.3 Searching and Matching

- `occursin(substr, s)` - Checks if `substr` exists in `s`.
- `findfirst(substr, s)` - Finds the first occurrence of `substr` in `s` and returns its index.
- `findlast(substr, s)` - Finds the last occurrence of `substr` in `s`.
- `findall(substr, s)` - Returns all indices where `substr` occurs in `s`.
- `startswith(s, prefix)` - Checks if `s` starts with `prefix`.
- `endswith(s, suffix)` - Checks if `s` ends with `suffix`.

6.11.4 Splitting and Joining

- `split(s, delim)` - Splits `s` into an array of substrings using `delim`.
- `rsplit(s, delim)` - Splits `s` into substrings from the right.
- `split(s)` - Splits `s` by whitespace.
- `join(strings, delim)` - Joins an array of strings into one string using `delim`.

6.11.5 String Transformation

- `uppercase(s)` - Converts all characters in `s` to uppercase.
- `lowercase(s)` - Converts all characters in `s` to lowercase.
- `titlecase(s)` - Converts the first character of each word to uppercase.
- `capitalize(s)` - Converts the first character of `s` to uppercase.
- `replace(s, pattern => replacement)` - Replaces all occurrences of `pattern` with `replacement`.

6.11.6 Character Inspection

- `isuppercase(c)` - Checks if a character `c` is uppercase.
- `islowercase(c)` - Checks if a character `c` is lowercase.
- `isletter(c)` - Checks if a character `c` is a letter.
- `isdigit(c)` - Checks if a character `c` is a digit.
- `iswhitespace(c)` - Checks if a character `c` is a whitespace.
- `isascii(c)` - Checks if a character `c` is an ASCII character.
- `isprint(c)` - Checks if a character `c` is printable.
- `isalnum(c)` - Checks if a character `c` is alphanumeric.

Splatting and slurping

7.1 Splatting

The ellipsis (...) can unpack elements from collections like arrays or tuples when calling a function or constructing another collection.

```
0  function sum_numbers(a, b, c)
1      return a + b + c
2  end
3
4  nums = [1, 2, 3]
5  result = sum_numbers(nums...) # Equivalent to sum_numbers(1, 2,
    ↪ 3)
6  println(result) # 6
```

7.2 Slurping

In function definitions, ... allows a function to accept a variable number of arguments, known as varargs.

```
0  function greet(names...)
1      for name in names
2          println("Hello, $name!")
3      end
4  end
5
6  greet("Alice", "Bob", "Charlie")
7  # Output:
8  # Hello, Alice!
9  # Hello, Bob!
10 # Hello, Charlie!
```

Arrays

Arrays in Julia are a core data structure used to store collections of elements in an ordered and indexed manner. They can have multiple dimensions and are highly efficient, making them suitable for numerical computations and general-purpose programming.

8.1 Creating Arrays

- 1D Array (Vector):

```
0 arr = [1, 2, 3, 4] # Creates a 1D array with elements 1, 2,  
  ↪ 3, and 4
```

- 2D Array (Matrix):

```
0 mat = [1 2; 3 4] # Creates a 2x2 matrix
```

- Using Array constructor:

```
0 arr = Array{Int64}(undef, 5) # Creates a 1D array of  
  ↪ integers with 5 elements, uninitialized  
1 mat = Array{Float64}(undef, 3, 3) # Creates a 3x3 matrix of  
  ↪ uninitialized floats
```

- Zeros and Ones:

```
0 zeros(3, 3) # 3x3 matrix filled with zeros  
1 ones(4)     # 1D array with four elements, all ones
```

8.2 Accessing elements

Julia uses 1-based indexing

```
0 arr = [10, 20, 30]  
1 println(arr[1]) # Access the first element (output: 10)  
2  
3 subarr = arr[1:2] # Creates a subarray with elements 10 and 20  
4  
5 mat = [1 2; 3 4]  
6 println(mat[1, 2]) # Accesses the element at row 1, column 2  
  ↪ (output: 2)
```


8.3 Modifying arrays

```
0 arr[2] = 25 # Modifies the second element of arr to 25
1
2 push!(arr, 40) # Appends 40 to the end of the array
3 pop!(arr) # Removes the last element from the array
```

8.4 Properties of arrays

```
0 println(size(mat)) # Returns the dimensions of the array
1 println(length(arr)) # Returns the total number of elements
2 println(eltype(arr)) # Returns the type of elements stored in
  ↪ the array
```

8.5 Broadcasting

```
0 arr = [1, 2, 3]
1 result = arr .+ 10 # Adds 10 to each element (output: [11, 12,
  ↪ 13])
```

8.6 Key features

- Arrays in Julia are mutable, meaning you can change their contents.
- Arrays are column-major (like Fortran and MATLAB), which impacts performance for linear algebra operations.
- Built-in functions like `map`, `reduce`, and comprehensions make array manipulation concise and powerful.

8.7 Traversing an array

The most common way to traverse the elements of an array is with a `for` loop. The syntax is the same as for strings

```
0 a = [1,2,3]
1 for item in a
2     println(item)
3 end
```

If you want to write or update the elements, you need the indices. One technique is to use the builtin function `eachindex`

```

0  a = [1,2,3]
1  for idx in eachindex(a)
2      println(idx)
3  end
4
5  # Output
6      1
7      2
8      3

```

8.8 Array slicing

The slice operator `[start : stop : step]`, or just `[start : stop]` also works on arrays.

```

0  a = [1,2,3]
1  println(a[1:2]) # From 1 to 2
2  println(a[1:]) # From 1 to the end
3  println(a[1:]) # From start to the end
4  println(a[begin:end]) # From start to the end
5  println(a[end:-1:begin]) # Reverse the array

```

8.9 Arrays and strings

We can break a string into an array of characters with the `collect` function. We can also split a string into an array of words with the `split` function. The function `join` is the inverse of `split`

```

0  a = "Julia is cool"
1
2  # ['J', 'u', 'l', 'i', 'a', ' ', ' ', 'i', 's', ' ', ' ', 'c', 'o', 'l',
   ↪ 'l']
3  println(collect(a))
4
5  # SubString{String}["Julia", "is", "cool"]
6  println(split(a))
7
8  b = split(a)
9  c = join(b, ' ') # ' ' is the delimiter
10 println(c)

```

8.10 Objects and values

An object is something a variable can refer to. Consider the two statements

```
0 a = "Julia"
1 b = "Julia"
```

To check whether two variables refer to the same object, you can use the \equiv (`\equiv` in the REPL), or `===` operator.

```
0 println(a === b) # True
```

In this example, Julia only created one string object, and both `a` and `b` refer to it. But when you create two arrays, you get two objects

```
0 a = [1,2,3]
1 b = [1,2,3]
2
3 println(a === b) # False
```

In this case we would say that the two arrays are equivalent, because they have the same elements, but not identical, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

To be precise, an object has a value. If you evaluate `[1, 2, 3]`, you get an array object whose value is a sequence of integers. If another array has the same elements, we say it has the same value, but it is not the same object.

8.11 Aliasing

If a refers to an object and you assign $b = a$, then both variables refer to the same object:

```
0 a = [1,2,3]
1 b = a
2 println(b === a) # True
```

The association of a variable with an object is called a *reference*. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is *aliased*.

If the aliased object is mutable, changes made with one alias affect the other

For immutable objects like strings, aliasing is not as much of a problem.

8.12 Array functions

8.12.1 Array Creation

- `Array` - Constructs an uninitialized array.
- `zeros` - Creates an array filled with zeros.
- `ones` - Creates an array filled with ones.
- `fill` - Creates an array filled with a specific value.
- `rand` - Creates an array with random values.
- `randn` - Creates an array with random values from the normal distribution.
- `reshape` - Changes the shape of an array without changing its data.
- `hcat` - Horizontally concatenates arrays.
- `vcat` - Vertically concatenates arrays.
- `hvcat` - Concatenates arrays in multiple dimensions.
- `repeat` - Repeats an array in specified dimensions.
- `range` - Creates an array of evenly spaced numbers.
- `collect` - Converts an iterable (like a range) into an array.

8.12.2 Array Properties

- `size` - Returns the dimensions of the array.
- `length` - Returns the total number of elements in the array.
- `ndims` - Returns the number of dimensions.
- `eltype` - Returns the type of elements stored in the array.
- `axes` - Returns the valid indices for each dimension.
- `eachindex` - Iterates efficiently over all indices.
- `stride` - Returns the memory stride for a given dimension.
- `strides` - Returns the memory strides for all dimensions.
- `isempty` - Checks if the array is empty.

8.12.3 Array Indexing and Slicing

- `getindex` - Accesses array elements (`arr[i]`).
- `setindex!` - Sets array elements (`arr[i] = value`).
- `view` - Creates a view (a lightweight subset) of an array.
- `@view` - A macro for creating views.
- `slice` - Extracts a slice of an array.

8.12.4 Array Modification

- `push!` - Appends an element to the end of a 1D array.
- `pop!` - Removes the last element of a 1D array.
- `append!` - Appends all elements of one array to another.
- `insert!` - Inserts an element at a specific position.
- `deleteat!` - Deletes elements at specified indices.
- `splice!` - Removes elements and optionally replaces them.
- `empty!` - Removes all elements from an array.
- `resize!` - Resizes an array to a specified size.

8.12.5 Combining and Splitting Arrays

- `cat` - Concatenates arrays along a specified dimension.
- `hcat`, `vcate`, `hvcate` - Horizontally, vertically, or multi-dimensionally concatenate arrays.
- `splitdims` - Splits a multi-dimensional array into its slices along a specified dimension.
- `reshape` - Reshapes an array into a different dimensionality.
- `permutedims` - Permutes the dimensions of an array.

8.12.6 Mapping and Broadcasting

- `map` - Applies a function to each element of an array.
- `broadcast` - Applies a function element-wise to arrays.
- `@.` - Macro for broadcasting every operation in an expression.
- `map!` - In-place version of `map`.
- `broadcast!` - In-place version of `broadcast`.

8.12.7 Statistical Functions

- `sum` - Computes the sum of all elements.
- `prod` - Computes the product of all elements.
- `mean` - Computes the mean of all elements.
- `std` - Computes the standard deviation.
- `var` - Computes the variance.
- `maximum` - Finds the maximum value.
- `minimum` - Finds the minimum value.
- `argmax` - Returns the index of the maximum value.
- `argmin` - Returns the index of the minimum value.
- `count` - Counts the number of elements satisfying a condition.

8.12.8 Array Manipulation

- reverse - Reverses the order of elements.
- sort - Sorts the elements of an array.
- sort! - Sorts an array in-place.
- shuffle - Randomly shuffles the elements.
- permute! - Rearranges elements in a specific order.
- unique - Returns unique elements of an array.
- filter - Filters elements based on a condition.
- findall - Finds all indices where a condition is true.
- findfirst - Finds the first index where a condition is true.
- findlast - Finds the last index where a condition is true

8.12.9 Matrix-Specific Functions

- transpose - Computes the transpose of a matrix.
- adjoint - Computes the conjugate transpose.
- det - Computes the determinant of a square matrix.
- inv - Computes the inverse of a square matrix.
- rank - Computes the rank of a matrix.
- trace - Computes the trace of a square matrix.
- diag - Extracts or creates a diagonal matrix.
- tril - Extracts the lower triangular part of a matrix.
- triu - Extracts the upper triangular part of a matrix.

8.12.10 Linear Algebra Functions

- eigen - Computes eigenvalues and eigenvectors.
- svd - Computes the singular value decomposition.
- qr - Computes the QR factorization.
- lu - Computes the LU decomposition.
- chol - Computes the Cholesky decomposition.
- pinv - Computes the pseudo-inverse.

Other containers

9.1 Tuples

Immutable ordered collections of elements

```
0 t = (1, "hello", 3.14)
1 println(t[2]) # Access elements by index (1-based)
```

9.2 Named Tuples

Tuples with named fields for each element.

```
0 nt = (a = 1, b = "hello", c = 3.14)
1 println(nt.b) # Access elements by name
```

9.3 Dictionaries

Key-value pairs, like hash maps in other languages.

```
0 dict = Dict{"a" => 1, "b" => 2}
1 dict["c"] = 3 # Add a new key-value pair
2 println(dict["b"]) # Access value by key
```

9.4 Sets

Unordered collections of unique elements.

```
0 s = Set{1, 2, 2, 3}
1 push!(s, 4) # Add an element
2 println(3 in s) # Check membership
```

9.5 Ranges

Compact representations of sequences of numbers.

```
0 r = 1:2:10 # Start at 1, increment by 2, stop at 10
1 println(collect(r)) # Convert to an array
```

9.6 Sparse Arrays

Efficient storage for arrays with mostly zero elements.

```
0 using SparseArrays
1 sp = sparse([1, 0, 0; 0, 2, 0; 0, 0, 3])
2 println(sp[2, 2]) # Access elements
```

9.7 Deques (from the DataStructures package)

Double-ended queues, allowing efficient insertion and deletion from both ends.

```
0 using DataStructures
1 dq = Deque([1, 2, 3])
2 push!(dq, 4) # Add to the end
3 pushfirst!(dq, 0) # Add to the front
4 println(dq)
```

9.8 Stores elements with associated priorities.

```
0 using DataStructures
1 pq = PriorityQueue()
2 enqueue!(pq, "task1", 1) # Add element with priority
3 enqueue!(pq, "task2", 0)
4 println(dequeue!(pq)) # Returns "task2" (lowest priority first)
```


Dictionaries

The function `Dict` creates a new dictionary with no items. To add items to the dictionary, you can use square brackets:

```
0 d = Dict()
1 d["key1"] = "item1"
2
3 println(d) # Dict{Any, Any}{"key1" => "item1"}
```

We can create a dictionary and fill it with values all in one motion with the constructor function and the `=>` syntax

$$d = \text{Dict}(\text{"key1"} \Rightarrow \text{"item1"}, \text{"key2"} \Rightarrow \text{"item2"})$$

10.0.1 Getting the length

The length function works on dictionaries; it returns the number of key-value pairs:

```
0 println(length(d))
```

10.1 Getting the keys

The function `keys` returns a collection with the keys of the dictionary:

```
0 println(keys(d)) # e["key2", "key1"]
```

10.1.1 Checking if a key exists

You can use the `in` operator `∈` paired with `keys` to check if a key exists

```
0 ks = keys(d)
1 println("key1" in ks) # True
2 println("key3" in ks) # False
```

10.2 Getting the values

You can use the `values` function to get a collection of the dict's values

```
0  vals = values(d)
1  println("item1" in vals)
2  println("item3" in vals)
```

Note: For dictionaries, Julia uses an algorithm called a hash table

10.3 Looping through dictionaries

We can use a `for` loop to loop through the key value pairs

```
0  for item in d
1      println(item)
2  end
3
4  # Output
5  "key2" => "item2"
6  "key1" => "item1"
```

Note that `item` is a `pair{String, String}`. Julia offers the C++ style `.first`, `.second` to access the key and value separately

```
0  for item in d
1      println(item.first)
2  end
3
4  # output
5  key2
6  key1
```

Pairs

In Julia, a pair is a simple data structure that holds two values, often used to represent key-value pairs. Pairs are created using the `=>` operator and are of the type `Pair`.

```
o p1 = "f" => 100
```

11.1 Key features

- **Immutable:** Pairs are immutable, meaning you cannot change their key or value after creation.
- **Type:** A pair has the type `Pair{T1, T2}` where `T1` is the type of the key and `T2` is the type of the value.
- **Common Use:** Pairs are commonly used in dictionaries, where they represent individual key-value entries.

11.2 Accessing elements

Use `.first` for the key, `.second` for the value

Sets

Sets in Julia represent unordered collections of unique elements. They are implemented as the `Set` type and are part of Julia's standard library. Sets are efficient for membership testing and can perform common set operations such as union, intersection, and difference.

12.1 Creating sets

You can create a `Set` using the `Set` constructor or by converting an iterable.

```
0 s = Set([1,2,3])
1 empty_set = Set()
```

Or from a range

```
0 s = Set(1:5) # Output: Set{Int64}([1, 2, 3, 4, 5])
```

Sets in Julia are mutable, meaning you can add or remove elements after creation.

12.2 Basic Operations

Sets support operations like adding, removing, and checking membership.

```
0 2 in s
```

Adding and removing elements

```
0 push!(s,4)
1 pop!(s)
2 delete!(s,2)
```

Getting the length

```
0 length(s)
```

Julia provides functions for common set operations:

```
0 a = Set([1,2,3])
1 b = Set([2,3,4])
2 c = union(a, b)
3 d = intersect(a, b)
4 e = setdiff(a, b) # Elements in `a` but not in `b`
5 symdiff(a, b) # Elements in `a` or `b`, but not both
6
7 issubset(Set([1, 2]), a)
8 issuperset(a, Set([1, 2]))
```

Comprehensions

Comprehensions in Julia provide a concise and expressive way to create collections such as arrays, sets, or dictionaries by specifying their elements in terms of an existing set of data, combined with transformations and filters. They are similar to comprehensions in Python, but with Julia's unique syntax and capabilities.

```
0 [expression for item in collection]
```

- **expression:** A computation or transformation to apply to each element.
- **item:** A variable representing each element of the collection.
- **collection:** The iterable (e.g., array, range) to traverse.

For example

```
0 a = [x^2 for x in 1:5]
```

With a Condition:

```
0 even_squares = [x^2 for x in 1:10 if x % 2 == 0]
1 # Output: [4, 16, 36, 64, 100]
```

Nested Loops:

```
0 pairs = [(x, y) for x in 1:2, y in 3:4]
```

13.1 Set Comprehensions

You can create sets using a similar syntax but with Set:

```
0 squares_set = Set(x^2 for x in 1:5)
1 # Output: Set([1, 4, 9, 16, 25])
```

13.2 Dictionary Comprehensions

Dictionary comprehensions use a key-value pair syntax:

```
0 squared_dict = Dict{x => x^2 for x in 1:5}
1 # Output: Dict{1 => 1, 2 => 4, 3 => 9, 4 => 16, 5 => 25}
```

Comprehensions are often more expressive for creating collections than broadcasting, but both are idiomatic in Julia:

```
0  # Using comprehension
1  squared = [x^2 for x in 1:5]
2
3  # Using broadcasting
4  squared = (1:5) .^ 2
```

Iterators

iterators provide a way to traverse through elements of a collection (or any iterable object) in a systematic way

For an object to be iterable in Julia, it must implement the `iterate` function. The `iterate` function acts as the foundation for traversing elements of a collection.

```
0  iterate(iterable, state)
```

- **iterable:** The object to iterate over.
- **state:** Tracks the current position in the iteration.
- **Returns:** A tuple (value, new_state) if there are elements remaining, or nothing when iteration is complete

Iteration is automatically handled by Julia's built-in constructs, like loops:

```
0  for x in collection
1      println(x)
2  end
```

You can make any type iterable by defining the `iterate` function.

```
0  struct Counter
1      max::Int
2  end
3
4  function Base.iterate(counter::Counter, state=1)
5      if state > counter.max
6          return nothing
7      end
8      return (state, state + 1)
9  end
10
11 counter = Counter(5)
12 for x in counter
13     println(x)
14 end
```


Structs

In Julia, a struct (short for "structure") is a composite data type that groups together related variables (called fields) under one entity. It is a way to create custom types in Julia, allowing you to organize data more effectively and work with it in a structured manner.

15.1 Key features

- **Immutable by Default:** Fields in a struct cannot be modified once the struct is created. This immutability ensures safety and efficiency in operations.
- **Custom Types:** Structs allow you to define your own types, which can be used in functions, arrays, or as fields in other structs.
- **Can Be Made Mutable:** By using mutable struct, the fields can be modified after the struct is instantiated.

15.2 Creating structs

```
0 struct StructName
1     field1::DataType1
2     field2::DataType2
3     # Add as many fields as needed
4 end
```

For example,

```
0 struct Point
1     x::Float64
2     y::Float64
3 end
```

To create an instance of a struct:

```
0 p = Point(3.0, 4.0)
1
2 # Access fields with the dot notation
3 println(p.x) # Output: 3.0
4 println(p.y) # Output: 4.0
```

15.3 Mutable Structs

Use mutable struct when you want to modify fields after the struct is created

```
0 mutable struct PointMutable
1     x::Float64
2     y::Float64
3 end
4
5 p = PointMutable(3.0, 4.0)
6 p.x = 5.0 # Now p.x is updated to 5.0
```

15.4 Structs Without Field Type Annotations

Fields can omit type annotations, making them more flexible but less type-specific

```
0 struct FlexiblePoint
1     x
2     y
3 end
4
5 p = FlexiblePoint(3.0, "hello")
```

15.5 Default Values

Julia does not directly support default field values in struct definitions. However, you can achieve this using constructors

```
0 struct PointDefault
1     x::Float64
2     y::Float64
3     PointDefault(x::Float64, y::Float64 = 0.0) = new(x, y)
4 end
5
6 p1 = PointDefault(3.0) # y defaults to 0.0
7 p2 = PointDefault(3.0, 4.0) # y is explicitly set to 4.0
```

15.6 Parametric Structs

Parametric structs allow you to define types that work with different data types

```

0 struct PointParametric{T}
1     x::T
2     y::T
3 end
4
5 p = PointParametric{Int}(3, 4)
6 q = PointParametric{Float64}(3.0, 4.0)

```

Here, {T} is a type parameter that makes the struct generic.

15.7 Abstract types and Subtypes

Julia does not have traditional object-oriented inheritance like some other languages (e.g., Python, Java, C++). However, Julia supports hierarchical relationships using its type system. Specifically, Julia enables inheritance via abstract types and parametric types.

15.7.1 Abstract types

In Julia, abstract types are part of the type hierarchy and are used to define a blueprint for other types. They are placeholders in the type system, allowing you to group related types without specifying implementation details. Abstract types cannot be instantiated directly but serve as parent types for concrete types.

- **Abstract:** You cannot create instances of an abstract type.
- **Purpose:** Abstract types define a common interface or grouping for a set of related types.
- **Inheritance:** Concrete types or other abstract types can inherit (subtype) from an abstract type using the <: symbol.

```

0 abstract type AbstractName end
1
2 # For example
3 abstract type Shape end

```

Here, Shape is an abstract type that can act as a parent for all kinds of shapes like circles, squares, and triangles.

15.7.2 Concrete types

Concrete types inherit from abstract types by using the <: symbol.

```

0  abstract type Shape end # Abstract type
1
2  # Concrete subtypes
3  struct Circle <: Shape
4      radius::Float64
5  end
6
7  struct Rectangle <: Shape
8      width::Float64
9      height::Float64
10 end

```

15.8 Usage of Abstract Types

Abstract types are often used with functions and multiple dispatch to define methods that can operate on a group of related types.

```

0  abstract type Shape end
1
2  struct Circle <: Shape
3      radius::Float64
4  end
5
6  struct Rectangle <: Shape
7      width::Float64
8      height::Float64
9  end
10
11 # Define a generic method for `Shape`
12 area(::Shape) = throw(NotImplementedError("area method not
    ↳ implemented for this shape"))
13
14 # Specialize the method for each subtype
15 area(c::Circle) = * c.radius^2
16 area(r::Rectangle) = r.width * r.height
17
18 # Usage
19 circle = Circle(5.0)
20 rect = Rectangle(4.0, 3.0)
21
22 println(area(circle)) # Output: 78.53981633974483
23 println(area(rect)) # Output: 12.0

```

Julia's type hierarchy has a top-level abstract type called `Any`, from which all other types derive.

```

0  abstract type Animal end # Abstract type
1
2  struct Dog <: Animal
3      name::String
4  end
5
6  struct Cat <: Animal
7      name::String
8  end
9
10 # Animal is a subtype of Any
11 println(Animal <: Any) # Output: true

```

15.9 Abstract Types and Parametric Types

Abstract types can also be parametric, which means they accept type parameters. This allows for more generic and flexible type definitions.

```

0  abstract type Container{T} end # Abstract type with a type
   ↪ parameter T
1
2  struct Box{T} <: Container{T}
3      value::T
4  end
5
6  # Usage
7  int_box = Box{Int}(42)
8  string_box = Box{String}("Hello")
9
10 println(typeof(int_box)) # Output: Box{Int64}
11 println(typeof(string_box)) # Output: Box{String}

```

15.10 Constructors

When you define a struct in Julia, it automatically gets a default constructor that initializes its fields.

```

0  struct Point
1      x::Float64
2      y::Float64
3  end
4
5  # Using the default constructor
6  p = Point(3.0, 4.0) # Creates a Point with x = 3.0, y = 4.0

```

Custom constructors let you define how objects of the struct should be created. These can include additional logic or default values.

```

0  struct Point
1      x::Float64
2      y::Float64
3
4      # Custom constructor
5      Point(x, y) = new(x, y) # Explicitly calls the default
    ↪ constructor
6  end
7
8  # Example usage
9  p = Point(3, 4) # Automatically converts integers to Float64

```

Memory

16.1 Stack Allocation

Used for small, fixed-size objects like immutable values and local variables. Allocation and deallocation are very fast because they operate on the call stack

```
0 function add(a, b)
1     return a + b
2 end
```

Here, a and b are allocated on the stack.

16.2 Heap Allocation

Used for larger or dynamically sized objects, such as arrays and mutable structs. Allocation is slower than stack allocation because it uses the heap memory, which requires garbage collection to free unused memory.

```
0 arr = [1, 2, 3] # Array is allocated on the heap
```

16.3 Garbage collection

Julia uses automatic garbage collection to free memory that is no longer in use. The garbage collector periodically identifies objects in heap memory that are no longer referenced and reclaims that memory.

You can trigger garbage collection manually using

```
0 GC.gc()
```

16.4 Memory Views and Sharing

Julia avoids unnecessary copying by using views for slicing arrays. Instead of creating a new array, a view provides a lightweight reference to the original array's data.

```
0 arr = [1, 2, 3, 4]
1 v = @view arr[1:2] # Does not allocate new memory
```

16.5 Static arrays

For small, fixed-size arrays, use `StaticArrays.jl` to avoid heap allocations.

```
0 using StaticArrays
1 v = SVector{1.0, 2.0, 3.0}
```

16.6 Advanced: Memory Buffers and Pointers

Julia allows low-level memory management through buffers and pointers. For example

```
0 buf = Base.Libc.malloc(1024) # Allocate 1024 bytes manually
1 Base.Libc.free(buf)          # Free the allocated memory
```

16.7 References

In Julia, references play a key role in determining how objects are handled in memory, particularly for mutable and immutable objects. Understanding references is crucial for working effectively with Julia's memory model, especially for mutable objects like arrays and mutable struct

A reference is essentially a pointer to an object's location in memory. When you assign or pass objects in Julia, whether by value or by reference depends on the type of object.

16.7.1 Immutable types

Immutable types are passed by value. When you assign or pass an immutable object (e.g., `Int`, `Float64`, or a struct without mutable), Julia creates a copy of the object rather than a reference.

```
0 x = 5 # `x` holds the value directly
1 y = x # `y` is a copy of `x`
2 y += 1
3 println(x) # 5 (unchanged)
4 println(y) # 6 (modified copy)
```

16.7.2 Mutable Types

Mutable types are passed by reference. When you assign or pass a mutable object (e.g., arrays or mutable struct), Julia creates a reference to the same memory location. Any modifications affect the original object.


```

0 arr = [1, 2, 3] # Array is mutable
1 arr_copy = arr # `arr_copy` references the same array
2 arr_copy[1] = 10 # Modify through `arr_copy`
3 println(arr) # [10, 2, 3] (original is modified)

```

16.7.3 Copying Mutable Objects

If you need to modify a mutable object without affecting the original, you must create a deep or shallow copy.

- **Shallow Copy (copy):** Creates a new container but references the same inner objects.

```

0 a = [[1, 2], [3, 4]]
1 b = copy(a)
2 b[1][1] = 99
3 println(a) # [[99, 2], [3, 4]] (inner objects are shared)

```

- **Deep Copy (deepcopy):** Recursively copies all objects, creating completely independent copies.

```

0 a = [[1, 2], [3, 4]]
1 c = deepcopy(a)
2 c[1][1] = 99
3 println(a) # [[1, 2], [3, 4]] (completely independent)

```

16.7.4 References in Functions

Mutable objects are passed by reference to functions, so any modifications made within the function affect the original object

```

0 function modify_array(arr)
1     arr[1] = 100
2 end
3
4 x = [1, 2, 3]
5 modify_array(x)
6 println(x) # [100, 2, 3] (modified in-place)

```

If you want to avoid modifying the original, you can pass a copy

```

0 modify_array(copy(x))

```

16.7.5 Ref for Explicit References

The `Ref` type in Julia provides a way to create explicit references to values, even for immutable types. This is useful for interfacing with low-level code or when you want mutable behavior for a normally immutable value.

```
0  r = Ref{Int}(5) # Create a reference to an immutable value
1  println(r[])    # Access the value (5)
2  r[] = 10        # Modify the value
3  println(r[])    # Access the updated value (10)
```

The syntax `r[]` in Julia is used to access or modify the value stored in a `Ref` object. The `Ref` type in Julia is essentially a container for a single value, allowing you to treat the contained value as mutable, even if it's normally immutable (like numbers).

Working with files

17.1 The basics

Files are accessed using their file paths, which can be absolute or relative. You specify how the file will be opened with the following modes

- **"r"**: Read-only.
- **"w"**: Write-only (truncates if the file exists).
- **"a"**: Append mode (writes at the end of the file).
- **"r+"**: Read and write.
- **"w+"**: Read and write (truncates if the file exists).
- **"a+"**: Read and write (writes at the end of the file).

17.2 Reading Files

Reading Entire File

```
0 content = read("example.txt", String)
1 println(content)
```

Line-by-Line Reading

```
0 open("example.txt", "r") do file
1     for line in eachline(file)
2         println(line)
3     end
4 end
```

17.3 Writing Files

Writing Strings

```
0 write("output.txt", "Hello, Julia!")
```

Appending to a File

```
0 open("output.txt", "a") do file
1     write(file, "\nAdding a new line.")
2 end
```

17.4 Checking File Properties

- Does the file exist?

```
0  if isfile("example.txt")
1      println("File exists!")
2  else
3      println("File does not exist.")
4  end
```

- Is it a directory?

```
0  if isdir("my_directory")
1      println("It's a directory.")
2  end
```

- Get file size

```
0  size = filesize("example.txt")
1  println("File size: $size bytes")
```

17.5 Directory Operations

- List Files in Directory

```
0  for entry in readdir(".")
1      println(entry)
2  end
```

- Create Directory

```
0  mkdir("new_directory")
```

- Remove File or Directory

```
0  rm("example.txt") # Removes a file
1  rm("empty_directory", recursive=true) # Removes a directory
   ↪ and its contents
```

17.6 Example: Copying a File

```
0  open("source.txt", "r") do src
1      open("destination.txt", "w") do dest
2          write(dest, read(src, String))
3      end
4  end
```

Error handling

18.1 Throwing errors

To signal an error, you use the `throw` function. Julia provides several built-in error types, and you can define custom error types if needed.

- **ErrorException:** A general-purpose error.
- **ArgumentError:** Raised when a function argument is invalid.
- **DomainError:** Raised when a value is outside the domain of a function.
- **MethodError:** Raised when a method is not found for the given arguments.
- **KeyError:** Raised when a key is not found in a dictionary.
- **BoundsError:** Raised when an index is out of bounds.
- **DivideError:** Throw to prevent division by zero

Note: Division by zero yields `Inf` (infinity) in Julia, no exception will be thrown automatically

```
0  function divide(a, b)
1      if b == 0
2          throw(DivideError()) # Raise a DivideError
3      end
4      return a / b
5  end
```

18.2 Handling Errors

You use the try-catch block to handle errors. If an error occurs in the try block, control is transferred to the appropriate catch block.

```
0  try
1      # Code that might throw an error
2  catch e
3      # Handle the error
4  end
```

18.3 Finally Block

The finally block runs after the try and catch blocks, regardless of whether an error occurred. It is typically used for cleanup operations.

```
0  function file_operations()  
1      file = open("example.txt", "w")  
2      try  
3          write(file, "Hello, world!")  
4      catch e  
5          println("Caught an error: $e")  
6      finally  
7          close(file) # Ensures the file is closed  
8      end  
9  end
```

18.4 Custom Errors

You can define custom error types by creating a struct that inherits from Exception

```
0  struct CustomError <: Exception  
1      msg::String  
2  end  
3  
4  function risky_function()  
5      throw(CustomError("This is a custom error"))  
6  end  
7  
8  try  
9      risky_function()  
10 catch e  
11     println("Caught error: $e")  
12 end
```

Regular expressions

Regular expressions (regex) in Julia are a powerful tool for matching and manipulating strings based on patterns. Julia's regular expressions are based on the Perl-compatible Regular Expressions (PCRE) library, which provides rich functionality for pattern matching.

19.1 Basic Syntax

In Julia, regular expressions are written as strings prefixed with the *r* macro

```
0  r"pattern"
```

The *r* before the quotes signifies that the string contains a regular expression.

```
0  r"\d+" # Matches one or more digits
1  r"\s"  # Matches any whitespace character
2  r"[A-Za-z]+" # Matches one or more alphabetic characters
```

19.2 Key Functions for Regex in Julia

- **occursin:** Checks if a pattern exists in a string.

```
0  occursin(r"\d+", "There are 123 apples.") # true
```

- **match:** Returns the first match of the pattern in a string.

```
0  m = match(r"\d+", "There are 123 apples.")
1  println(m.match) # "123"
```

- **eachmatch:** Returns an iterator of all matches in a string.

```
0  for m in eachmatch(r"[a-z]+", "Julia is fun")
1      println(m.match)
2  end
3  # Output:
4  # "ulia"
5  # "is"
6  # "fun"
```

- **replace:** Replaces parts of a string that match the pattern.


```
0  replace("123-456", r"\d", "X")  # "XXX-XXX"
```

- **split:** Splits a string based on a pattern.

```
0  split("one, two, three", r",\s*")  # ["one", "two", "three"]
```

- **ismatch:** Checks if the entire string matches the pattern.

```
0  ismatch(r"^\d+$", "12345")  # true
1  ismatch(r"^\d+$", "123abc")  # false
```

19.3 Flags

Flags modify the behavior of the regex.

- **i:** Case-insensitive matching.
- **m:** Multi-line mode (allows `^` and `$` to match the start and end of lines).
- **s:** Dot-all mode (makes `.` match newline characters).

```
0  r"hello"i  # Case-insensitive match for "hello"
```

19.4 RegexpMatch Object

The `match` and `eachmatch` functions return `RegexpMatch` objects. These contain details about the match:

- **m.match:** The full match.
- **m.captures:** Captured groups.

```
0  m = match(r"(\d+)-(\d+)", "123-456")
1  println(m.match)      # "123-456"
2  println(m.captures)   # ["123", "456"]
```

Math

20.1 Plots.jl

20.1.1 Installation

To use Plots.jl, first install the package:

```
0 using Pkg
1 Pkg.add("Plots")
```

Then, load the package

```
0 using Plots
```

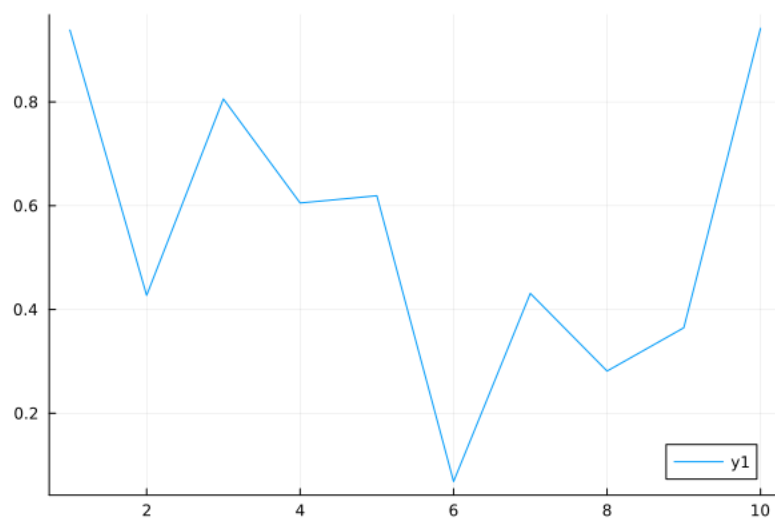
20.1.2 Create and save plots

Consider the simple example

```
0 x = 1:10
1 y = rand(10) # Random data
2 plot(x, y)
```

Save your plot as an image or PDF:

```
0 savefig("path")
```



20.1.3 Titles and Labels

```
o plot(x, y, title="My Plot", xlabel="X-axis", ylabel="Y-axis")
```

20.1.4 Line and Marker Styles

The `linestyle` attribute controls the style of the plot's lines. Common options are

- **:solid** (default) – A solid line
- **:dash** A dashed line
- **:dot** A dotted line
- **:dashdot** A dash-dotted line
- **:dashdotdot** A dash-dot-dot line
- **:none** No line (useful for scatter plots)

The `marker` attribute defines the symbols used to represent individual data points. `MarkerSize` defines the size of the marker

- **:none** : No marker
- **:circle** : Circle markers
- **:rect** : Rectangular markers
- **:star** : Star markers
- **:cross** : Cross markers
- **:x** : X markers
- **:diamond** : Diamond markers
- **:pentagon** : Pentagon markers
- **:hexagon** : Hexagon markers
- **:upwardtriangle** : Upward-pointing triangle markers
- **:downtriangle** : Downward-pointing triangle markers
- **:righttriangle** : Rightward-pointing triangle markers
- **:lefttriangle** : Leftward-pointing triangle markers
- **:vline** : Vertical line markers
- **:hline** : Horizontal line markers
- **:plus** : Plus markers
- **:circlex** : Circle with a cross inside
- **:xcross** : X inside a circle

```
o plot(x, y, marker=:star, markersize=8)
```

The color attribute defines the color of the line

```
o plot(x,y, color=:pink)
```

20.1.5 Legends

The legend attribute controls whether the legend is displayed and its position. Options include

- :top (default) – Legend at the top.
- :bottom – Legend at the bottom.
- :left – Legend on the left.
- :right – Legend on the right.
- :none – Hides the legend.
- legendfontsize adjusts the font size of the legend text.
- legendfontcolor changes the color of the legend text
- legendfontfamily Specifies the font family for the legend text.
- legendbackgroundcolor Sets the background color of the legend box.
- legendborder Toggles the border around the legend box.
- legendtitle Adds a title to the legend.
- legendtitlefontsize Changes the font size of the legend title.
- legendmarker Specifies whether to display markers in the legend.
- legendposition Fine-tunes the legend's position using coordinates.

20.1.6 Plot Types

Plots.jl supports various types of plots. Some common ones are:

- **Scatter Plot:**

```
o scatter(x, y)
```

- **Bar Plot:**

```
0 bar(x, y)
```

- **Histogram:**

```
0 histogram(randn(1000), bins=20)
```

- **Heatmap:**

```
0 heatmap(rand(10, 10))
```

- **3D Plot:**

```
0 x = 1:10
1 y = 1:10
2 z = [sin(i) * cos(j) for i in x, j in y]
3 plot(x, y, z, st=:surface)
```

20.1.7 Create multiple subplots in a grid:

```
0 plot(x, y, layout=(2, 2), legend=false) # 2x2 grid, no legends
1 plot!(x, y .+ 1) # Add to a specific subplot
```

Packages

Julia's package manager, Pkg, is integrated into the language and is used for installing, updating, and managing packages. Use the REPL's package mode by typing `]` in the Julia REPL. Commands can also be run programmatically using the Pkg module.

21.1 Installing a Package

- **REPL:** `add PackageName`
- **Programmatically:** `using Pkg; Pkg.add("PackageName")`

21.2 Updating Installed Packages

- **REPL:** `update`
- **Programmatically:** `Pkg.update()`

21.3 Removing a Package:

- **REPL:** `rm PackageName`
- **Programmatically:** `Pkg.rm("PackageName")`

21.4 Listing Installed Packages

- **REPL:** `status`
- **Programmatically:** `Pkg.status()`

21.5 To use a package in your project

Install it using `Pkg.add`, Load it into your session using the `using` or `import` keywords

```
0  using packagename
1  # or
2  import packagename
```

Recall: `using` loads the package and makes its exported functions directly accessible. `import` loads the package but requires you to explicitly qualify its functions (e.g., `PackageName.function`).

21.6 Build or rebuild a package

- **REPL:** `build packagename`
- **Programmatically:** `using Pkg; Pkg.build("PackageName")`