Julia for Numerical Analysis

Introduction to Scientific Computing

Meik Hellmund

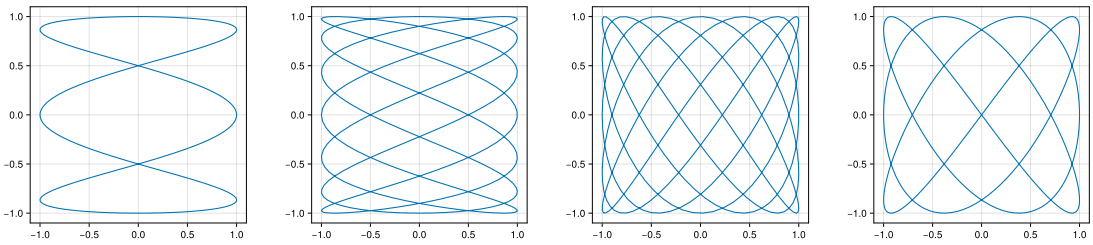
Contents

What is Julia?

Julia is a relatively new, modern programming language designed for *scientific computing*.

A code example:

```
using CairoMakie
a = [3, 7, 5, 3]
b = [1, 3, 7, 4]
δ = π/2
t = LinRange(-π, π, 300)
f = Figure(size=(1600, 360))
for i in 1:4
    x = sin.( a[i] .* t .+ δ )
    y = sin.( b[i] .* t )
    lines(f[1, i], x, y, axis=(; aspect = 1))
end
f
```



History

- 2009: Development started at MIT's *Computer Science and Artificial Intelligence Laboratory*
- 2012: First release (v0.1)
- 2018: Version 1.0 released
- February 2026: Version 1.12.5

In their 2012 inaugural blog post Why we created Julia, the developers provide an insightful and humorous overview of their objectives and motivations for creating Julia.

A photo of *Stefan Karpinski, Viral Shah, Jeff Bezanson*, and *Alan Edelman* can be found here: https://news.mit.edu/2018/julia-language-co-creators-win-james-wilkinson-prize-numerical-software-1226.

Why Julia?

from The fast track to Julia

"Julia is an open-source, multi-platform, high-level, high-performance programming language for technical computing.

Julia has an LLVM-based JIT compiler that allows it to match the performance of languages such as C and FORTRAN without the hassle of low-level code. Because the code is compiled on the fly you can run (bits of) code in a shell or REPL, which is part of the recommended workflow.

Julia is dynamically typed, provides multiple dispatch, and is designed for parallelism and distributed computation.

Julia has a built-in package manager."

open source

- open development on GitHub

- implementations for all common operating systems

high-performance programming language for technical computing

- many functions for *scientific computing* built-in
- (intentional) similarity to Python, R and Matlab
- complex calculations in a few lines
- simple interface to other languages like C or Python

JIT compilation

- supports interactive workflow via the `read-eval-print loop (REPL)`
- just-in-time (JIT) compilation
- resulting in runtimes comparable to static languages like C/C++, Fortran, or Rust

a built-in package manager

- huge *ecosystem* of easily installable packages, e.g.
  - Mathematical Optimization
  - Machine Learning
  - Data Visualization
  - Differential Equations
  - Mathematical Modeling

Selected Links

- Documentation – the official documentation
- Cheat Sheet – "a quick overview"
- Introducing Julia – a WikiBook
- The Julia Express – Julia in 16 pages
- Think Julia – introduction to programming using Julia as first language
- The Julia Forum
- For the eyes: Examples for the Julia graphics package `Makie`

1. Development Environments

For this course, we will use the web-based development environment Jupyterhub. It is available to all participants for the duration of the course.

For long-term work, a personal installation is recommended.

1.1 Installing on Your Own Computer (Linux/MacOS/MS Windows)

1. Install Julia with the installation and update manager **juliaup**: https://github.com/JuliaLang/juliaup/.
2. Install **Visual Studio Code** as editor/IDE: https://code.visualstudio.com/.
3. Install the **Julia language extension** in VS Code: https://www.julia-vscode.org/docs/stable/gettingstarted/.

Getting started:

- Create a new file with the extension `.jl` in VS Code
- Write Julia code
- `Shift-Enter` or `Ctrl-Enter` at the end of a statement or block starts a Julia-REPL: the code is copied to the REPL and executed
- Key bindings for VS Code and for Julia in VS Code

1.1.1 The Julia-REPL

When Julia is started directly from the command line, the Julia-REPL (*read-eval-print* loop) opens, allowing interactive work.

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.12.5 (2026-02-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

1.2 Working on the Jupyterhub Webserver

1.2.1 Jupyterhub & Jupyter

- JupyterHub is a multi-user server for Jupyter notebooks.
- Jupyter is a web-based interactive programming environment
- Originally written for and in Python, but now supports many programming languages
- In Jupyter, one works with so-called *notebooks*: structured text files (JSON) with the file extension `*.ipynb`.

Our Jupyterhub server: https://misun103.mathematik.uni-leipzig.de/

After logging into Jupyterhub, a file manager appears:

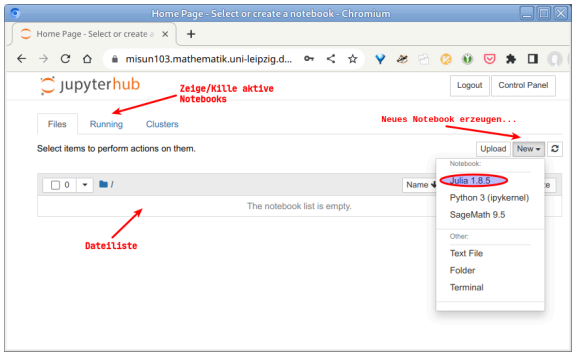


Figure 1.1: Jupyterhub file manager

This can be used to:

- open existing *notebooks*,
- create new *notebooks*,
- upload files, e.g., notebooks, from your local computer,
- log out when finished (Please don't forget this!)

1.2.2 Jupyter notebooks

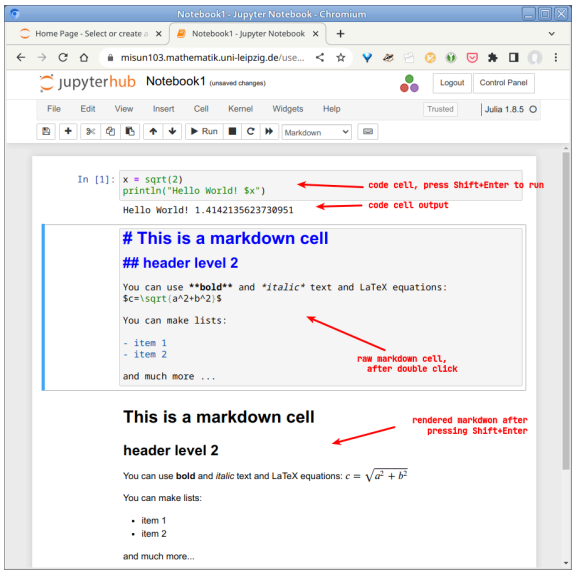


Figure 1.2: Jupyter Notebook

Notebooks consist of cells. Cells can contain

- Code, or
- Text/documentation (Markdown)

In text cells, the markup language Markdown can be used for formatting and LaTeX for mathematical equations.

> **Tip**
>
> Please check the `User Interface Tour` and `Keyboard Shortcuts` topics in the `Help` menu!

The cell currently being worked on can be in `command mode` or `edit mode`.

	Command mode	*Edit mode*
Activate mode	`ESC`	Double-click or `Enter` in cell
New cell	`b`	`Alt-Enter`
Delete cell	`dd`	
Save notebook	`s`	`Ctrl-s`
Rename notebook	`Menu → File → Rename`	`Menu → File → Rename`
Close notebook	`Menu → File → Close & Halt`	`Menu → File → Close & Halt`
Run cell	`Ctrl-Enter`	`Ctrl-Enter`
Run cell, move to next cell	`Shift-Enter`	`Shift-Enter`
Run cell, insert new cell below	`Alt-Enter`	`Alt-Enter`

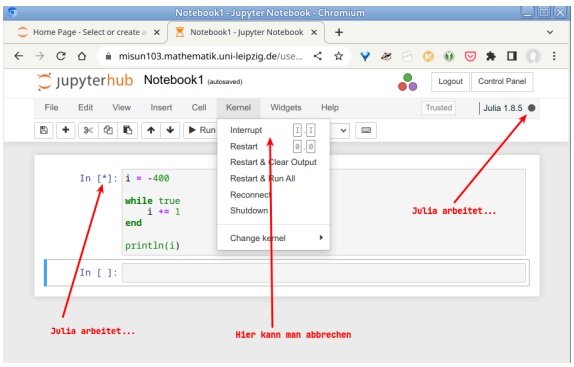


Figure 1.3: Julia working...

When a cell is working, its cell number becomes a `*` and the `kernel busy` indicator appears (solid black dot at the top right next to the Julia version). If this takes too long (an *infinite loop* can easily happen):

- click `Menu → Kernel → Interrupt`. If this doesn't work,
- click `Menu → Kernel → Restart`.

! Important

After a `kernel restart`, all cells containing the required definitions, `using` statements, etc. must be executed again.

At the end of each session, please always:

- `Menu → File → Save & Checkpoint`
- `Menu → File → Close & Halt`
- `Logout`

2. First Contact

This chapter helps you get started. It omits many details, and the code examples are often not optimal.

2.1 Julia as a Calculator

Compute the following: $12^{1/3} + \frac{3\sqrt{2}}{\sin(0.5) - \cos(\frac{\pi}{4})\log(3)} + e^5$

```
12^(1/3) + 3sqrt(2) / (sin(.5) - cos(pi/4)*log(3)) + exp(5)
```

```
136.43732662344087
```

Note that:

- Powers are written as `a^b`.
- The constant `π` is predefined.
- `log()` is the natural logarithm.
- The multiplication operator `*` can be omitted after a number when followed by a variable, function, or opening parenthesis.

2.2 The most important keys: `Tab` and `?`

When programming, press the Tab key as soon as you've typed 2–3 letters of a word. Potential completions are then displayed or completed if the completion is unique. This saves time and you learn a lot about Julia.

```
lo ▷ Tab
```

```
log
lock
log2
log1p
log10
local
logrange
lowercase
load_path
lowercasefirst
locate_package
```

```
pri ▷ Tab
```

```
print
println
printstyled
primitive type
```

Julia's built-in help system `?name` provides comprehensive documentation for all functions and constructs. Here is a relatively brief example:

```
?for
```

```
search: for nor xor foldr floor Core sort

for

for loops repeatedly evaluate a block of statements while iterating over a sequence of
values.
```

```
The iteration variable is always a new variable, even if a variable of the same name exists
in the enclosing scope. Use outer to reuse an existing local variable for iteration.

Examples
========

julia> for i in [1, 4, 0]
println(i)
end
1
4
0
```

2.3 Variables and Assignments

Variables are created through assignment with the assignment operator `=`.

```
x = 1 + sqrt(5)
y = x / 2
```

```
1.618033988749895
```

In interactive mode, Julia displays the result of the last statement.

> **Note**
>
> Assignments are not mathematical equations. The semantics of the assignment operator (the equals sign) is:
>
> - Compute the right side and
> - Assign the result to the left side.
>
> Expressions like `x + y = sin(2)` are therefore invalid. Only a variable name may appear on the left side.

2.4 Data types

Julia is a strongly typed language where every object has a type. Among the fundamental types are:

- Integers
- Floating-point numbers
- Strings
- Booleans.

The type of a variable can be determined using the `typeof()` function.

```
for x ∈ (42, 12.0, 3.3e4, "Hello!", true)
    println("x = ", x, " ... Type: ", typeof(x))
end
```

```
x = 42 … Type: Int64
x = 12.0 … Type: Float64
x = 33000.0 … Type: Float64
x = Hello! … Type: String
x = true … Type: Bool
```

The standard floating-point number has a size of 64 bits, which corresponds to a `double` in C/C++/Java.

Julia is a dynamically typed language. Variables have no type; they are typeless references (pointers) to typed objects. When people speak of the "type of a variable", they mean the type of the object currently assigned to the variable.

```
x = sqrt(2)

println( typeof(x), " - Value of x = $x" )
```

```
x = "Now I'm no longer a floating-point number!"

println( typeof(x), "  - Value of x = $x" )
```

```
Float64 - Value of x = 1.4142135623730951
String - Value of x = Now I'm no longer a floating-point number!
```

2.5 Print statements

The `println()` function adds a line break at the end; `print()` does not.

```
print(y)
print("...the line continues...")
print("still...")
println(y)
println("New line")
println("New line")
```

```
1.618033988749895…the line continues…still…1.618033988749895
New line
New line
```

Both functions can accept a list of string literals and variables as arguments. Variables can also be embedded in strings by prefixing the variable name with a dollar sign *(string interpolation)*.

```
x = 23
y = 3x + 5
zz = "Done!"
println("x= ", x, " ... and y= ", y, "...", zz)   # 1. variant
println("x= $x ... and y= $y...$zz")              # variant with string interpolation
```

```
x= 23 … and y= 74...Done!
x= 23 … and y= 74...Done!
```

If you want to print a dollar sign…

you must escape it with a *backslash*. To print a *backslash* itself, you must double it.

```
println("One dollar: 1\$ and three backslashes: \\\\\\ ")
```

```
One dollar: 1$ and three backslashes: \\\
```

2.6 Functions

Function definitions begin with the keyword `function` and end with the keyword `end`. Typically, they have one or more arguments and return a computed result via a `return` statement.

```
function hypotenuse(a, b)    # a bit cumbersome
    c2 = a^2 + b^2
    c  = sqrt(c2)
    return c
end
```

```
hypotenuse (generic function with 1 method)
```

After definition, the function can be used (called). The variables `a,b,c,c2` used in the definition are local and not available outside the function definition.

```
x = 3
z = hypotenuse(x, 4)
```

```
println("z = $z")
println("c = $c")
```

```
z = 5.0
UndefVarError: `c` not defined in `Main.Notebook`
Suggestion: check for spelling errors or missing imports.
Stacktrace:
 [1] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/first_contact.qmd:219
```

Very simple functions can also be defined as one-liners:

```
hypotenuse(a, b) = sqrt(a^2+b^2)
```

```
hypotenuse (generic function with 1 method)
```

2.7 Tests

Tests return a Boolean value (`true` or `false`).

```
x = 3^2
x < 2^3
```

```
false
```

In addition to the usual arithmetic comparisons `==, !=, <, <=, >, >=` there are many other tests. The result of a test can also be assigned to a variable, which is then of type `Bool`. The logical operators `&&`, `||` and negation `!` can be used in tests.

```
test1 = "Car" in ["Bicycle", "Car", "Train"]
test2 = x == 100 ||  !(x <= 30 && x > 8)
test3 = startswith("lampshade", "Lamp")
println("$test1 $test2 $test3")
```

```
true false false
```

2.8 Conditional Statements

A simple `if` statement has the form

```
if <test>
    <statement1>
    <statement2>
    ...
end
```

There can be one or more `elseif` blocks and an optional final `else` block.

```
x = sqrt(100)

if x > 20
    println("Strange!")
else
    println("OK")
    y = x + 3
end
```

```
OK
13.0
```

Indentation enhances readability but is optional. Line breaks separate statements, as do semicolons. The code above is equivalent to the following single line in Julia:

```
# Please don't program like this! You will regret it!
x=sqrt(100); if x > 20 println("Strange!") else println("OK"); y = x + 3 end
```

```
OK
13.0
```

2.9 Simple `for` loops

`for` loops for repeating the execution of statements have the form

```
for  <counter> = start:end
    <statement1>
    <statement2>
    ...
end
```

Example:

```
sum = 0
for i = 1:100
    sum = sum + i
end
sum
```

```
5050
```

2.10 Arrays

One-dimensional arrays (vectors) are a simple container type. They can be created with square brackets and accessed by index, with indexing starting at 1.

```
v = [12, 33.2,  17, 19, 22]
```

```
5-element Vector{Float64}:
12.0
33.2
17.0
19.0
22.0
```

```
typeof(v)
```

```
Vector{Float64} (alias for Array{Float64, 1})
```

```
v[1] = v[4] + 10
v
```

```
5-element Vector{Float64}:
29.0
33.2
17.0
19.0
22.0
```

Empty vectors can be created and extended with `push!()`.

```
v = []     # empty vector
push!(v, 42)
push!(v, 13)
v
```

```
2-element Vector{Any}:
42
13
```

Appendix: how the effect of the Tab key was simulated on this page

```
using REPL
function Tab(s)
    dc = map(x->x.name, REPL.doc_completions(s))
    l = filter(x->startswith(x,s), dc)
    println.(l)
    return   # return nothing, since broadcast println produces empty vector
end

▷ = ▷    # https://docs.julialang.org/en/v1/manual/functions/#Function-composition-and-piping

pri = "pri";
```

```
pri ▷ Tab
```

```
print
println
printstyled
primitive type
```

3. A Small Example Program

3.1 What Skills Are Needed for Programming?

- **Thinking in algorithms:**
  What steps are required to solve the problem? What data must be stored, what data structures must be created? What cases can occur and must be recognized and handled?
- **Implementing the algorithm in a program:**
  What data structures, constructs, functions… does the programming language provide?
- **Formal syntax:**
  Humans understand 'broken English'; computers don't understand 'broken C++' or 'broken Julia'. The syntax must be mastered.
- **The "ecosystem" of the language:**
  How do I run my code? How do the development environments work? What options does the compiler understand? How do I install additional libraries? How do I read error messages? Where can I get information?
- **The art of debugging:**
  Programming beginners are often happy when they have eliminated all syntax errors and the program finally 'runs'. For more complex programs, the work only just begins, as testing, finding, and fixing errors in the algorithm must now be done.
- **Intuition for the efficiency and complexity of algorithms**
- **Specifics of computer arithmetic**, especially floating point numbers

These cannot all be mastered at once. Be patient with yourself and 'play' with the language.
The following examples should serve as inspiration.

3.2 Project Euler

A nice source for programming tasks with mathematical character and very different levels of difficulty is Project Euler. The tasks are designed so that no inputs are necessary and the desired result is a single number. This allows you to fully concentrate on programming the algorithm.

3.2.1 Example 1

> Project Euler Problem No. 1
>
> If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.
> Find the sum of all the multiples of 3 or 5 below 1000.

1. Algorithm:
   - Test all natural numbers <1000 for divisibility by 3 or 5 and
   - sum the divisible numbers.
2. Implementation:

   How does Julia provide the remainder of integer division? Functions like this are typically called `rem()` (for *remainder*) or `mod()`. You can look it up in the documentation or try `?rem` and `?mod` in the Julia REPL. Both functions exist; the difference lies in the treatment of negative integers. For natural numbers `n,m`, `mod(n,m)` and `rem(n,m)` give the same result, and the latter also has the infix form `n % m`.

   How does one test a sequence of values and sum them up? There is a standard pattern: `for` loop and accumulator variable:

One possible solution:

```
s = 0                                # initialize accumulator variable
for i in 1:999                       # loop
    if i % 3 == 0 || i % 5 == 0      # test
        s += i                       # add to accumulator
    end                              # end test
end                                  # end loop
println("The answer is: $s")         # output result
```

```
The answer is: 233168
```

Of course, this can also be done a bit shorter:

Another solution:

```
sum([i for i in 1:999 if i%3==0||i%5==0])
```

```
233168
```

3.2.2 Example 2

Project Euler Problem No. 92

A number chain is created by continuously adding the square of the digits in a number to form a new number until it has been seen before.

For example,

44 → 32 → 13 → 10 → **1** → **1**

85 → **89** → 145 → 42 → 20 → 4 → 16 → 37 → 58 → **89**

Therefore any chain that arrives at 1 or 89 will become stuck in an endless loop. What is most amazing is that EVERY starting number will eventually arrive at 1 or 89.

How many starting numbers below ten million will arrive at 89?

Programs can be developed *top-down* and *bottom-up*. *Top-down* would probably mean here: "We start with a loop `for i = 1:9999999`." The other approach works through individually testable components and is often more effective. (And with a certain project size, one approaches the goal from both 'top' and 'bottom' simultaneously.)

Function No. 1

We want to investigate how numbers behave under repeated calculation of the 'square digit sum' (sum of squares of digits). Therefore, write and test a function that calculates this 'square digit sum'.

`q2sum(n)` calculates the sum of the squares of the digits of n in base 10:

```
function q2sum(n)
    s = 0                          # accumulator for the sum
    while n > 9                    # as long as n is still multi-digit...
        q, r = divrem(n, 10)         # calculate integer quotient and remainder of
division by 10
        s += r^2                   # add squared digit to accumulator
        n = q                      # continue with the number divided by 10
    end
    s += n^2                       # add the square of the last digit
    return s
end
```

Let's test it now:

```
for i in [1, 7, 33, 111, 321, 1000, 73201]
   j = q2sum(i)
   println("Square digit sum of $i = $j")
end
```

```
Square digit sum of 1 = 1
Square digit sum of 7 = 49
Square digit sum of 33 = 18
Square digit sum of 111 = 3
Square digit sum of 321 = 14
Square digit sum of 1000 = 1
Square digit sum of 73201 = 63
```

In the spirit of the task, we apply the function repeatedly:

```
n = 129956799
for i in 1:14
    n = q2sum(n)
    println(n)
end
```

```
439
106
37
58
89
145
42
20
4
16
37
58
89
145
```

... and we have now hit one of the '89-cycles'.

Function No. 2

We need to test whether repeated application of the function `q2sum()` finally leads to **1** or ends in this **89**-cycle.

`q2test(n)` determines whether repeated square digit sum calculation leads to the 89-cycle:

```
function q2test(n)
    while n !=1 && n != 89       # as long as we have not reached 1 or 89...
        n = q2sum(n)             # repeat
    end
    if n == 1 return false end   # not an 89 cycle
    return true                  # 89 cycle found
end
```

... and with this, we can solve the task:

```
c = 0                          # once again an accumulator
for i = 1 : 10_000_000 - 1   # this is how you can separate thousands for better readability
    if q2test(i)               # q2test() returns a boolean, no further test needed
      c += 1             #    'if x == true' is redundant, 'if x' is perfectly sufficient
    end
end
println("$c numbers below ten million arrive at 89.")
```

```
8581146 numbers below ten million arrive at 89.
```

Numbers for which the iterative square digit sum calculation ends at 1 are called happy numbers; all other numbers eventually reach the 89-cycle.

Here is the complete program again:

Our solution to Project Euler No 92:

```
function q2sum(n)
    s = 0
    while n > 9
       q, r = divrem(n, 10)
       s += r^2
       n = q
    end
    s += n^2
    return s
end

function q2test(n)
    while n !=1 && n != 89
        n = q2sum(n)
    end
    if n==1 return false end
    return true
end

c = 0
for i = 1 : 10_000_000 - 1
    if q2test(i)
        c += 1
    end
end
println("$c numbers below ten million arrive at 89.")
```

```
8581146 numbers below ten million arrive at 89.
```

4. Fundamentals of Syntax

4.1 Names of Variables, Functions, Types, etc.

- Names may contain letters, digits, underscores `_`, and exclamation marks `!`.
- The first character must be a letter or an underscore.
- Case is significant: `Nmax` and `NMAX` are different variables.
- The character set used is Unicode, which covers over 150 scripts and numerous symbols.
- There is a short list of reserved keywords: `if, then, function, true, false,...`

Example

Permissible: `i, x, Ω, x2, TheUnknownNumber, new_Value, 🎷, Counter_2, лічильник, yes!!!!,...`
Impermissible: `Karen's_Funktion, 3achsen, A#B, $this_is_not_Perl, true,...`

Note:

In addition to the *reserved keywords* of the core language, numerous additional functions and objects are predefined, such as the mathematical functions `sqrt(), log(), sin()`. These definitions are found in the `Base` module, which Julia loads automatically on startup. Names from `Base` can be redefined as long as they have not yet been used:

```
log = 3
1 + log
```

```
4
```

Now, of course, `log()` no longer works:

```
x = log(10)
```

```
MethodError: objects of type Int64 are not callable
The object of type `Int64` exists, but no method is defined for this combination of argument
types when trying to treat it as a callable object.
Maybe you forgot to use an operator such as \*, ^, %, / etc. ?
Stacktrace:
 [1] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/syntax.qmd:49
```

(see also https://stackoverflow.com/questions/65902105/how-to-reset-any-function-in-julia-to-its-original-state)

4.2 Statements

- Usually, each line contains one statement.
- If a statement is recognizable as incomplete at the end of a line through
  - open parentheses
  - operators,

  then the next line is regarded as a continuation.
- Multiple statements per line can be separated by semicolons.
- In interactive mode (REPL or notebook), a semicolon after the last statement suppresses the output of the result of that statement.

Example:

In interactive mode, the value of the last statement is also displayed without explicit `print()`:

```
println("Hallo 🌍!")
x = sum([i^2 for  i=1:10])
```

```
Hallo 🌍!
385
```

The semicolon suppresses this:

```
println("Hallo 🌍!")
x = sum([i^2 for  i=1:10]);
```

```
Hallo 🌍!
```

Warning

For multi-line statements, a continued line should end with an open operator or parenthesis.

```
x = sin(π/2) +
   3 * cos(0)
```

```
4.0
```

Therefore, the following fails, but—unfortunately—**without an error message**!

```
x = sin(π/2)
  + 3 * cos(0)
println(x)
```

```
1.0
```

Here, the `+` in the second line is interpreted as a prefix operator (sign). Thus, lines 1 and 2 are each complete, correct expressions on their own (even though line 2 is of course completely useless) and are processed as such.

Moral: If you want to split longer expressions across multiple lines, you should always open a parenthesis. Then it doesn't matter where the line break occurs.

```
x = ( sin(π/2)
   + 3 * cos(0) )
println(x)
```

```
4.0
```

4.3 Comments

Julia knows two types of comments in program text:

```
# Single-line comments begin with a hash symbol.

x = 2    # everything from '#' to the end of the line is a comment and is ignored. x = 3
```

```
2
```

```
#=
Single and multi-line comments can be enclosed in `#= ...  =#`. Nested comments are
possible.
`#=`
```

```
i.e., unlike in C/C++/Java, the comment does not end with the first comment-end character,
but the `#=...=#` pairs act like parentheses.
`=#`
   The automatic 'syntax highlighter' doesn't support this yet, as the alternating
   gray shading of this comment shows.
=#


x #= this is a really rare name for a variable! =#  = 3
```

```
3
```

4.4 Data Types Part I

- Julia is a strongly typed language. All objects have a type. Functions and operations expect arguments of the correct type.
- Julia is a dynamically typed language. Variables have no type. They are names that can be bound to objects via assignment `x = ...`.
- When speaking of the "type of a variable", one means the type of the object currently assigned to the variable.
- Functions and operators can implement different *methods* for different argument types.
- Depending on the concrete argument types, it is decided at call time which method is selected (*dynamic dispatch*).

Simple basic types are, for example:

```
Int64, Float64, String, Char, Bool
```

```
x = 2
x, typeof(x), sizeof(x)
```

```
(2, Int64, 8)
```

```
x = 0.2
x, typeof(x), sizeof(x)
```

```
(0.2, Float64, 8)
```

```
x = "Hello!"
x, typeof(x), sizeof(x)
```

```
("Hello!", String, 6)
```

```
x = 'Ω'
x, typeof(x), sizeof(x)
```

```
('Ω', Char, 4)
```

```
x = 3 > π
x, typeof(x), sizeof(x)
```

```
(false, Bool, 1)
```

- `sizeof()` returns the size of an object or type in bytes (1 byte = 8 bits)
- 64-bit integers and 64-bit floating-point numbers correspond to the instruction set of modern processors and are therefore the standard numeric types.
- Character literals like `'A'` and single-character strings like `"A"` are different objects.

4.5 Control Flow

4.5.1 `if` Blocks

- An `if` block can contain any number of `elseif` branches and, at the end, at most one `else` branch.
- The block has a value: the value of the last executed statement.

```
x = 33
y = 44
z = 34

if x < y && z != x                    # elseif or else branches are optional
    println("yes")
    x += 10
elseif x < z                          # any number of elseif branches
    println(" x is smaller than z")
elseif x == z+1
    println(" x is successor of z")
else                                  # at most one else block
    println("All wrong")
end                                   # value of the entire block is the value of the
                                      # last evaluated statement
```

```
yes
43
```

Short blocks can be written on one line:

```
if x > 10 println("x is larger than 10") end
```

```
x is larger than 10
```

The value of an `if` block can be assigned:

```
y = 33
z = if y > 10
       println("y is larger than 10")
       y += 1
    end
z
```

```
y is larger than 10
34
```

4.5.2 Conditional Operator (ternary operator) `test ? exp1 : exp2`

```
x = 20
y = 15
z = x < y ? x+1 : y+1
```

```
16
```

is equivalent to

```
z = if x < y
        x+1
    else
        y+1
    end
```

```
16
```

4.6 Comparisons, Tests, Logical Operations

4.6.1 Arithmetic Comparisons

- `==`
- `!=`, `≠`
- `>`
- `>=`, `≥`
- `<`
- `<=`, `≤`

As usual, the equality test `==` must be distinguished from the assignment operator `=`. Almost anything can be compared.

```
"Aachen" < "Leipzig", 10 ≤ 10.01, [3,4,5] < [3,6,2]
```

```
(true, true, true)
```

Well, almost anything:

```
3 < "four"
```

```
MethodError: no method matching isless(::Int64, ::String)
The function `isless` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  isless(::Missing, ::Any)
   @ Base missing.jl:87
  isless(::Integer, ::Base.CoreLogging.LogLevel)
   @ Base logging/logging.jl:133
  isless(::Real, ::AbstractFloat)
   @ Base operators.jl:223
  ...

Stacktrace:
 [1] <(x::Int64, y::String)
   @ Base ./operators.jl:399
 [2] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/syntax.qmd:261
```

The error message shows a few fundamental principles of Julia:

- Operators are also just functions: `x < y` becomes the function call `isless(x, y)`.
- Functions (and thus operators) can implement different *methods* for different argument types.
- Depending on the concrete argument types, it is decided at function call which method is used (*dynamic dispatch*).

One can display all methods for a function. This provides insight into Julia's complex type system:

```
methods(<)
```

```
# 74 methods for generic function "<" from Base:
  [1] <(x::Int128, y::Float64)
     @ float.jl:665
  [2] <(x::Int128, y::Float32)
     @ float.jl:665
  [3] <(x::Bool, y::Bool)
     @ bool.jl:160
  [4] <(x::Float64, y::UInt128)
     @ float.jl:674
  [5] <(x::Float64, y::Int128)
     @ float.jl:674
  [6] <(x::Float64, y::UInt64)
     @ float.jl:674
  [7] <(x::Float64, y::Int64)
     @ float.jl:674
  [8] <(x::Float64, y::AbstractIrrational)
     @ irrationals.jl:128
  [9] <(::Missing, ::Missing)
```

```
    @ missing.jl:83
[10] <(::Missing, ::Any)
    @ missing.jl:84
[11] <(x::BigFloat, y::BigFloat)
    @ mpfr.jl:993
[12] <(x::BigFloat, y::AbstractIrrational)
    @ irrationals.jl:136
[13] <(x::BigFloat, y::Union{Float16, Float32, Float64})
    @ mpfr.jl:1025
[14] <(x::BigFloat, y::Integer)
    @ mpfr.jl:1023
[15] <(x::Rational{BigInt}, y::AbstractIrrational)
    @ irrationals.jl:175
[16] <(x::BigInt, y::BigInt)
    @ gmp.jl:731
[17] <(x::BigInt, f::Union{Float16, Float32, Float64})
    @ gmp.jl:734
[18] <(x::BigInt, i::Integer)
    @ gmp.jl:732
[19] <(a::Base.BinaryPlatforms.CPUID.ISA, b::Base.BinaryPlatforms.CPUID.ISA)
    @ cpuid.jl:21
[20] <(x::Float16, y::AbstractIrrational)
    @ irrationals.jl:132
[21] <(x::Float16, y::Union{Int16, UInt16})
    @ float.jl:693
[22] <(x::Float16, y::Union{Int128, Int64, UInt128, UInt64})
    @ float.jl:687
[23] <(x::UInt64, y::Float64)
    @ float.jl:665
[24] <(x::UInt64, y::Float32)
    @ float.jl:665
[25] <(x::Float32, y::UInt128)
    @ float.jl:674
[26] <(x::Float32, y::Int128)
    @ float.jl:674
[27] <(x::Float32, y::UInt64)
    @ float.jl:674
[28] <(x::Float32, y::Int64)
    @ float.jl:674
[29] <(x::Float32, y::AbstractIrrational)
    @ irrationals.jl:130
[30] <(::Tuple{}, ::Tuple{})
    @ tuple.jl:589
[31] <(::Tuple{}, ::Tuple)
    @ tuple.jl:590
[32] <(x::Int64, y::Float64)
    @ float.jl:665
[33] <(x::Int64, y::Float32)
    @ float.jl:665
[34] <(x::UInt128, y::Float64)
    @ float.jl:665
[35] <(x::UInt128, y::Float32)
    @ float.jl:665
[36] <(::Irrational{s}, ::Irrational{s}) where s
    @ irrationals.jl:113
[37] <(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8}
    @ int.jl:83
[38] <(x::T, y::T) where T<:Union{UInt128, UInt16, UInt32, UInt64, UInt8}
    @ int.jl:519
[39] <(x::T, y::T) where T<:Union{Float16, Float32, Float64}
    @ float.jl:623
[40] <(x::Rational, y::Rational)
    @ rational.jl:425
[41] <(i::Integer, x::BigInt)
    @ gmp.jl:733
[42] <(x::AbstractIrrational, y::AbstractIrrational)
    @ irrationals.jl:114
[43] <(x::T, y::T) where T<:Real
    @ promotion.jl:638
[44] <(x::Union{Int128, Int16, Int32, Int64, Int8}, y::Union{UInt128, UInt16, UInt32, UInt64,
```

```
UInt8})
     @ int.jl:525
 [45] <(x::Union{UInt128, UInt16, UInt32, UInt64, UInt8}, y::Union{Int128, Int16, Int32, Int64,
Int8})
     @ int.jl:526
 [46] <(x::Union{Int128, Int64, UInt128, UInt64}, y::Float16)
     @ float.jl:688
 [47] <(x::Union{Float16, Float32}, y::Union{Int32, UInt32})
     @ float.jl:690
 [48] <(x::Union{Int32, UInt32}, y::Union{Float16, Float32})
     @ float.jl:691
 [49] <(x::Union{Int16, UInt16}, y::Float16)
     @ float.jl:694
 [50] <(x::Rational, y::Integer)
     @ rational.jl:433
 [51] <(x::Integer, y::Rational)
     @ rational.jl:434
 [52] <(x::Rational, y::AbstractFloat)
     @ rational.jl:450
 [53] <(x::AbstractFloat, y::Rational)
     @ rational.jl:450
 [54] <(f::Union{Float16, Float32, Float64}, x::BigInt)
     @ gmp.jl:735
 [55] <(x::Integer, y::BigFloat)
     @ mpfr.jl:1024
 [56] <(x::Union{Float16, Float32, Float64}, y::BigFloat)
     @ mpfr.jl:1026
 [57] <(x::AbstractIrrational, y::Float64)
     @ irrationals.jl:127
 [58] <(x::AbstractIrrational, y::Float32)
     @ irrationals.jl:129
 [59] <(x::AbstractIrrational, y::Float16)
     @ irrationals.jl:131
 [60] <(x::AbstractIrrational, y::BigFloat)
     @ irrationals.jl:133
 [61] <(x::AbstractIrrational, y::Rational{BigInt})
     @ irrationals.jl:174
 [62] <(x::AbstractIrrational, y::Rational{T}) where T
     @ irrationals.jl:156
 [63] <(x::Rational{T}, y::AbstractIrrational) where T
     @ irrationals.jl:165
 [64] <(x::Real, y::Real)
     @ promotion.jl:488
 [65] <(x::Ptr, y::Ptr)
     @ pointer.jl:311
 [66] <(a::AbstractSet, b::AbstractSet)
     @ abstractset.jl:619
 [67] <(a::NamedTuple{n}, b::NamedTuple{n}) where n
     @ namedtuple.jl:252
 [68] <(x::Base.TwicePrecision{T}, y::Base.TwicePrecision{T}) where T
     @ twiceprecision.jl:785
 [69] <(::Tuple, ::Tuple{})
     @ tuple.jl:591
 [70] <(t1::Tuple{Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any,
Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Vararg{Any, N}}
where N, t2::Tuple{Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any,
Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Vararg{Any, N}}
where N)
     @ tuple.jl:602
 [71] <(t1::Tuple, t2::Tuple)
     @ tuple.jl:592
 [72] <(::Any, ::Missing)
     @ missing.jl:85
 [73] <(x)
     @ operators.jl:1292
 [74] <(x, y)
     @ operators.jl:399
```

Finally: comparisons can be chained.

```
10 < x ≤ 100  # this is equivalent to
              #   10 < x && x ≤ 100
```

```
true
```

4.6.2 Tests

Some functions of type `f(c::Char) → Bool`

```
isnumeric('a'), isnumeric('7'), isletter('a')
```

```
(false, true, true)
```

and of type `f(s1::String, s2::String) → Bool`

```
contains("Lampenschirm",      "pensch"),      startswith("Lampenschirm",      "Lamb"),
endswith("Lampenschirm", "rm")
```

```
(true, false, true)
```

- The function `in(item, collection) → Bool` tests whether `item` is in `collection`.
- It also has the alias `∈(item, collection)` and
- both `in` and `∈` can also be written as infix operators.

```
x = 3
x in [1, 2, 3, 4, 5]
```

```
true
```

```
x ∈ [1, 2, 33, 4, 5]
```

```
false
```

4.6.3 Logical Operations: `&&`, `||`, `!`

```
3 < 4 && !(2 > 8) && !contains("aaa", "b")
```

```
true
```

4.6.3.1 Conditional Evaluation (*short-circuit evaluation*)

- In `a && b`, `b` is only evaluated if `a == true`
- In `a || b`, `b` is only evaluated if `a == false`

(i) Thus, `if test statement end` can also be written as `test && statement`.

(ii) Thus, `if !test statement end` can be written as `test || statement`.
As an example[1] here is an implementation of the factorial function:

```
function fact(n::Int)
    n >= 0 || error("n must be non-negative")
    n == 0 && return 1
    n * fact(n-1)
end

fact(5)
```

```
120
```

Of course, all these tests can also be assigned to variables of type `Bool` and these variables can be used as tests in `if` and `while` blocks:

[1] from the Julia documentation

```
x = 3 < 4
y = 5 ∈ [1, 2, 5, 7]
z = x && y
if z                        # equivalent to:  if 3 < 4  &&  5 in [1,2,5,7]
    println("All correct!")
end
```

```
All correct!
```

- In Julia, all tests in a logical expression must be of type `Bool`.
- There is no implicit conversion such as *"0 is false and 1 (or anything != 0) is true"*
- If `x` is a numeric type, then the C idiom `if(x)` must be written as `if x != 0`.
- There is an exception to support the *short circuit evaluation*:
  - in the constructs `a && b && c...` or `a || b || c...` the last subexpression does not need to be of type `Bool` if these constructs are not used as tests in `if` or `while`:

```
z = 3 < 4 && 10 < 5 &&  sqrt(3^3)
z, typeof(z)
```

```
(false, Bool)
```

```
z = 3 < 4 && 10 < 50 &&  sqrt(3^3)
z, typeof(z)
```

```
(5.196152422706632, Float64)
```

4.7 Loops

4.7.1 The `while` loop

Syntax:

```
while *condition*
*loop body*
end
```

A series of statements (the loop body) is repeatedly executed as long as a condition is satisfied.

```
i = 1          # typically the test of the
               # while loop needs preparation ...
while i < 10
   println(i)
   i += 2      # ... and an update
end
```

```
1
3
5
7
9
```

The body of a `while` and `for` loop can contain the statements `break` and `continue`. `break` stops the loop, `continue` skips the rest of the loop body and immediately starts the next loop iteration.

```
i = 0

while i<10
   i += 1

   if i == 3
      continue     # start next iteration immediately,
   end             # skip rest of loop body

   println("i = $i")
```

```
   if i ≥ 5
      break        # break loop
   end
end

println("Done!")
```

```
i = 1
i = 2
i = 4
i = 5
Done!
```

With `break` one can also exit infinite loops:

```
i = 1

while true
    println(2^i)
    i += 1
    if i > 8 break end
end
```

```
2
4
8
16
32
64
128
256
```

4.7.2 `for` Loops

Syntax:

```
for *var* in *iterable container*
*loop body*
end
```

The loop body is executed for all items from a container.

Instead of `in`, ∈ can always be used. In the header of a `for` loop, `=` can also be used.

```
for i ∈ ["Mother", "Father", "Daughter"]
    println(i)
end
```

```
Mother
Father
Daughter
```

A numerical loop counter is often needed. For this purpose, we have the `range` construct. The simplest forms are `Start:End` and `Start:Step:End`.

```
end_value = 5

for i ∈ 1:end_value
    println(i^2)
end
```

```
1
4
9
16
25
```

```
for i = 1:5.5  print(" $i") end
```

```
1.0 2.0 3.0 4.0 5.0
```

```
for i = 1:2:14 print("  $i") end
```

```
1 3 5 7 9 11 13
```

```
for k = 14 : -2.5 : 1 print("   $k") end
```

```
14.0 11.5 9.0 6.5 4.0 1.5
```

4.7.2.1 Nested Loops

A `break` ends the innermost loop.

```
for i = 1:3
    for j = 1:3
        println( (i,j) )
        if j == 2
            break
        end
    end
end
```

```
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

Nested loops can also be combined in a single `for` statement. Then a `break` ends the entire loop.

```
for i = 1:3, j=1:3     # essentially the same as above, but:
    println( (i,j) )
    if j == 2
        break          # break ends the entire loop here
    end
end
```

```
(1, 1)
(1, 2)
```

Important: The semantics are completely different from C-style `for` loops!

In each loop iteration, the loop variable is re-initialized with the next element from the container.

```
for i = 1:5
    print(i," ... ")
    i += 2
    println(i)
end
```

```
1 … 3
2 … 4
3 … 5
4 … 6
5 … 7
```

The C semantics of `for(i=1; i<5; i++)` corresponds to the `while` loop:

```
i = 1
while i<5
   *loop body*   # here one can also mess with i effectively
   i += 1
end
```

4.8 Unicode

Julia uses Unicode as its character set. This allows identifiers in non-Latin scripts (e.g., Cyrillic, Korean, Sanskrit, runes, emojis,...) to be used for variables, functions, etc. The question of how one can enter such characters in their editor and whether the used screen font can display them is not Julia's problem.

- Some Unicode characters, e.g., `≤, ≠, ≥, π, ∈, √`, can be used instead of `<=, !=, >=, pi, in, sqrt`.
- Over 3000 Unicode characters can be entered in Julia in a LaTeX-like manner using tab completion.
  - `\alpha<TAB>` becomes α,
  - `\euler<TAB>` becomes *e* (Euler's number `exp(1)`, special script e, `U+0212F`)
  - `\le<TAB>` becomes ≤,
  - `\in<TAB>` becomes ∈,
  - `\:rainbow:<TAB>` becomes 🌈
    Here is the list.

4.9 Idiosyncrasies and Pitfalls of Syntax

- After a numeric constant, the multiplication operator `*` can be omitted when a variable, function, or opening parenthesis follows.

```
z = 3.4x + 2(x+y) + xy
```

is therefore valid Julia. Note, however, that the term `xy` is interpreted as a single variable named xy **and not** as the product of x and y!

- This rule has a few pitfalls:

This works as expected:

```
e = 7
3e
```

```
21
```

Here, the input is interpreted as a floating-point number – and `3E+2` or `3f+2` (Float32) as well.

```
3e+2
```

```
300.0
```

A space creates clarity:

```
3e + 2
```

```
23
```

This works:

```
x = 4
3x + 3
```

```
15
```

...and this does not. `0x`, `0o`, `0b` are interpreted as the beginning of a hexadecimal, octal, or binary constant.

```
3y + 0x
```

```
ParseError:
# Error @ /home/hellmund/Julia/Book26/JuliaBook/chapters/syntax.qmd:580:6
3y + 0x
#    └┘ ── invalid numeric constant
Stacktrace:
 [1] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/syntax.qmd:580
```

- There are a few other cases where the very permissive syntax leads to surprises.

```
Important  = 21
Important! = 42       # identifiers can also contain !
(Important, Important!)
```

```
(21, 42)
```

```
Important!=88
```

```
true
```

Julia interprets this as the *comparison* `Important != 88`.
Again, spaces around operators help:

```
Important! = 88
Important!
```

```
88
```

- Operators of the form `.*`, `.+`,... have a special meaning in Julia (*broadcasting*, i.e., vectorized operations).

```
1.+2.
```

```
ParseError:
# Error @ /home/hellmund/Julia/Book26/JuliaBook/chapters/syntax.qmd:612:1
1.+2.
└┘ ── ambiguous '.' syntax; add whitespace to clarify (eg `1.+2` might be `1.0+2` or `1 .+ 2`)
Stacktrace:
 [1] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/syntax.qmd:612
```

Again, spaces create clarity!

```
1. + 2.
```

```
3.0
```

5. Working with Julia: The REPL, Packages, and Introspection

5.1 Official Documentation

The official Julia documentation https://docs.julialang.org/ contains several overviews, including:

- https://docs.julialang.org/en/v1/base/punctuation/ List of symbols
- https://docs.julialang.org/en/v1/manual/unicode-input/ List of special Unicode symbols and their input methods via tab completion in Julia
- https://docs.julialang.org/en/v1/manual/mathematical-operations/#Rounding-functions List of mathematical functions

5.2 Julia REPL (Read - Eval - Print - Loop)

After starting Julia in a terminal, you can enter both Julia code and various commands:

Command	Action
`exit()` or `Ctrl-d`	exit Julia
`Ctrl-c`	interrupt
`Ctrl-l`	clear screen
End command with `;`	suppress output
`include("filename.jl")`	read and execute file with Julia code

The REPL supports several modes:

Mode	Prompt	Start mode	Exit mode
default	`julia>`		`Ctrl-d` (exits Julia)
Package manager	`pkg>`	`]`	`backspace`
Help	`help?>`	`?`	`backspace`
Shell	`shell>`	`;`	`backspace`

5.3 Jupyter Notebooks (IJulia)

In a Jupyter notebook, the modes are usable as single-line commands in their own input cells:
(i) a package manager command:

```
] status
```

(ii) a help query:

```
?sin
```

(iii) a shell command:

```
;ls
```

5.4 The Package Manager

An important part of the *Julia ecosystem* is the extensive collection of packages that extend Julia's functionality.

- Some packages are part of every Julia installation and only need to be activated with `using Packagename`.
  - They form the so-called *standard library*, which includes
  - `LinearAlgebra`, `Statistics`, `SparseArrays`, `Printf`, `Pkg`, and others.
- Over 10000 packages are officially registered, see https://julialang.org/packages/.
  - These can be downloaded and installed with just a few keystrokes using the *package manager* `Pkg`.
  - `Pkg` can be called in two ways:
    - as normal Julia statements that can also be in a `.jl` program file:

```
using Pkg
Pkg.add("PackageXY")
```

    - in the special pkg-mode of the Julia REPL:

```
] add PackageXY
```

  - Afterward, the package can be used with `using PackageXY`.
- You can also install packages from other sources and self-written packages.

5.4.1 Some Package Manager Functions

Function	`pkg` - Mode	Explanation
`Pkg.add("PackageXY")`	`pkg> add PackageXY`	add to current environment
`Pkg.rm("PackageXY")`	`pkg> remove PackageXY`	remove from current environment
`Pkg.update()`	`pkg> update`	update packages in current environment
`Pkg.activate("mydir")`	`pkg> activate mydir`	activate directory as current environment
`Pkg.status()`	`pkg> status`	list packages
`Pkg.instantiate()`	`pkg> instantiate`	install all packages according to `Project.toml`

5.4.2 Installed Packages and Environments

- Julia's package manager maintains:
  1. a list of packages explicitly installed with the command `Pkg.add()` or `]add` with exact version specifications in a file `Project.toml` and
  2. a list of all packages installed as implicit dependencies in the file `Manifest.toml`.
- The directory in which these files are located is the `environment` and is displayed with `Pkg.status()` or `]status`.
- Without an expplicit project environment, this looks as follows:

```
(@v1.10) pkg> status
Status ~\.julia\environments\v1.12\Project.toml
[6e4b80f9] BenchmarkTools v1.6.3
[5fb14364] OhMyREPL v0.5.29
[91a5bcdd] JuliaFormatter v2.3.0
[295af30f] Revise v3.13.2
```

- You can use separate `environments` for different projects. You can either start Julia with

```
julia --project=path/to/myproject
```

or activate the environment in Julia with `Pkg.activate("path/to/myproject")`. Then `Project.toml` and `Manifest.toml` files are created and managed there. (The installation of package files still takes place somewhere under `$HOME/.julia`)

5.4.3 Installing Packages on our Jupyter Server `misun103`:

- A central repository already contains all packages mentioned in this course.
- You have no write permissions there.
- However, you can install additional packages in your `HOME`. As a first command, you need to activate the current directory:

```
] activate .
```

(Note the dot!)

After that, you can install packages with `add` in the pkg-mode:

```
] add PackageXY
```

Package installation can take a significant amount of time. Many packages have complex dependencies and trigger the installation of further packages. Many packages are precompiled during installation. You can see the installation progress in the REPL, but unfortunately not in the Jupyter notebook.

5.5 The Julia JIT *(just in time)* Compiler: Introspection

The Julia compiler is based on the *LLVM Compiler Infrastructure Project.*
Stages of Compilation

stage & result	introspection command
Parse ⇒ Abstract Syntax Tree (AST)	`Meta.parse()`
Lowering: transform AST ⇒ Static Single Assignment (SSA) form	`@code_lowered`
Type Inference	`@code_warntype`, `@code_typed`
Generate LLVM intermediate representation	`@code_llvm`
Generate native machine code	`@code_native`

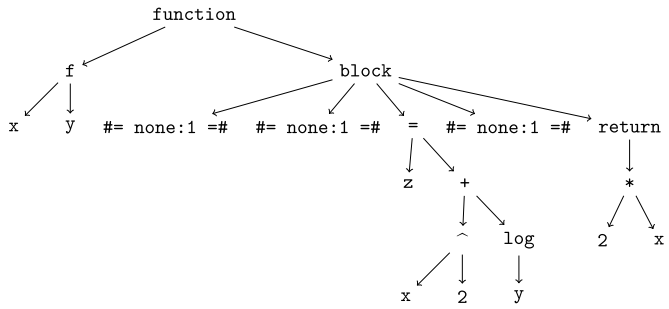
```
function f(x,y)
    z = x^2 + log(y)
    return 2z
end
```

```
f (generic function with 1 method)
```

```
p = Meta.parse( "function f(x,y); z=x^2+log(y); return 2x; end")
```

```
:(function f(x, y)
      #= none:1 =#
      #= none:1 =#
      z = x ^ 2 + log(y)
      #= none:1 =#
      return 2x
  end)
```

```
using TreeView

walk_tree(p)
```




```
Error showing value of type TreeView.LabelledTree
StackOverflowError:
Stacktrace:
 [1] show(io::IOContext{IOBuffer}, x::LabelledTree) (repeats 79983 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:26
```

```
@code_lowered f(2,4)
```

```
CodeInfo(
1 — %1  = Main.Notebook.:+
│   %2  = Main.Notebook.:^
│   %3  =   builtin Core.apply_type(Base.Val, 2)
│   %4  =   dynamic (%3)()
│   %5  =   dynamic Base.literal_pow(%2, x, %4)
│   %6  = Main.Notebook.log
│   %7  =   dynamic (%6)(y)
│         z = (%1)(%5, %7)
│   %9  = Main.Notebook.:\*
│   %10 = z
│   %11 =   dynamic (%9)(2, %10)
└──       return %11
)
```

```
@code_warntype f(2,4)
```

```
MethodInstance for Main.Notebook.f(::Int64, ::Int64)
  from f(x, y) @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:222
Arguments
  #self#Warning: detected a stack overflow; program state may be corrupted, so further execution
might be unreliable.
StackOverflowError:
Stacktrace:
 [1] show(io::IOBuffer, x::Core.Const) (repeats 79984 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:26
Error showing value of type StackOverflowError
StackOverflowError:
Stacktrace:
 [1] show(io::IOContext{IOBuffer}, x::StackOverflowError) (repeats 79983 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:26
```

```
@code_typed f(2,4)
```

```
CodeInfo(
1 — %1 = intrinsic Base.mul_int(x, x)::Int64
│   %2 = intrinsic Base.sitofp(Float64, y)::Float64
│   %3 =    invoke Base.Math.log(%2::Float64)::Float64
│   %4 = intrinsic Base.sitofp(Float64, %1)::Float64
│   %5 = intrinsic Base.add_float(%4, %3)::Float64
│   %6 = intrinsic Base.mul_float(2.0, %5)::Float64
└──      return %6
) ⇒ Float64
```

```
@code_llvm f(2,4)
```

```
; Function Signature: f(Int64, Int64)
;  @ /home/hellmund/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:222 within `f`
define double @julia_f_11726(i64 signext %"x::Int64", i64 signext %"y::Int64") #0 {
top:
;  @ /home/hellmund/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:223 within `f`
; ┌ @ intfuncs.jl:437 within `literal_pow`
; │┌ @ int.jl:88 within `*`
   %0 = mul i64 %"x::Int64", %"x::Int64"
; └└
; ┌ @ math.jl:1544 within `log`
; │┌ @ float.jl:378 within `float`
; ││┌ @ float.jl:352 within `AbstractFloat`
; │││┌ @ float.jl:245 within `Float64`
     %1 = sitofp i64 %"y::Int64" to double
; │└└└
; │ @ math.jl:1546 within `log`
   %2 = call double @j_log_11729(double %1)
; └
; ┌ @ promotion.jl:433 within `+`
; │┌ @ promotion.jl:404 within `promote`
; ││┌ @ promotion.jl:379 within `_promote`
; │││┌ @ number.jl:7 within `convert`
; ││││┌ @ float.jl:245 within `Float64`
       %3 = sitofp i64 %0 to double
; │└└└└
; │ @ promotion.jl:433 within `+` @ float.jl:495
   %4 = fadd double %2, %3
; └
;  @ /home/hellmund/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:224 within `f`
; ┌ @ promotion.jl:434 within `*` @ float.jl:497
   %5 = fmul double %4, 2.000000e+00
; └
  ret double %5
}
```

```
@code_native f(2,4)
```

```
    .text
    .file   "f"
    .section    .ltext,"axl",@progbits
    .globl  julia_f_11843                   # -- Begin function julia_f_11843
    .p2align    4, 0x90
    .type   julia_f_11843,@function
julia_f_11843:                          # @julia_f_11843
; Function Signature: f(Int64, Int64)
; ┌ @ /home/hellmund/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:222 within `f`
# %bb.0:                                # %top
    #DEBUG_VALUE: f:x <- \$rdi
    #DEBUG_VALUE: f:y <- \$rsi
    push rbp
    mov  rbp, rsp
    push rbx
    push rax
    mov  rbx, rdi
; │ @ /home/hellmund/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:223 within `f`
; │┌ @ intfuncs.jl:437 within `literal_pow`
; ││┌ @ int.jl:88 within `*`
    imul rbx, rdi
; │└└
; │┌ @ math.jl:1544 within `log`
; ││┌ @ float.jl:378 within `float`
; │││┌ @ float.jl:352 within `AbstractFloat`
; ││││┌ @ float.jl:245 within `Float64`
    vcvtsi2sd    xmm0, xmm0, rsi
; ││└└└
; ││ @ math.jl:1546 within `log`
    movabs   rax, offset j_log_11846
    call rax
; │└
```

```
; │┌ @ promotion.jl:433 within `+`
; ││┌ @ promotion.jl:404 within `promote`
; │││┌ @ promotion.jl:379 within `_promote`
; ││││┌ @ number.jl:7 within `convert`
; │││││┌ @ float.jl:245 within `Float64`
    vcvtsi2sd    xmm1, xmm1, rbx
; │└└└└└
; │ @ promotion.jl:433 within `+` @ float.jl:495
    vaddsd   xmm0, xmm0, xmm1
; │└
; │ @ /home/hellmund/Julia/Book26/JuliaBook/chapters/5_TricksHelp.qmd:224 within `f`
; │┌ @ promotion.jl:434 within `\*` @ float.jl:497
    vaddsd   xmm0, xmm0, xmm0
; │└
    add  rsp, 8
    pop  rbx
    pop  rbp
    ret
.Lfunc_end0:
    .size   julia_f_11843, .Lfunc_end0-julia_f_11843
; └
                                        # -- End function
    .type   ".L+Core.Float64#11845",@object # @"+Core.Float64#11845"
    .section    .lrodata,"al",@progbits
    .p2align    3, 0x0
".L+Core.Float64#11845":
    .quad   ".L+Core.Float64#11845.jit"
    .size   ".L+Core.Float64#11845", 8

.set ".L+Core.Float64#11845.jit", 140237500508288
    .size   ".L+Core.Float64#11845.jit", 8
    .section    ".note.GNU-stack","",@progbits
```

6. Machine Numbers

```
for x ∈ ( 3, 3.3e4,  Int16(20), Float32(3.3e4), UInt16(9) )
    @show x sizeof(x) typeof(x)
    println()
end
```

```
x = 3
sizeof(x) = 8
typeof(x) = Int64

x = 33000.0
sizeof(x) = 8
typeof(x) = Float64

x = 20
sizeof(x) = 2
typeof(x) = Int16

x = 33000.0f0
sizeof(x) = 4
typeof(x) = Float32

x = 0x0009
sizeof(x) = 2
typeof(x) = UInt16
```

6.1 Integers

Integers are stored as fixed-length bit patterns. Therefore, the value range is finite. **Within this value range** addition, subtraction, multiplication, and integer division with remainder are exact operations without rounding errors.

Integer numbers come in two types: `Signed` and `Unsigned`, which can be viewed as machine models for $\mathbb{Z}$ and $\mathbb{N}$ respectively.

6.1.1 *Unsigned integers*

```
subtypes(Unsigned)
```

```
5-element Vector{Any}:
UInt128
UInt16
UInt32
UInt64
UInt8
```

`UInts` are binary numbers with a bit width of 8, 16, 32, 64, or 128 and the corresponding value range of

$$0 \leq x < 2^n$$

They are used rarely in *scientific computing*. In low-level hardware programming, they are used, e.g., for handling binary data and memory addresses. By default, Julia displays them as hexadecimal numbers (with prefix `0x` and digits `0-9a-f`).

```
x = 0x0033efef
@show x typeof(x) Int(x)
```

```
z = UInt(32)
@show z typeof(z);
```

```
x = 0x0033efef
typeof(x) = UInt32
Int(x) = 3403759
z = 0x0000000000000020
typeof(z) = UInt64
```

6.1.2 *Signed Integers*

```
subtypes(Signed)
```

```
6-element Vector{Any}:
BigInt
Int128
Int16
Int32
Int64
Int8
```

Integers have the value range

$$-2^{n-1} \leq x < 2^{n-1}$$

In Julia, integer numbers are 64-bit by default:

```
x = 42
typeof(x)
```

```
Int64
```

Therefore, they have the value range:

$$-9.223.372.036.854.775.808 \leq x \leq 9.223.372.036.854.775.807$$

32-bit integers have the value range

$$-2.147.483.648 \leq x \leq 2.147.483.647$$

By the way, the maximum value $2^{31} - 1$ is a Mersenne prime:

```
using Primes
isprime(2^31-1)
```

```
true
```

Negative numbers are represented in two's complement:
$x \Rightarrow -x$ corresponds to: *flip all bits, then add 1*
This looks as follows:

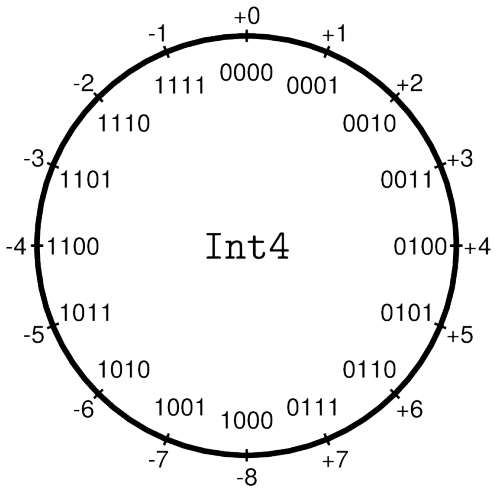


Figure 6.1: A representation of the fictional data type `Int4`

The most significant bit indicates the sign. However, its "flipping" does not correspond to the operation `x -> -x`.

> ! Important
>
> All integer arithmetic is **arithmetic modulo 2^n**
>
> ```
> x = 2^62 - 10 + 2^62
> ```
>
> ```
> 9223372036854775798
> ```
>
> ```
> x + 20
> ```
>
> ```
> -9223372036854775798
> ```
>
> No error message, no warning! Fixed-length integers do not lie on a line, but on a **circle.**

Let's look at a few *integers*:

```
using Printf

for i in (1, 2, 7, -7, 2^50, -1, Int16(7), Int8(7))
    s = bitstring(i)
    t = typeof(i)
    @printf "%20i %-5s ⇒  %s\n" i t s
end
```

```
1 Int64 ⇒ 0000000000000000000000000000000000000000000000000000000000000001
2 Int64 ⇒ 0000000000000000000000000000000000000000000000000000000000000010
7 Int64 ⇒ 0000000000000000000000000000000000000000000000000000000000000111
-7 Int64 ⇒ 1111111111111111111111111111111111111111111111111111111111111001
1125899906842624                                                        Int64
⇒ 0000000000000100000000000000000000000000000000000000000000000000
-1 Int64 ⇒ 1111111111111111111111111111111111111111111111111111111111111111
7 Int16 ⇒ 0000000000000111
7 Int8 ⇒ 00000111
```

Julia has functions that provide type information (*introspection*):

```
typemin(Int64), typemax(Int64)
```

```
(-9223372036854775808, 9223372036854775807)
```

```
typemin(UInt64), typemax(UInt64), BigInt(typemax(UInt64))
```

```
(0x0000000000000000, 0xffffffffffffffff, 18446744073709551615)
```

```
typemin(Int8), typemax(Int8)
```

```
(-128, 127)
```

6.2 Integer Arithmetic

6.2.0.1 Addition, Multiplication

The operations `+`,`-`,`*` have the usual exact arithmetic **modulo 2^{64}**.

6.2.0.2 Powers `a^b`

- Powers `a^n` are computed exactly modulo 2^{64} for natural exponents `n`.
- For negative exponents, the result is a `Float`.
- `0^0` is naturally equal to 1.

```
(-2)^63, 2^64, 3^(-3), 0^0
```

```
(-9223372036854775808, 0, 0.03703703703703704, 1)
```

- For natural exponents, *exponentiation by squaring* is used, so for example `x^23` requires only 7 multiplications:

$$x^{23} = \Big(\big((x^2)^2 \cdot x\big)^2 \cdot x\Big)^2 \cdot x$$

6.2.0.3 Division

- Division `/` produces a floating-point number.

```
x = 40/5
```

```
8.0
```

6.2.0.4 Integer Division and Remainder

- The functions `div(a,b)`, `rem(a,b)`, and `divrem(a,b)` compute the quotient of integer division, the corresponding remainder, or both as a tuple.
- For `div(a,b)` there is the operator form `a ÷ b` (input: `\div<TAB>`), and for `rem(a,b)` the operator form `a % b`.
- By default, division uses "rounding toward zero", so the corresponding remainder has the same sign as the dividend `a`:

```
@show divrem( 27, 4)
@show ( 27 ÷  4,  27 %  4)
@show (-27 ÷  4, -27 %  4 )
@show ( 27 ÷ -4,  27 % -4);
```

```
divrem(27, 4) = (6, 3)
(27 ÷ 4, 27 % 4) = (6, 3)
(-27 ÷ 4, -27 % 4) = (-6, -3)
(27 ÷ -4, 27 % -4) = (-6, 3)
```

- A rounding rule other than `RoundToZero` can be specified as the third optional argument for these functions.
- `?RoundingMode` shows the possible rounding modes.
- For the rounding rule `RoundDown` ("toward minus infinity" – so that the corresponding remainder has the same sign as the divisor `b`), there are also the functions `fld(a,b)` *(floored division)* and `mod(a,b)`:

```
@show divrem(-27, 4, RoundDown)
@show (fld(-27,  4), mod(-27,  4))
@show (fld( 27, -4), mod( 27, -4));
```

```
divrem(-27, 4, RoundDown) = (-7, 1)
(fld(-27, 4), mod(-27, 4)) = (-7, 1)
(fld(27, -4), mod(27, -4)) = (-7, -1)
```

For all rounding modes, the following holds:

```
div(a, b, RoundingMode) * b + rem(a, b, RoundingMode) = a
```

6.2.0.5 The `BigInt` Type

The `BigInt` type supports arbitrary-precision integers with dynamically allocated memory.

Numeric constants automatically have a sufficiently large type:

```
z = 10
@show typeof(z)
z = 10_000_000_000_000_000    # 10 quadrillion
@show typeof(z)
z = 10_000_000_000_000_000_000  # 10 quintillion
@show typeof(z)
z = 10_000_000_000_000_000_000_000_000_000_000_000_000   # 10 sextillion
@show typeof(z);
```

```
typeof(z) = Int64
typeof(z) = Int64
typeof(z) = Int128
typeof(z) = BigInt
```

In most cases, you must explicitly specify the `BigInt` type to avoid modulo 2^{64} arithmetic:

```
@show 3^300        BigInt(3)^300;
```

```
3 ^ 300 = 4157753088978724465
BigInt(3) ^ 300 = 136891479058588375991326027382088315966463695625337436471480190078368997177499076593800200
```

Arbitrary precision arithmetic comes at a cost of significant memory and computation time.

We compare the time and memory requirements for summing 10 million integers as `Int64` versus `BigInt`.

```
# 10^7 random numbers, uniformly distributed between -10^7 and 10^7
vec_int = rand(-10^7:10^7, 10^7)

# The same numbers as BigInts
vec_bigint = BigInt.(vec_int)
```

```
10000000-element Vector{BigInt}:
1552342
-9405309
-3287697
-6957172
7906631
-2272384
6674556
-3076566
411596
-3667357
⋮
7860807
9834428
-2977646
3679895
2271996
2786825
-9293127
```

```
6949692
9397632
```

The `@time` macro provides a rough estimate of the required time and memory:

```
@time x = sum(vec_int)
@show x typeof(x)
```

```
0.002977 seconds (46 allocations: 2.125 KiB)
x = 17784527794
typeof(x) = Int64
Int64
```

```
@time x = sum(vec_bigint)
@show x typeof(x);
```

```
0.080422 seconds (18 allocations: 856 bytes)
x = 17784527794
typeof(x) = BigInt
```

Due to Julia's just-in-time compilation, timing a single function call is not very informative. The `BenchmarkTools` package provides the `@benchmark` macro, which calls a function multiple times and displays the execution times as a histogram.

```
using BenchmarkTools

@benchmark sum($vec_int)
```

```
BenchmarkTools.Trial: 1793 samples with 1 evaluation per sample.
 Range (min … max):  2.621 ms …   8.171 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     2.748 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   2.783 ms ± 288.289 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%

  2.62 ms         Histogram: frequency by time        3.31 ms <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark sum($vec_bigint)
```

```
BenchmarkTools.Trial: 69 samples with 1 evaluation per sample.
 Range (min … max):  72.618 ms … 81.348 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     73.371 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   73.438 ms ±  1.062 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

  72.6 ms         Histogram: frequency by time        74.6 ms <

 Memory estimate: 72 bytes, allocs estimate: 3.
```

The `BigInt` addition is more than 30 times slower.

6.3 Floating-Point Numbers

In numerical mathematics, the term **machine numbers** is also commonly used.

6.3.1 Basic Idea

- A "fixed number of digits before and after the decimal point" is unsuitable for many problems.
- A separation between "significant digits" and magnitude (mantissa and exponent), as in scientific notation, is much more flexible.

$$345.2467 \times 10^3 \qquad 34.52467 \times 10^4 \qquad \underline{3.452467 \times 10^5} \qquad 0.3452467 \times 10^6 \qquad 0.03452467 \times 10^7$$

- For uniqueness, one must choose one of these forms. In the mathematical analysis of machine numbers, one often chooses the form where the first digit after the decimal point is nonzero. Thus, for the mantissa m:

$$1 > m \geq (0.1)_b = b^{-1},$$

where b denotes the base of the number system.
- We choose the form that corresponds to the actual implementation on the computer and specify: The representation with exactly one nonzero digit before the decimal point is the **normalized representation**. Thus,

$$(10)_b = b > m \geq 1.$$

- For binary numbers `1.01101`, this digit is always equal to one, and one can omit storing this digit. This actually stored (shortened) mantissa we denote by M, so that

$$m = 1 + M$$

holds.

i Machine Numbers

The set of machine numbers $\mathbb{M}(b, p, e_{min}, e_{max})$ is characterized by the base b, the mantissa length p, and the value range of the exponent $\{e_{min}, ..., e_{max}\}$.

In our convention, the mantissa of a normalized machine number has one digit (of base b) before the decimal point and $p - 1$ digits after the decimal point.

If $b = 2$, one needs only $p - 1$ bits to store the mantissa of normalized floating-point numbers.

The IEEE 754 standard, implemented by most modern processors and programming languages, defines

- `Float32` as $\mathbb{M}(2, 24, -126, 127)$ and
- `Float64` as $\mathbb{M}(2, 53, -1022, 1023)$.

6.3.2 Structure of `Float64` according to the IEEE 754 standard

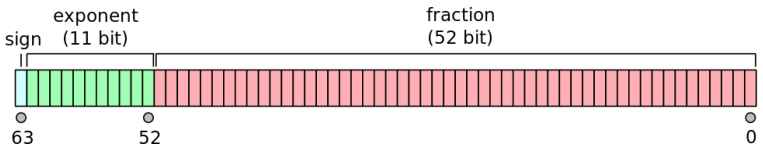


Figure 6.2: Structure of a `Float64`

- 1 sign bit S
- 11 bits for the exponent, thus $0 \leq E \leq 2047$
- The values $E = 0$ and $E = (11111111111)_2 = 2047$ are reserved for encoding special values such as $\pm 0, \pm\infty$, NaN *(Not a Number)* and subnormal numbers.
- 52 bits for the (shortened) mantissa M, $\quad 0 \leq M < 1$, corresponding to approximately 16 decimal digits
- Thus, the number represented is:

$$x = (-1)^S \cdot (1 + M) \cdot 2^{E-1023}$$

An example:

```
x = 27.56640625
bitstring(x)
```

```
"0100000000111011100100010000000000000000000000000000000000000000"
```

This can be displayed more clearly:

```
function printbitsf64(x::Float64)
    s = bitstring(x)
    printstyled(s[1], color = :blue, reverse=true)
    printstyled(s[2:12], color = :green, reverse=true)
    printstyled(s[13:end], color=:red, bold=true, reverse=true)
    print("\n")
end

printbitsf64(x)
```

```
0100000000111011100100010000000000000000000000000000000000000000
```

and we can read S (blue), E (green), and M (red):

$$
\begin{aligned}
S &= 0 \\
E &= (10000000011)_2 = 2^{10} + 2^1 + 2^0 = 1027 \\
M &= (.1011100100001)_2 = \frac{1}{2} + \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^{12}} \\
x &= (-1)^S \cdot (1 + M) \cdot 2^{E-1023}
\end{aligned}
$$

... and thus reconstruct the number:

```
x = (1 + 1/2 + 1/8 + 1/16 + 1/32 + 1/256 + 1/4096) * 2^4
```

```
27.56640625
```

- The set of machine numbers $\mathbb{M}$ forms a finite, discrete subset of $\mathbb{R}$. There exists a smallest and a largest machine number; all other elements $x \in \mathbb{M}$ have both a predecessor and successor in $\mathbb{M}$.
- What is the successor of x in $\mathbb{M}$? To do this, we set the smallest mantissa bit from 0 to 1.
- Converting a string of zeros and ones into the corresponding machine number:

```
ux = parse(UInt64, "0100000000111011100100010000000000000000000000000000000000000001",
base=2)
reinterpret(Float64, ux)
```

```
27.566406250000004
```

However, Julia can do this more simply:

```
y = nextfloat(x)
```

```
27.566406250000004
```

The predecessor of x is:

```
z = prevfloat(x)
println(z)
printbitsf64(z)
```

```
27.566406249999996
0100000000111011100100001111111111111111111111111111111111111111
```

6.4 Machine Epsilon

- The distance between `1` and its successor `nextfloat(1)` is called **machine epsilon**.
- For `Float64` with a mantissa length of 52 bits, $\epsilon = 2^{-52}$.

```
@show nextfloat(1.) - 1   2^-52   eps(Float64);
```

```
nextfloat(1.0) - 1 = 2.220446049250313e-16
2 ^ -52 = 2.220446049250313e-16
eps(Float64) = 2.220446049250313e-16
```

- Machine epsilon measures the relative distance between machine numbers and quantifies the statement: "64-bit floating-point numbers have a precision of approximately 16 decimal digits."
- Machine epsilon should not be confused with the smallest positive floating-point number:

```
floatmin(Float64)
```

```
2.2250738585072014e-308
```

- Part of the literature uses a different definition of machine epsilon, which is half as large.

$$\epsilon' = \frac{\epsilon}{2} \approx 1.1 \times 10^{-16}$$

  This is the maximum relative error that can occur when rounding a real number to the nearest machine number.
- Since numbers in the interval $(1 - \epsilon', 1 + \epsilon']$ are rounded to the machine number 1, one can also define ϵ' as: *the largest number for which* $1 + \epsilon' = 1$ *still holds in machine arithmetic.*

This allows to compute machine epsilon using the floating point arithmetic:

```
Eps = 1
while(1 != 1 + Eps)
    Eps /= 2
    println(1+Eps)
end
Eps
```

```
1.5
1.25
1.125
1.0625
1.03125
1.015625
1.0078125
1.00390625
1.001953125
1.0009765625
1.00048828125
1.000244140625
1.0001220703125
1.00006103515625
1.000030517578125
1.0000152587890625
1.0000076293945312
1.0000038146972656
1.0000019073486328
1.0000009536743164
1.0000004768371582
1.000000238418579
1.0000001192092896
1.0000000596046448
1.0000000298023224
1.0000000149011612
1.0000000074505806
1.0000000037252903
1.0000000018626451
1.0000000009313226
1.0000000004656613
1.0000000002328306
1.0000000001164153
1.0000000000582077
1.0000000000291038
1.000000000014552
1.000000000007276
1.000000000003638
1.000000000001819
1.0000000000009095
1.0000000000004547
```

```
1.0000000000002274
1.0000000000001137
1.0000000000000568
1.0000000000000284
1.0000000000000142
1.000000000000007
1.0000000000000036
1.0000000000000018
1.0000000000000009
1.0000000000000004
1.0000000000000002
1.0
1.1102230246251565e-16
```

or as a bit pattern:

```
Eps=1
while 1 != 1 + Eps
    Eps/= 2
    printbitsf64(1+Eps)
end
Eps
```

```
0011111111111000000000000000000000000000000000000000000000000000
0011111111111010000000000000000000000000000000000000000000000000
0011111111111001000000000000000000000000000000000000000000000000
0011111111111000100000000000000000000000000000000000000000000000
0011111111111000010000000000000000000000000000000000000000000000
0011111111111000001000000000000000000000000000000000000000000000
0011111111111000000100000000000000000000000000000000000000000000
0011111111111000000010000000000000000000000000000000000000000000
0011111111111000000001000000000000000000000000000000000000000000
0011111111111000000000100000000000000000000000000000000000000000
0011111111111000000000010000000000000000000000000000000000000000
0011111111111000000000001000000000000000000000000000000000000000
0011111111111000000000000100000000000000000000000000000000000000
0011111111111000000000000010000000000000000000000000000000000000
0011111111111000000000000001000000000000000000000000000000000000
0011111111111000000000000000100000000000000000000000000000000000
0011111111111000000000000000010000000000000000000000000000000000
0011111111111000000000000000001000000000000000000000000000000000
0011111111111000000000000000000100000000000000000000000000000000
0011111111111000000000000000000010000000000000000000000000000000
0011111111111000000000000000000001000000000000000000000000000000
0011111111111000000000000000000000100000000000000000000000000000
0011111111111000000000000000000000010000000000000000000000000000
0011111111111000000000000000000000001000000000000000000000000000
0011111111111000000000000000000000000100000000000000000000000000
0011111111111000000000000000000000000010000000000000000000000000
0011111111111000000000000000000000000001000000000000000000000000
0011111111111000000000000000000000000000100000000000000000000000
0011111111111000000000000000000000000000010000000000000000000000
0011111111111000000000000000000000000000001000000000000000000000
0011111111111000000000000000000000000000000100000000000000000000
0011111111111000000000000000000000000000000010000000000000000000
0011111111111000000000000000000000000000000001000000000000000000
0011111111111000000000000000000000000000000000100000000000000000
0011111111111000000000000000000000000000000000010000000000000000
0011111111111000000000000000000000000000000000001000000000000000
0011111111111000000000000000000000000000000000000100000000000000
0011111111111000000000000000000000000000000000000010000000000000
0011111111111000000000000000000000000000000000000001000000000000
0011111111111000000000000000000000000000000000000000100000000000
0011111111111000000000000000000000000000000000000000010000000000
0011111111111000000000000000000000000000000000000000001000000000
0011111111111000000000000000000000000000000000000000000100000000
0011111111111000000000000000000000000000000000000000000010000000
0011111111111000000000000000000000000000000000000000000001000000
```

```
0011111111110000000000000000000000000000000000000000000000100000
0011111111110000000000000000000000000000000000000000000000010000
0011111111110000000000000000000000000000000000000000000000001000
0011111111110000000000000000000000000000000000000000000000000100
0011111111110000000000000000000000000000000000000000000000000010
0011111111110000000000000000000000000000000000000000000000000001
0011111111110000000000000000000000000000000000000000000000000000
1.1102230246251565e-16
```

i The Set of (normalized) Machine Numbers

- In the interval $[1, 2)$ there are 2^{52} equidistant machine numbers.
- After that, the exponent increases by 1 and the mantissa M is reset to 0. Thus, the interval $[2, 4)$ again contains 2^{52} equidistant machine numbers, as does the interval $[4, 8)$ up to $[2^{1023}, 2^{1024})$.
- Likewise, in the intervals $[\frac{1}{2}, 1), [\frac{1}{4}, \frac{1}{2}), \ldots$ there are 2^{52} equidistant machine numbers each, down to $[2^{-1022}, 2^{-1021})$.
- This forms the set $\mathbb{M}_+$ of positive machine numbers, and we have

$$\mathbb{M} = -\mathbb{M}_+ \cup \{0\} \cup \mathbb{M}_+$$

The largest and smallest positive representable normalized floating-point numbers of a floating-point type can be queried:

```
@show floatmax(Float64)
printbitsf64(floatmax(Float64))

@show floatmin(Float64)
printbitsf64(floatmin(Float64))
```

```
floatmax(Float64) = 1.7976931348623157e308
0111111111101111111111111111111111111111111111111111111111111111
floatmin(Float64) = 2.2250738585072014e-308
0000000000010000000000000000000000000000000000000000000000000000
```

6.5 Rounding to Machine Numbers

- The map rd: $\mathbb{R} \to \mathbb{M}$ should round to the nearest representable number.
- Standard rounding mode is *round to nearest, ties to even*: when a value falls exactly midway between two machine numbers *(tie)*, the one with 0 as its last mantissa bit is selected.
- Justification: this way, we round up in 50% of the cases and down in 50% of the cases, thus avoiding a "statistical drift" in longer calculations.
- It holds:

$$\frac{|\, x - \mathrm{rd}\,(x)\,|}{|\, x\,|} \leq \frac{1}{2}\epsilon$$

6.6 Machine Number Arithmetic

The machine numbers, as a subset of $\mathbb{R}$, are not algebraically closed. Even the sum of two machine numbers is generally not representable as a machine number.

> ! Important
>
> The IEEE 754 standard requires that machine number arithmetic produces the *rounded exact result*:
> The result must be equal to the one that would result from an exact execution of the corresponding operation followed by rounding.
>
> $$a \oplus b = \text{rd}\ (a + b)$$
>
> The same must hold for the implementation of standard functions such as `sqrt()`, `log()`, `sin()`, ... – they also return the machine number closest to the exact result.

Arithmetic is *not associative*:

```
1 +  10^-16 + 10^-16
```

```
1.0
```

```
1 + (10^-16 + 10^-16)
```

```
1.0000000000000002
```

In the first case (without parentheses), evaluation proceeds from left to right:

$$\begin{aligned} 1 \oplus 10^{-16} \oplus 10^{-16} &= (1 \oplus 10^{-16}) \oplus 10^{-16} \\ &= \text{rd}\ (\text{rd}\ (1 + 10^{-16}) + 10^{-16}) \\ &= \text{rd}\ (1 + 10^{-16}) \\ &= 1 \end{aligned}$$

In the second case, one obtains:

$$\begin{aligned} 1 \oplus (10^{-16} \oplus 10^{-16}) &= \text{rd}\ (1 + \text{rd}\ (10^{-16} + 10^{-16})) \\ &= \text{rd}\ (1 + 2 * 10^{-16}) \\ &= 1.0000000000000002 \end{aligned}$$

One should also remember that even “simple” decimal fractions cannot always be represented exactly as machine numbers:

$$\begin{aligned} (0.1)_{10} &= (0.0001100110011001100110011001100...)_2 = (0.000\overline{1100})_2 \\ (0.3)_{10} &= (0.0100110011001100110011001100..)_2 = (0.0\overline{1001})_2 \end{aligned}$$

```
printbitsf64(0.1)
printbitsf64(0.3)
```

```
0011111111011001100110011001100110011001100110011001100110011010
0011111111010011001100110011001100110011001100110011001100110011
```

Consequence:

```
0.1 + 0.1 == 0.2
```

```
true
```

```
0.2 + 0.1 == 0.3
```

```
false
```

```
0.2 + 0.1
```

```
0.30000000000000004
```

When outputting a machine number, the binary fraction must be converted to a decimal fraction. Julia can display more digits of this decimal fraction expansion:

```
using Printf
@printf("%.30f", 0.1)
```

```
0.100000000000000005551115123126
```

```
@printf("%.30f", 0.3)
```

```
0.299999999999999988897769753748
```

The binary fraction mantissa of a machine number can have a long or even infinitely periodic decimal expansion. But one should not be misled into thinking that this is "higher precision"!

> ! Important
>
> Key message: When testing `Float`s for equality, one should almost always define a realistic tolerance `epsilon` appropriate to the problem and test:
>
> ```
> epsilon = 1.e-10
>
> if abs(x-y) < epsilon
>    # ...
> end
> ```

6.7 Normalized and Subnormal Machine Numbers

The gap between zero and the smallest normalized machine number $2^{-1022} \approx 2.22 \times 10^{-308}$ is filled with subnormal machine numbers.

Let's look at a simple model:

- Let $\mathbb{M}(10,4,\pm5)$ be the set of machine numbers to base 10 with 4 mantissa digits (one before the decimal point, 3 after) and the exponent range $-5 \le E \le 5$.
- Then the normalized representation (nonzero leading digit) of e.g. 1234.0 is 1.234e3 and of 0.00789 is 7.890e-3.
- It is essential that machine numbers remain normalized at each computation step. Only then is the full mantissa length utilized, maximizing accuracy.
- The smallest positive normalized number in our model is `x = 1.000e-5`. Already `x/2` would have to be rounded to 0.
- But for many applications, it is advantageous to allow also subnormal numbers and represent `x/2` as `0.500e-5` or `x/20` as `0.050e-5`.
- This *gradual underflow* is of course associated with a loss of significant digits and thus accuracy.

In the `Float` data type, such *subnormal values* are represented by an exponent field in which all bits are equal to zero:

```
using Printf

x = 2 * floatmin(Float64)   # 2*smallest normalized floating-point number > 0

while x != 0
    x /= 2
    @printf "%14.6g  " x
    printbitsf64(x)
end
```

```
  2.22507e-308  0000000000010000000000000000000000000000000000000000000000000000
  1.11254e-308  0000000000001000000000000000000000000000000000000000000000000000
  5.56268e-309  0000000000000100000000000000000000000000000000000000000000000000
  2.78134e-309  0000000000000010000000000000000000000000000000000000000000000000
  1.39067e-309  0000000000000001000000000000000000000000000000000000000000000000
  6.95336e-310  0000000000000000100000000000000000000000000000000000000000000000
```

```
 3.47668e-310  0000000000000000010000000000000000000000000000000000000000000000
 1.73834e-310  0000000000000000001000000000000000000000000000000000000000000000
 8.69169e-311  0000000000000000000100000000000000000000000000000000000000000000
 4.34585e-311  0000000000000000000010000000000000000000000000000000000000000000
 2.17292e-311  0000000000000000000001000000000000000000000000000000000000000000
 1.08646e-311  0000000000000000000000100000000000000000000000000000000000000000
 5.43231e-312  0000000000000000000000010000000000000000000000000000000000000000
 2.71615e-312  0000000000000000000000001000000000000000000000000000000000000000
 1.35808e-312  0000000000000000000000000100000000000000000000000000000000000000
 6.79039e-313  0000000000000000000000000010000000000000000000000000000000000000
 3.39519e-313  0000000000000000000000000001000000000000000000000000000000000000
  1.6976e-313  0000000000000000000000000000100000000000000000000000000000000000
 8.48798e-314  0000000000000000000000000000010000000000000000000000000000000000
 4.24399e-314  0000000000000000000000000000001000000000000000000000000000000000
   2.122e-314  0000000000000000000000000000000100000000000000000000000000000000
   1.061e-314  0000000000000000000000000000000010000000000000000000000000000000
 5.30499e-315  0000000000000000000000000000000001000000000000000000000000000000
 2.65249e-315  0000000000000000000000000000000000100000000000000000000000000000
 1.32625e-315  0000000000000000000000000000000000010000000000000000000000000000
 6.63124e-316  0000000000000000000000000000000000001000000000000000000000000000
 3.31562e-316  0000000000000000000000000000000000000100000000000000000000000000
 1.65781e-316  0000000000000000000000000000000000000010000000000000000000000000
 8.28905e-317  0000000000000000000000000000000000000001000000000000000000000000
 4.14452e-317  0000000000000000000000000000000000000000100000000000000000000000
 2.07226e-317  0000000000000000000000000000000000000000010000000000000000000000
 1.03613e-317  0000000000000000000000000000000000000000001000000000000000000000
 5.18065e-318  0000000000000000000000000000000000000000000100000000000000000000
 2.59033e-318  0000000000000000000000000000000000000000000010000000000000000000
 1.29516e-318  0000000000000000000000000000000000000000000001000000000000000000
 6.47582e-319  0000000000000000000000000000000000000000000000100000000000000000
 3.23791e-319  0000000000000000000000000000000000000000000000010000000000000000
 1.61895e-319  0000000000000000000000000000000000000000000000001000000000000000
 8.09477e-320  0000000000000000000000000000000000000000000000000100000000000000
 4.04739e-320  0000000000000000000000000000000000000000000000000010000000000000
 2.02369e-320  0000000000000000000000000000000000000000000000000001000000000000
 1.01185e-320  0000000000000000000000000000000000000000000000000000100000000000
 5.05923e-321  0000000000000000000000000000000000000000000000000000010000000000
 2.52962e-321  0000000000000000000000000000000000000000000000000000001000000000
 1.26481e-321  0000000000000000000000000000000000000000000000000000000100000000
 6.32404e-322  0000000000000000000000000000000000000000000000000000000010000000
 3.16202e-322  0000000000000000000000000000000000000000000000000000000001000000
 1.58101e-322  0000000000000000000000000000000000000000000000000000000000100000
 7.90505e-323  0000000000000000000000000000000000000000000000000000000000010000
 3.95253e-323  0000000000000000000000000000000000000000000000000000000000001000
 1.97626e-323  0000000000000000000000000000000000000000000000000000000000000100
 9.88131e-324  0000000000000000000000000000000000000000000000000000000000000010
 4.94066e-324  0000000000000000000000000000000000000000000000000000000000000001
            0  0000000000000000000000000000000000000000000000000000000000000000
```

6.8 Special Values

Floating-point arithmetic defines certain special values, e.g.,

```
nextfloat(floatmax(Float64))
```

```
Inf
```

```
for x ∈ (NaN, Inf, -Inf, -0.0)
    @printf "%6g  " x
    printbitsf64(x)
end
```

```
   NaN  0111111111111000000000000000000000000000000000000000000000000000
   Inf  0111111111110000000000000000000000000000000000000000000000000000
  -Inf  1111111111110000000000000000000000000000000000000000000000000000
    -0  1000000000000000000000000000000000000000000000000000000000000000
```

- An exponent overflow leads to the result `Inf` or `-Inf`.

```
2/0, -3/0, floatmax(Float64) * 1.01, exp(1300)
```

```
(Inf, -Inf, Inf, Inf)
```

- One can continue calculating with these values:

```
-Inf + 20, Inf/30, 23/-Inf, sqrt(Inf),  Inf * 0, Inf - Inf
```

```
(-Inf, Inf, -0.0, Inf, NaN, NaN)
```

- `NaN` *(Not a Number)* represents the result of an undefined operation. All further operations with `NaN` also result in `NaN`.

```
0/0, Inf - Inf, 2.3NaN, sqrt(NaN)
```

```
(NaN, NaN, NaN, NaN)
```

- Since `NaN` represents an undefined value, it is not equal to anything, not even to itself. This is sensible, because if two variables `x` and `y` are computed as `NaN`, one should not conclude that they are equal.
- There is therefore a boolean function `isnan()` to test for `NaN`.

```
x = 0/0
y = Inf - Inf
@show x==y   NaN==NaN  isfinite(NaN) isinf(NaN) isnan(x) isnan(y);
```

```
x == y = false
NaN == NaN = false
isfinite(NaN) = false
isinf(NaN) = false
isnan(x) = true
isnan(y) = true
```

- There is a "minus zero". It signals a numerical underflow of a small *negative* quantity.

```
@show 23/-Inf   -2/exp(1200)    -0.0==0.0;
```

```
23 / -Inf = -0.0
-2 / exp(1200) = -0.0
-0.0 == 0.0 = true
```

6.9 Mathematical Functions

Julia has the usual mathematical functions
`sqrt, exp, log, log2, log10, sin, cos,..., asin, acos,..., sinh,..., gcd, lcm, factorial,...,abs, max, min,...`,
including e.g. the rounding functions

- `floor(T,x)` = $\lfloor x \rfloor$
- `ceil(T,x)` = $\lceil x \rceil$

```
floor(3.4),   floor(Int64, 3.5),   floor(Int64, -3.5)
```

```
(3.0, 3, -4)
```

```
ceil(3.4),    ceil(Int64, 3.5),    ceil(Int64, -3.5)
```

```
(4.0, 4, -3)
```

Also worth noting is `atan(y, x)`, the two-argument arctangent (known as `atan2` in many programming languages, see atan2). This solves the problem of converting from Cartesian to polar coordinates without awkward case distinctions.

- `atan(y,x)` is the angle of the polar coordinates of (x,y) in the interval $(-\pi, \pi]$. In the 1st and 4th quadrants, it is therefore equal to `atan(y/x)`

```
atan(3, -2),    atan(-3, 2),    atan(-3/2)
```

```
(2.1587989303424644, -0.982793723247329, -0.982793723247329)
```

6.10 Conversion Between Strings and Numbers

Use the functions `parse()` and `string()` for such conversions:

```
parse(Int64, "1101", base=2)
```

```
13
```

```
string(13, base=2)
```

```
"1101"
```

```
string(1/7)
```

```
"0.14285714285714285"
```

```
string(77, base=16)
```

```
"4d"
```

For conversion of numerical types into each other, one can use the type names. Type names are also constructors:

```
x = Int8(67)
@show x   typeof(x);
```

```
x = 67
typeof(x) = Int8
```

```
z = UInt64(3459)
```

```
0x0000000000000d83
```

```
y = Float64(z)
```

```
3459.0
```

6.11 Literature

- D. Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic
- C. Vuik, Some Disasters caused by numerical errors

7. A Case Study on Floating-Point Arithmetic Stability

This chapter is inspired by the example in *Christoph Überhuber*, "Computer-Numerik" Vol. 1, Springer 1995, Chap. 2.3.

7.1 Calculation of π According to Archimedes

A lower bound for 2π – the circumference of the unit circle – is obtained by summing the side lengths of a regular n-gon inscribed in the unit circle. The figure on the left illustrates the iterative doubling of the number of vertices starting from a square with side length

$$s_4 = \sqrt{2}$$

.

Fig. 1

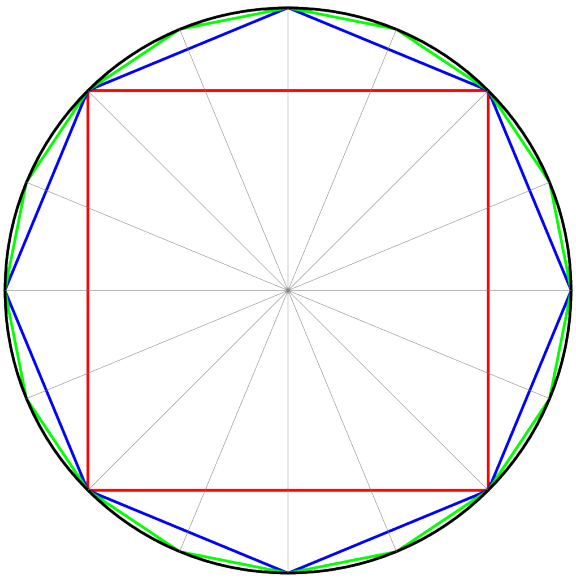
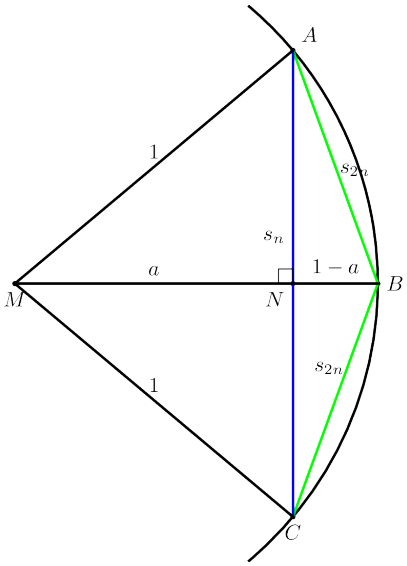


Fig. 2



The second figure shows the geometry of the vertex doubling.

With $|\overline{AC}| = s_n$, $|\overline{AB}| = |\overline{BC}| = s_{2n}$, $|\overline{MN}| = a$, $|\overline{NB}| = 1 - a$, the Pythagorean theorem applied to triangles MNA and NBA respectively yields

$$a^2 + \left(\frac{s_n}{2}\right)^2 = 1 \qquad \text{and} \qquad (1-a)^2 + \left(\frac{s_n}{2}\right)^2 = s_{2n}^2$$

Elimination of a gives the recursion

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}} \qquad \text{with initial value} \qquad s_4 = \sqrt{2}$$

for the length s_n of one side of the inscribed regular n-gon.

The sequence $(n \cdot s_n)$ converges monotonically from below to the limit 2π:

$$n\, s_n \to 2\pi$$

The relative error of approximating 2π by an n-gon is:

$$\epsilon_n = \left| \frac{n\, s_n - 2\pi}{2\pi} \right|$$

7.2 Two Iteration Formulas

The equation

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}} \qquad \text{(Iteration A)}$$

is mathematically equivalent to:

$$s_{2n} = \frac{s_n}{\sqrt{2 + \sqrt{4 - s_n^2}}} \qquad \text{(Iteration B)}$$

However, Iteration A is ill-conditioned and numerically unstable, as the following code demonstrates. The output shows the approximation for π for both formulae iteration.

```
using Printf

iterationA(s) =  sqrt(2 - sqrt(4 - s^2))
iterationB(s) = s / sqrt(2 + sqrt(4 - s^2))

s_B = s_A = sqrt(2)  # initial value

ϵ(x) = abs(x - 2π)/2π  # relative error

ϵ_A = Float64[]  # vectors for the plot
ϵ_B = Float64[]
is = Float64[]

@printf """Approx. value of π
           n-gon       Iteration A            Iteration B
"""

for i in 3:35
    push!(is, i)
    s_A = iterationA(s_A)
    s_B = iterationB(s_B)
    doublePi_A = 2^i * s_A
    doublePi_B = 2^i * s_B
    push!(ϵ_A, ϵ(doublePi_A))
    push!(ϵ_B, ϵ(doublePi_B))
      @printf "%14i %20.15f %20.15f\n" 2^i doublePi_A/2 doublePi_B/2
end
```

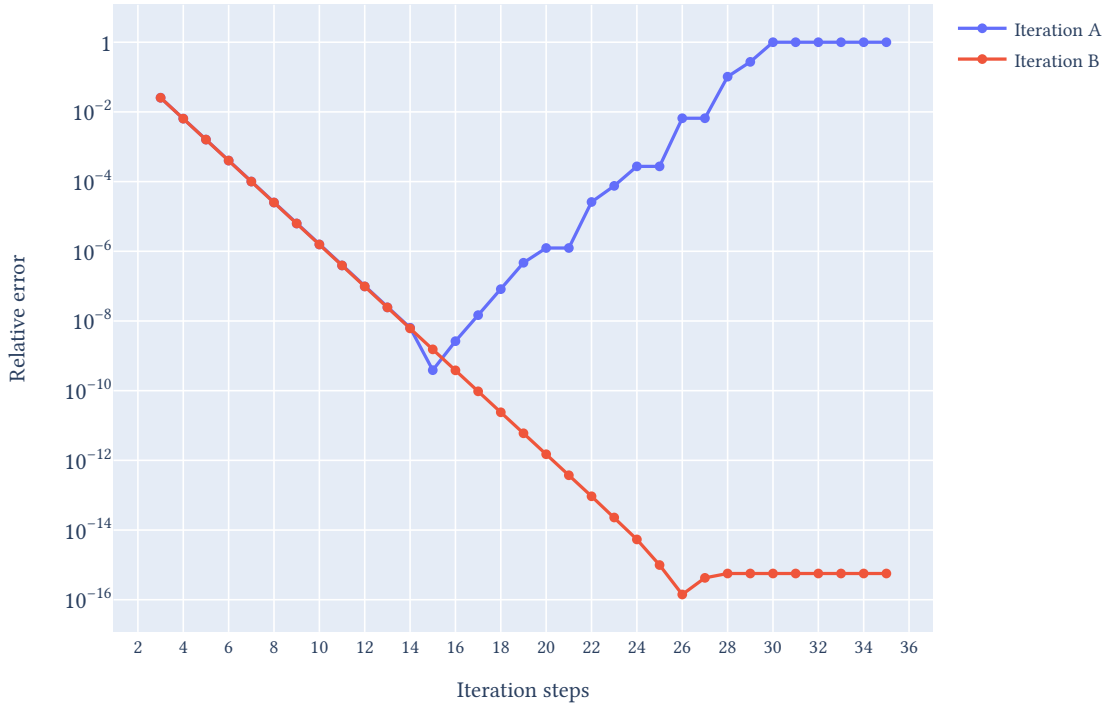
```
Approx. value of π
n-gon Iteration A Iteration B
8 3.061467458920719 3.061467458920718
16 3.121445152258053 3.121445152258052
32 3.136548490545941 3.136548490545939
64 3.140331156954739 3.140331156954753
128 3.141277250932757 3.141277250932773
256 3.141513801144145 3.141513801144301
512 3.141572940367883 3.141572940367092
1024 3.141587725279961 3.141587725277160
2048 3.141591421504635 3.141591421511200
4096 3.141592345611077 3.141592345570118
8192 3.141592576545004 3.141592576584873
16384 3.141592633463248 3.141592634338564
32768 3.141592654807589 3.141592648776986
65536 3.141592645321215 3.141592652386592
```

```
131072 3.141592607375720 3.141592653288993
262144 3.141592910939673 3.141592653514594
524288 3.141594125195191 3.141592653570994
1048576 3.141596553704820 3.141592653585094
2097152 3.141596553704820 3.141592653588619
4194304 3.141674265021758 3.141592653589501
8388608 3.141829681889202 3.141592653589721
16777216 3.142451272494134 3.141592653589776
33554432 3.142451272494134 3.141592653589790
67108864 3.162277660168380 3.141592653589794
134217728 3.162277660168380 3.141592653589794
268435456 3.464101615137754 3.141592653589795
536870912 4.000000000000000 3.141592653589795
1073741824 0.000000000000000 3.141592653589795
2147483648 0.000000000000000 3.141592653589795
4294967296 0.000000000000000 3.141592653589795
8589934592 0.000000000000000 3.141592653589795
17179869184 0.000000000000000 3.141592653589795
34359738368 0.000000000000000 3.141592653589795
```

While Iteration B stabilizes to a value accurate within machine precision, Iteration A is unstable and diverges. A plot of the relative errors ϵ_i confirms this:

```
using PlotlyJS

layout = Layout(xaxis_title="Iteration steps", yaxis_title="Relative error",
            yaxis_type="log", yaxis_exponentformat="power",
            xaxis_tick0=2, xaxis_dtick=2)

plot([scatter(x=is, y=ϵ_A, mode="markers+lines", name="Iteration A", yscale=:log10),
      scatter(x=is, y=ϵ_B, mode="markers+lines", name="Iteration B", yscale=:log10)],
      layout)
```

```
QuartoNotebookWorkerPlotlyBaseExt.PlotlyRequireJSConfig()
```



7.3 Stability and Cancellation

At $i = 26$, Iteration B reaches a relative error on the order of machine epsilon:

```
ϵ_B[22:28]
```

```
7-element Vector{Float64}:
5.3716034620272725e-15
9.895059008997609e-16
1.4135798584282297e-16
4.240739575284689e-16
5.654319433712919e-16
5.654319433712919e-16
5.654319433712919e-16
```

Further iterations do not improve the result; it stabilizes at a relative error of approximately 2.5 machine epsilons.

```
ϵ_B[end]/eps(Float64)
```

```
2.5464790894703255
```

Iteration A is unstable; already at $i = 16$, the relative error begins to grow again.

The cause is a typical numerical cancellation effect. The side lengths s_n become very small very quickly. Thus $a_n = \sqrt{4 - s_n^2}$ is only slightly smaller than 2, and computing $s_{2n} = \sqrt{2 - a_n}$ leads to a catastrophic cancellation.

```
setprecision(80) # precision for BigFloat

s = sqrt(BigFloat(2))

@printf "     a = √(4-s^2) as BigFloat       and as Float64\n\n"

for i= 3:44
    s = iterationA(s)
    x = sqrt(4-s^2)
    if i > 20
        @printf "%i %30.26f %20.16f \n" i  x  Float64(x)
    end
end
```

```
a = √(4-s^2) as BigFloat and as Float64

21 1.99999999999775591177215422 1.9999999999977560
22 1.99999999999943897794303856 1.9999999999994389
23 1.99999999999985974448576005 1.9999999999998597
24 1.99999999999996493612143919 1.9999999999999649
25 1.99999999999999123403035980 1.9999999999999913
26 1.99999999999999780850758995 1.9999999999999978
27 1.99999999999999945212689707 1.9999999999999996
28 1.99999999999999986303172344 1.9999999999999998
29 1.99999999999999996575793045 2.0000000000000000
30 1.99999999999999999143948303 2.0000000000000000
31 1.99999999999999999785987034 2.0000000000000000
32 1.99999999999999999946496800 2.0000000000000000
33 1.99999999999999999986624159 2.0000000000000000
34 1.99999999999999999996656040 2.0000000000000000
35 1.99999999999999999999163886 2.0000000000000000
36 1.99999999999999999999790889 2.0000000000000000
37 1.99999999999999999999947722 2.0000000000000000
38 1.99999999999999999999986931 2.0000000000000000
39 1.99999999999999999999996691 2.0000000000000000
40 1.99999999999999999999999173 2.0000000000000000
41 1.99999999999999999999999835 2.0000000000000000
42 1.99999999999999999999999835 2.0000000000000000
43 1.99999999999999999999999835 2.0000000000000000
44 1.99999999999999999999999835 2.0000000000000000
```

This demonstrates the loss of significant digits. It also shows that using `BigFloat` with 80 bits of precission (mantissa length) only slightly delays the onset of the cancellation effect.

Countermeasures against unstable algorithms typically require better, stable algorithms, not more precise machine numbers.

8. The Julia Type System

One can write extensive programs in Julia without using a single type declaration. This is, of course, intentional and designed to simplify users' work.

However, for a deeper understanding we will now examine the underlying type system.

8.1 The Type Hierarchy: A Case Study with Numeric Types

The type system has the structure of a tree whose root is the type `Any`. The functions `subtypes()` and `supertype()` can be used to explore the tree. `subtypes()` displays all child nodes, while `supertype()` shows the parent.

```
subtypes(Int64)
```

```
Type[]
```

The result is an empty list of types. `Int64` is a so-called **concrete type** with no subtypes.

Let's now traverse this branch upward to the root (in computer science, trees are typically inverted).

```
supertype(Int64)
```

```
Signed
```

```
supertype(Signed)
```

```
Integer
```

```
supertype(Integer)
```

```
Real
```

```
supertype(Real)
```

```
Number
```

```
supertype(Number)
```

```
Any
```

This would have been faster, by the way: The function `supertypes()` (with plural-s) shows all ancestors.

```
supertypes(Int64)
```

```
(Int64, Signed, Integer, Real, Number, Any)
```

We can now examine the nodes:

```
subtypes(Real)
```

```
4-element Vector{Any}:
AbstractFloat
AbstractIrrational
Integer
Rational
```

A simple recursive function can display the entire subtree:

```
function show_subtype_tree(T, i=0)
    println("       "^i, T)
    for Ts ∈ subtypes(T)
        show_subtype_tree(Ts, i+1)
    end
end

show_subtype_tree(Number)
```

```
Number
Complex
Real
AbstractFloat
BigFloat
Float16
Float32
Float64
AbstractIrrational
Irrational
Integer
Bool
Signed
BigInt
Int128
Int16
Int32
Int64
Int8
Unsigned
UInt128
UInt16
UInt32
UInt64
UInt8
Rational
```

Below is the same hierarchy as an image (made with LaTeX/TikZ):

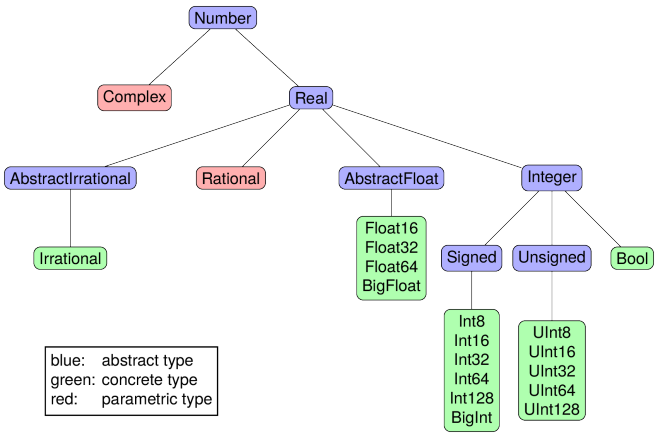


Figure 8.1: The hierarchy of numeric types

Beyond numeric types, Julia includes many others. The number of direct descendants (children) of `Any` is

```
length(subtypes(Any))
```

```
604
```

This number increases with (almost) every package loaded via `using ...`.

8.2 Abstract and Concrete Types

- An object always has a **concrete** type.
- Concrete types have no more subtypes, they are always the "leaves" of the tree.
- Concrete types specify a concrete data structure.

- Abstract types cannot be instantiated; that is, no objects can have an abstract type directly.
- They define a set of concrete types and common methods for these types.
- They can therefore be used in the definition of function types, argument types, element types of composite types, etc.

To **declare** *and* **test** the relationships in the type hierarchy, Julia provides a special operator:

```
Int64 <: Number
```

```
true
```

To test whether an object has a certain type (or an abstract supertype of it), `isa(object, typ)` is used. It is usually used in infix form and reads as the question `is x a T?`.

```
x = 17.2

42 isa Int64,  42 isa Real,  x isa Real,   x isa Float64,   x isa Integer
```

```
(true, true, true, true, false)
```

Since abstract types do not define data structures, they are simple to define. Either they are derived directly from `Any`:

```
abstract type MySuperType end

supertype(MySuperType)
```

```
Any
```

or from another abstract type:

```
abstract type MySpecialNumber <: Integer end

supertypes(MySpecialNumber)
```

```
(MySpecialNumber, Integer, Real, Number, Any)
```

By this definition, the abstract type is attached at a specific point in the type tree.

8.3 The Numeric Types `Bool` and `Irrational`

Though appearing in the numeric type tree, these types warrant brief explanation:
`Bool` is numeric in the sense that `true=1, false=0`:

```
true + true + true, false - true, sqrt(true), true/4
```

```
(3, -1, 1.0, 0.25)
```

`Irrational` is the type of certain predefined constants, such as π and *e*. According to the documentation, `Irrational` is a *"Number type representing an exact irrational value, which is automatically rounded to the correct precision in arithmetic operations with other numeric quantities".*

8.4 Union Types

When the tree structure is insufficient, abstract types can be defined as a union of arbitrary (abstract and concrete) types.

```
IntOrString = Union{Int64,String}
```

```
Union{Int64, String}
```

> **i** Example
>
> The command `methods(<)` reveals over 70 methods for the comparison operator, including methods with union type arguments, such as:
>
> ```
> <(x::Union{Float16, Float32, Float64}, y::BigFloat)
> ```
> a method comparing fixed-width machine numbers with arbitrary precision numbers.

8.5 Composite Types: `struct`

A `struct` defines a concrete type as a collection of named fields.

```
abstract type Point end

mutable struct Point2D <: Point
    x :: Float64
    y :: Float64
end

mutable struct Point3D <: Point
    x :: Float64
    y :: Float64
    z :: Float64
end
```

As seen with expressions like `x = Int8(33)`, type names can serve as constructors:

```
p1 = Point2D(1.4, 3.5)
```

```
Point2D(1.4, 3.5)
```

```
p1 isa Point3D,  p1 isa Point2D,   p1 isa Point
```

```
(false, true, true)
```

The fields of a `struct` are accessed by name using the `.` operator.

```
p1.y
```

```
3.5
```

Because we declared our `struct` as `mutable`, we can modify the object `p1` by assigning new values to the fields.

```
p1.x = 3333.4
p1
```

```
Point2D(3333.4, 3.5)
```

The `dump()` function displays structure information for types and objects.

```
dump(Point3D)
```

```
mutable struct Point3D <: Point
x::Float64
y::Float64
z::Float64
```

```
 dump(p1)
```

```
Point2D
x: Float64 3333.4
y: Float64 3.5
```

8.6 Functions and *Multiple Dispatch*

> **i** Objects, Functions, and Methods
>
> In classical object-oriented languages (C++, Java, Python), methods are bound to objects.
> Julia takes a different approach: methods belong to functions, not to objects. Constructors (functions sharing a type's name that create instances of that type) are the only exception.
> When we define a new type, we can define functions specific to that type but we can also add additional methods to existing functions.
>
> - A single functions can have multiple methods for different argument types.
> - At call time, Julia selects the most specific method matching the concrete argument types *(multiple dispatch)*.
> - Core functions in Julia often have numerous predefined methods and third-party packages or user code can extend them adding additional methods.

We define a distance function with two methods:

```
function distance(p1::Point2D, p2::Point2D)
    sqrt((p1.x-p2.x)^2 + (p1.y-p2.y)^2)
end

function distance(p1::Point3D, p2::Point3D)
    sqrt((p1.x-p2.x)^2 + (p1.y-p2.y)^2 + (p1.z-p2.z)^2)
end
```

```
distance (generic function with 2 methods)
```

```
distance(p1, Point2D(2200, -300))
```

```
1173.3319266090054
```

As mentioned earlier, `methods()` shows the method table of a function:

```
methods(distance)
```

```
# 2 methods for generic function "distance" from Main.Notebook:
 [1] distance(p1::Main.Notebook.Point3D, p2::Main.Notebook.Point3D)
     @ ~/Julia/Book26/JuliaBook/chapters/types.qmd:264
 [2] distance(p1::Main.Notebook.Point2D, p2::Main.Notebook.Point2D)
     @ ~/Julia/Book26/JuliaBook/chapters/types.qmd:260
```

The `@which` macro, applied to a function call with concrete arguments, shows which method is selected for these arguments:

```
@which sqrt(3.3)
```

```
sqrt(x::Union{Float32, Float64})
     @ Base.Math math.jl:626
```

```
z = "Hello" * '!'
println(z)

@which "Hello" * '!'
```

```
Hello!
\*(s1::Union{AbstractChar, AbstractString}, ss::Union{AbstractChar, AbstractString}...)
     @ Base strings/basic.jl:262
```

Methods can also have abstract types as arguments:

```
"""
  Calculate the angle ϕ (in degrees) of the polar coordinates (2D) or
    spherical coordinates (3D) of a point
"""
function phi_angle(p::Point)
    atand(p.y, p.x)
end

phi_angle(p1)
```

```
0.0601593431937626
```

> **Tip**
>
> Text enclosed in *triple quotes* immediately before the function definition is automatically integrated into Julia's help database:
>
> ```
> ?phi_angle
> ```
>
> ```
> search: phi_angle angle
> ```
>
> Calculate the angle ϕ (in degrees) of the polar coordinates (2D) or spherical coordinates (3D) of a point

With *multiple dispatch*, the method is applied that is the most specific among all matching ones. Here is a function with several methods (all but the last written in short *assignment form*):

```
f(x::String, y::Number)  = "Args: String + Number"
f(x::String, y::Int64)   = "Args: String + Int64"
f(x::Number, y::Int64)   = "Args: Number + Int64"
f(x::Int64,  y:: Number) = "Args: Int64  + Number"
f(x::Number)             = "Arg: a Number"

function f(x::Number, y::Number, z::String)
    return "Arg: 2 × Number + String"
end
```

```
f (generic function with 6 methods)
```

The first two methods match; the second is chosen as it is more specific (`Int64 <: Number`):

```
f("Hello", 42)
```

```
"Args: String + Int64"
```

Ambiguities may arise if methods are defined poorly:

```
f(42, 42)
```

```
MethodError: f(::Int64, ::Int64) is ambiguous.

Candidates:
  f(x::Int64, y::Number)
    @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:322
  f(x::Number, y::Int64)
    @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:321

Possible fix, define
  f(::Int64, ::Int64)

Stacktrace:
 [1] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/types.qmd:336
Error showing value of type MethodError
StackOverflowError:
Stacktrace:
```

```
[1] show(io::IOContext{IOBuffer}, x::MethodError) (repeats 79983 times)
  @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:25
```

8.7 Parametric Numeric Types: `Rational` and `Complex`

- For rational numbers (fractions), Julia uses `//` as an infix constructor:

```
@show Rational(23, 17)    4//16 + 1//3;
```

```
Rational(23, 17) = 23//17
4 // 16 + 1 // 3 = 7//12
```

- The imaginary unit $\sqrt{-1}$ is denoted `im`

```
@show Complex(0.4)     23 + 0.5im/(1-2im);
```

```
Complex(0.4) = 0.4 + 0.0im
23 + (0.5im) / (1 - 2im) = 22.8 + 0.1im
```

Like `Point2D`, both `Rational` and `Complex` consist of two fields: numerator and denominator or real and imaginary parts.

However, the type of these fields is not completely fixed. `Rational` and `Complex` are *parametric* types.

```
x = 2//7
@show typeof(x);
```

```
typeof(x) = Rational{Int64}
```

```
y = BigInt(2)//7
@show typeof(y)    y^48;
```

```
typeof(y) = Rational{BigInt}
y ^ 48 = 281474976710656//367033682172941254412302110320336601888801
```

```
x = 1 + 2im
typeof(x)
```

```
Complex{Int64}
```

```
y = 1.0 + 2.0im
typeof(y)
```

```
ComplexF64 (alias for Complex{Float64})
```

The concrete types `Rational{Int64}`, `Rational{BigInt}`,..., `Complex{Int64}`, `Complex{Float64}`,... are subtypes of `Rational` and `Complex`, respectively.

```
Rational{BigInt} <: Rational
```

```
true
```

The definitions look roughly like this:

```
struct MyComplex{T<:Real} <: Number
    re::T
    im::T
end

struct MyRational{T<:Integer} <: Real
    num::T
    den::T
end
```

The first definition says:

- `MyComplex` has two fields `re` and `im`, both of the same type `T`.
- This type `T` must be a subtype of `Real`.
- `MyComplex` and all its variants like `MyComplex{Float64}` are subtypes of `Number`.

and the second says analogously:

- `MyRational` has two fields `num` and `den`, both of the same type `T`.
- This type `T` must be a subtype of `Integer`.
- `MyRational` and its variants are subtypes of `Real`.

Since $\mathbb{Q} \subset \mathbb{R}$ (or `Rational <: Real` in Julia notation), the components of a complex number can also be rational:

```
z = 3//4 + 5im
dump(z)
```

```
Complex{Rational{Int64}}
re: Rational{Int64}
num: Int64 3
den: Int64 4
im: Rational{Int64}
num: Int64 5
den: Int64 1
```

These structures are declared without the `mutable` attribute, making them *immutable*:

```
x = 2.2 + 3.3im
println("The real part is: $(x.re)")

x.re = 4.4
```

```
The real part is: 2.2
setfield!: immutable struct of type Complex cannot be changed
Stacktrace:
 [1] setproperty!(x::ComplexF64, f::Symbol, v::Float64)
   @ Base ./Base_compiler.jl:58
 [2] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/types.qmd:426
Error showing value of type ErrorException
StackOverflowError:
Stacktrace:
 [1] show(io::IOContext{IOBuffer}, x::ErrorException) (repeats 79983 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:25
```

This is standard practice. The object `9` (of type `Int64`) is also immutable. The following, of course, still works:

```
x += 2.2
```

```
4.4 + 3.3im
```

Here, a new object of type `Complex{Float64}` is created and `x` then references this new object.

The type system's flexibility invites experimentation. Here we define a `struct` that can contain either a machine number or a pair of integers:

```
struct MyParms{T <: Union{Float64, Tuple{Int64, Int64}}}
    param::T
end

p1 = MyParms(33.3)
p2 = MyParms( (2, 4) )

@show p1.param  p2.param;
```

```
p1.param = 33.3
p2.param = (2, 4)
```

8.8 Types as Objects

(1) Types are also objects. Each type is an instance of one of three "meta-types":

- `Union` (union types)
- `UnionAll` (parametric types)
- `DataType` (all concrete and other abstract types)

```
@show    23779 isa Int64       Int64 isa DataType;
```

```
23779 isa Int64 = true
Int64 isa DataType = true
```

```
@show  2im isa Complex         Complex isa UnionAll;
```

```
2im isa Complex = true
Complex isa UnionAll = true
```

```
@show  2im isa Complex{Int64}     Complex{Int64} isa DataType;
```

```
2im isa Complex{Int64} = true
Complex{Int64} isa DataType = true
```

These three concrete "meta-types" are, by the way, subtypes of the abstract "meta-type" `Type`.

```
subtypes(Type)
```

```
4-element Vector{Any}:
Core.TypeofBottom
DataType
Union
UnionAll
```

---

(2) Types can be assigned to a variable:

```
x3 = Float64
@show   x3(4)      x3 <: Real   x3==Float64  ;
```

```
x3(4) = 4.0
x3 <: Real = true
x3 == Float64 = true
```

> **Note**
>
> This demonstrates that Julia's style guidelines such as "Types and type variables start with uppercase letters, other variables and functions are written in lowercase." are conventions, not language-enforced rules.
>
> They should still be followed for readability.

Declaring such assignments with `const` creates a *type alias*.

```
const MyCmplxF64 = MyComplex{Float64}

z = MyCmplxF64(1.1, 2.2)
typeof(z)
```

```
MyComplex{Float64}
```

---

(3) Types can be function arguments.

```
function myf(x, S, T)
    if S <: T
        println("$S is subtype of $T")
    end
    return S(x)
```

```
end

z = myf(43, UInt16, Real)

@show z  typeof(z);
```

```
UInt16 is subtype of Real
z = 0x002b
typeof(z) = UInt16
```

To define this function with type signatures, we can write

```
function myf(x, S::Type, T::Type) ... end
```

However, the (equivalent) special syntax

```
function myf(x, ::Type{S}, ::Type{T}) where {S,T} ... end
```

is more common. Here we can also impose restrictions on the permissible values of the type variables `S` and `T` in the `where` clause.

How can we define a special method of `myf` that should only be called when `S` and `T` are equal to `Int64`? This is possible as follows:

```
function myf(x, ::Type{Int64}, ::Type{Int64}) ... end
```

`Type{Int64}` acts like a "meta-type", whose only instance is the type `Int64`.

(4) There are numerous functions with types as arguments. We have already seen `<:(T1, T2)`, `supertype(T)`, `supertypes(T)`, `subtypes(T)`. Other useful operations include `typejoin(T1,T2)` (next common ancestor in the type tree) and tests like `isconcretetype(T)`, `isabstracttype(T)`, `isstructtype(T)`.

8.9 Invariance of Parametric Types

Can non-concrete types also be used as parameters in parametric types? Are there `Complex{AbstractFloat}` or `Complex{Union{Float32, Int16}}`?

Yes, such types exist. They are concrete, and objects can be instantiated.

```
z5 = Complex{Integer}(2, 0x33)
dump(z5)
```

```
Complex{Integer}
re: Int64 2
im: UInt8 0x33
```

This is a heterogeneous composite type. Each component has an individual type `T`, for which `T<:Integer` holds.

Note that

```
Int64 <: Integer
```

```
true
```

but it does not hold that

```
Complex{Int64} <: Complex{Integer}
```

```
false
```

These types are both concrete. Therefore, they cannot stand in a sub/supertype relation to each other in Julia's type hierarchy. Julia's parametric types are in theoretical computer science terminology, **invariant**. (If `S<:T` implied `ParamType{S} <: ParamType{T}`, this would be **covariance**.)

8.10 Generic Functions

The usual (and in many cases recommended!) programming style in Julia is writing generic functions:

```
function fmm(x, y)
    return x * x * y
end
```

```
fmm (generic function with 1 method)
```

This function works with any types supporting the required operations.

```
fmm( Complex(2,3), 10),  fmm("Hello", '!')
```

```
(-50 + 120im, "HelloHello!")
```

Type annotations can restrict applicability or implement different methods for different types:

```
function fmm2(x::Number, y::AbstractFloat)
    return x * x * y
end

function fmm2(x::String, y::String)
    println("Sorry, I don't take strings!")
end


@show fmm2(18, 2.0)   fmm2(18, 2);
```

```
fmm2(18, 2.0) = 648.0
MethodError: no method matching fmm2(::Int64, ::Int64)
The function `fmm2` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  fmm2(::Number, ::AbstractFloat)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:596
  fmm2(::String, ::String)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:600

Stacktrace:
 [1] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/types.qmd:605
```

> ! Important
>
> **Explicit type annotations are almost always irrelevant for the speed of the code!**
> This is one of the most important *advantages* of Julia.
> Once a function is called for the first time with certain types, a specialized form of the function is generated and compiled for these argument types. Thus, generic functions are usually just as fast as the specialized functions one writes in other languages.
> Generic functions enable seamless integration across packages and support high-level abstraction.
> A simple example: The `Measurements.jl` package defines a new data type `Measurement`, a value with error, and the arithmetic of this type. Thus, generic functions work automatically:
>
> ```
> using Measurements
>
> x = 33.56±0.3
> y = 2.3±0.02
>
> fmm(x, y)
> ```
>
> ```
> 2590.0 ± 52.0
> ```

8.11 Type Parameters in Function Definitions: the `where` Clause

We want to write a function that works for **all complex integers** (and only these), e.g., an implementation of prime factorization in $\mathbb{Z}[i]$. The definition

```
function isprime(x::Complex{Integer}) ... end
```

does not give the desired result, as we saw in Section 8.9. The function would not work for an argument of type `Complex{Int64}`, since the latter is not a subtype of `Complex{Integer}`.

We must introduce a type variable. The `where` clause serves this purpose.

```
function isprime(x::Complex{T}) where {T<:Integer}
    ...
end
```

This is to be read as:

> "The argument `x` should be one of the types `Complex{T}`, where the type variable `T` can be any subtype of `Integer`."

Another example:

```
function kgV(x::Complex{T}, y::Complex{S}) where {T<:Integer, S<:Integer}
    ...
end
```

> The arguments x and y can have different types, each a subtype of `Integer`.

If there is only one `where` clause as in the last example, one can omit the curly braces and write

```
function isprime(x::Complex{T}) where T<:Integer
    ...
end
```

This can still be shortened to

```
function isprime(x::Complex{<:Integer})
    ...
end
```

These different variants can be confusing, but it is only syntax.

```
C1 = Complex{T} where {T<:Integer}
C2 = Complex{T} where T<:Integer
C3 = Complex{<:Integer}

C1 == C2 == C3
```

```
true
```

Short syntax for simple cases; extended syntax for complex variants.

Finally, note that `where T` is shorthand for `where T<:Any`, which introduces a completely unrestricted type variable. Thus, something like this is possible:

```
function fgl(x::T, y::T) where T
    println("Congratulations! x and y are of the same type!")
end
```

```
fgl (generic function with 1 method)
```

This method requires that both arguments have exactly the same type; otherwise, any type is accepted.

```
fgl(33, 44)
```

```
Congratulations! x and y are of the same type!
```

```
fgl(33, 44.0)
```

```
MethodError: no method matching fgl(::Int64, ::Float64)
The function `fgl` exists, but no method is defined for this combination of argument types.

Closest candidates are:
  fgl(::T, ::T) where T
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:729

Stacktrace:
 [1] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/types.qmd:741
Error showing value of type MethodError
StackOverflowError:
Stacktrace:
 [1] show(io::IOContext{IOBuffer}, x::MethodError) (repeats 79983 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/types.qmd:25
```

9. Example: The Parametric Data Type `PComplex`

We want to introduce a new numeric type **complex numbers in polar representation** $z = re^{i\phi} = (r, \phi)$.

- The type should integrate into the type hierarchy as a subtype of 'Number'.
- r and ϕ should be floating point numbers. (Unlike complex numbers in 'Cartesian' coordinates, restricting to integer values of r or ϕ makes little mathematical sense.)

9.1 The Definition of `PComplex`

A first attempt could look like this:

```
struct PComplex{T <: AbstractFloat} <: Number
    r :: T
    ϕ :: T
end

z1 = PComplex(-32.0, 33.0)
z2 = PComplex{Float32}(12, 13)
@show z1 z2;
```

```
z1 = PComplex(-32.0, 33.0)
z2 = PComplex(12.0, 13.0)
```

Julia automatically provides *default constructors*:

- The constructor `PComplex` infers type `T` from the arguments, and
- Constructors like `PComplex{Float64}` accept explicit type specifications. Arguments are converted to the requested type.

We now want the constructor to do even more. In the polar representation, we want $0 \leq r$ and $0 \leq \phi < 2\pi$ to hold.

If the passed arguments do not satisfy this, they should be recalculated accordingly.
To this end, we define an *inner constructor* that replaces the *default constructor*.

- An *inner constructor* is a function within the `struct` definition.
- In an *inner constructor*, one can use the special function `new`, which acts like the *default constructor*.

```
struct PComplex{T <: AbstractFloat} <: Number
    r :: T
    ϕ :: T

    function PComplex{T}(r::T, ϕ::T) where T<:AbstractFloat
        if r<0              # flip the sign of r and correct phi
            r = -r
            ϕ += π
        end
        if r==0 ϕ=0 end  # normalize r=0 case to phi=0
        ϕ = mod(ϕ, 2π)   # map phi into interval [0,2pi)
        new(r, ϕ)        # new() is special function,
    end                  #   available only inside inner constructors

end
```

```
z1 = PComplex{Float64}(-3.3, 7π+1)
```

```
PComplex{Float64}(3.3, 1.0)
```

However, explicitly specifying an *inner constructor* has a consequence: Julia's *default constructors* are no longer available.

The constructor without explicit type specification, which infers the type from the arguments, is also needed:

```
PComplex(r::T, ϕ::T) where {T<:AbstractFloat} = PComplex{T}(r,ϕ)

z2 = PComplex(2.0, 0.3)
```

```
PComplex{Float64}(2.0, 0.3)
```

9.2 A New Notation

Julia uses `//` as an infix constructor for the type `Rational`. We want something equally nice.

In electronics/electrical engineering, AC quantities are described by complex numbers. A representation of complex numbers by "magnitude" and "phase" is common and is often represented in so-called phasor form:

$$z = r\angle\varphi = 3.4\angle 45°$$

where the angle is usually noted in degrees.

> **i** Possible Infix Operators in Julia
>
> In Julia, a large number of Unicode characters are reserved for use as operators. The definitive list is in the parser source code.
>
> Details will be discussed in a later chapter.

The angle bracket symbol ∠ is not available as a Julia operator. We use ⩤ as an alternative, entered as `\lessdot<Tab>`.

```
⩤(r::Real, ϕ::Real) = PComplex(r, π*ϕ/180)

z3 = 2. ⩤ 90.
```

```
PComplex{Float64}(2.0, 1.5707963267948966)
```

(The type annotation `Real` instead of `AbstractFloat` anticipates further constructors. Currently, the operator ⩤ works only with `Float64`.)

Of course, we also want the output to look nice. Details can be found in the documentation.

```
using Printf

function Base.show(io::IO, z::PComplex)
    # print phase in degrees, rounded to one decimal place
    p = z.ϕ * 180/π
    sp = @sprintf "%.1f" p
    print(io, z.r, "⩤", sp, '°')
end

@show z3;
```

```
z3 = 2.0⩤90.0°
```

9.3 Methods for PComplex

For our type to be a proper member of the family of types derived from `Number`, additional functionality is required: arithmetic operations, comparison operators, and conversions must all be defined.

We focus on multiplication and square root operations.

> **Modules**
>
> - Adding methods to existing functions requires using their fully qualified names.
> - All objects belong to a namespace or `module`.
> - Most basic functions belong to `Base`, which is loaded automatically.
> - Without user-defined modules, definitions reside in `Main`.
> - The macro `@which` applied to a name shows its defining module.

```
f(x) = 3x^3
@which f
```

```
Main.Notebook
```

```
wp = @which +
ws = @which(sqrt)
println("Module for addition: $wp, Module for sqrt: $ws")
```

```
Module for addition: Base, Module for sqrt: Base
```

```
sqrt_polar(z::PComplex) = PComplex(sqrt(z.r), z.ϕ / 2)
```

```
sqrt_polar (generic function with 1 method)
```

The function `sqrt()` already has some methods:

```
length(methods(sqrt))
```

```
19
```

Adding one more method:

```
Base.sqrt(z::PComplex) = sqrt_polar(z)

length(methods(sqrt))
```

```
20
```

```
sqrt(z2)
```

```
1.4142135623730951∠8.6°
```

For multiplication:

```
Base.:*(x::PComplex, y::PComplex) = PComplex(x.r * y.r, x.ϕ + y.ϕ)

@show z1 * z2;
```

```
z1 * z2 = 6.6∠74.5°
```

(Since `:` is not a valid identifier character, it must be qualified with `Base.`)

However, multiplication with other numeric types is not yet supported. Many corresponding methods could be defined, but Julia provides another mechanism for *numeric types* that simplifies this:

9.4 Type Promotion and Conversion

Julia supports freely mixing various numeric types:

```
1//3 + 5 + 5.2 + 0xff
```

```
265.53333333333336
```

Among the numerous methods defined for `+` and `*`, we find a catch-all definition:

```
+(x::Number, y::Number) = +(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
```

(The three dots form the splat operator, which decomposes the tuple returned by `promote()` into its components.)

Since the method with the types `(Number, Number)` is very general, it is only used when more specific methods do not apply.

What happens here?

9.4.1 The Function `promote(x,y,...)`

This function attempts to convert all arguments to a common type that can represent all values (as precisely as possible).

```
promote(12, 34.555, 77/99, 0xff)
```

```
(12.0, 34.555, 0.7777777777777778, 255.0)
```

```
z = promote(BigInt(33), 27)
@show z typeof(z);
```

```
z = (33, 27)
typeof(z) = Tuple{BigInt, BigInt}
```

The function `promote()` uses two helpers, the functions `promote_type(T1, T2)` and `convert(T, x)`

As usual in Julia, we can extend this mechanism with our own custom *promotion rules* and `convert(T,x)` methods.

9.4.2 The Function `promote_type(T1, T2,...)`

It determines to which type the conversion should take place. Arguments are types, not values.

```
@show promote_type(Rational{Int64}, ComplexF64, Float32);
```

```
promote_type(Rational{Int64}, ComplexF64, Float32) = ComplexF64
```

9.4.3 The Function `convert(T,x)`

The methods of `convert(T, x)` convert `x` into an object of type `T`. Such a conversion should be lossless.

```
z = convert(Float64, 3)
```

```
3.0
```

```
z = convert(Int64, 23.00)
```

```
23
```

```
z = convert(Int64, 2.3)
```

```
InexactError: Int64(2.3)
Stacktrace:
 [1] Int64
   @ ./float.jl:923 [inlined]
 [2] convert(::Type{Int64}, x::Float64)
   @ Base ./number.jl:7
 [3] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/pcomplex.qmd:312
```

The special role of `convert()` is that it is called implicitly at various points:

The following language constructs call convert:

- Assigning to an array converts to the array's element type.
- Assigning to a field of an object converts to the declared type of the field.
- Constructing an object with new converts to the object's declared field types.
- Assigning to a variable with a declared type (e.g. local x::T) converts to that type.
- A function with a declared return type converts its return value to that type.

– and of course in `promote()`
For user-defined types, `convert()` can be extended with custom methods.
Within the `Number` hierarchy, a generic method handles conversions:

```
convert(::Type{T}, x::Number) where {T<:Number} = T(x)
```

Therefore: If a type `T<:Number` has a constructor `T(x)` accepting a numeric argument, this constructor is automatically used for conversions. (More specific methods for `convert()` can also be defined and will take priority.)

9.4.4 Further Constructors for `PComplex`

```
## (a) Arbitrary real types for r and ϕ (e.g., integers, rationals)

PComplex{T}(r::T1, ϕ::T2) where {T<:AbstractFloat, T1<:Real, T2<: Real} =
    PComplex{T}(convert(T, r), convert(T, ϕ))

PComplex(r::T1, ϕ::T2) where {T1<:Real, T2<: Real} =
    PComplex{promote_type(Float64, T1, T2)}(r, ϕ)

## (b) For conversion from reals: constructor with
##      only one argument r

PComplex{T}(r::S) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(convert(T, r), convert(T, 0))

PComplex(r::S)  where {S<:Real} =
    PComplex{promote_type(Float64, S)}(r, 0.0)

## (c) Conversion Complex → PComplex

PComplex{T}(z::Complex{S}) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(abs(z), angle(z))

PComplex(z::Complex{S}) where {S<:Real} =
    PComplex{promote_type(Float64, S)}(abs(z), angle(z))
```

```
PComplex
```

Testing the new constructors:

```
3//5 ∢ 45,  PComplex(Complex(1,1)),   PComplex(-13)
```

```
(0.6∢45.0°, 1.4142135623730951∢45.0°, 13.0∢180.0°)
```

Promotion rules are needed to determine the result type of `promote(x::T1, y::T2)`. This mechanism extends `promote_type()` with the necessary methods.

9.4.5 *Promotion rules* for `PComplex`

```
Base.promote_rule(::Type{PComplex{T}}, ::Type{S}) where {T<:AbstractFloat,S<:Real} =
    PComplex{promote_type(T,S)}
```

```
Base.promote_rule(::Type{PComplex{T}}, ::Type{Complex{S}}) where
    {T<:AbstractFloat,S<:Real} = PComplex{promote_type(T,S)}
```

1. **Rule:** When a `PComplex{T}` and an `S<:Real` are combined, both convert to `PComplex{U}`, where `U` is the promoted type of `S` and `T`.
2. **Rule** When a `PComplex{T}` and a `Complex{S}` are combined, both convert to `PComplex{U}`, where `U` is the promoted type of `S` and `T`.

We can now multiply with arbitrary numeric types:

```
z3, 3z3
```

```
(2.0∠90.0°, 6.0∠90.0°)
```

```
(3.0+2im) * (12∠30.3),  12sqrt(z2)
```

```
(43.26661530556787∠64.0°, 16.970562748477143∠8.6°)
```

Summary: our type `PComplex`

```
struct PComplex{T <: AbstractFloat} <: Number
    r :: T
    ϕ :: T

    function PComplex{T}(r::T, ϕ::T) where T<:AbstractFloat
        if r<0              # flip the sign of r and correct phi
            r = -r
            ϕ += π
        end
        if r==0 ϕ=0 end  # normalize r=0 case to phi=0
        ϕ = mod(ϕ, 2π)   # map phi into interval [0,2pi)
        new(r, ϕ)        # new() is special function,
    end                  #   available only inside inner constructors

end

# additional constructors
PComplex(r::T, ϕ::T) where {T<:AbstractFloat} = PComplex{T}(r,ϕ)


PComplex{T}(r::T1, ϕ::T2) where {T<:AbstractFloat, T1<:Real, T2<: Real} =
    PComplex{T}(convert(T, r), convert(T, ϕ))

PComplex(r::T1, ϕ::T2) where {T1<:Real, T2<: Real} =
    PComplex{promote_type(Float64, T1, T2)}(r, ϕ)


PComplex{T}(r::S) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(convert(T, r), convert(T, 0))

PComplex(r::S)  where {S<:Real} =
    PComplex{promote_type(Float64, S)}(r, 0.0)


PComplex{T}(z::Complex{S}) where {T<:AbstractFloat, S<:Real} =
    PComplex{T}(abs(z), angle(z))

PComplex(z::Complex{S}) where {S<:Real} =
    PComplex{promote_type(Float64, S)}(abs(z), angle(z))

# nice input
∢(r::Real, ϕ::Real) = PComplex(r, π*ϕ/180)

# nice output
using Printf

function Base.show(io::IO, z::PComplex)
    # print phase in degrees, rounded to one decimal place
    p = z.ϕ * 180/π
    sp = @sprintf "%.1f" p
    print(io, z.r, "∢", sp, '°')
end

# arithmetic
Base.sqrt(z::PComplex) = PComplex(sqrt(z.r), z.ϕ / 2)

Base.:*(x::PComplex, y::PComplex) = PComplex(x.r * y.r, x.ϕ + y.ϕ)

# promotion rules
Base.promote_rule(::Type{PComplex{T}}, ::Type{S}) where
    {T<:AbstractFloat,S<:Real} = PComplex{promote_type(T,S)}

Base.promote_rule(::Type{PComplex{T}}, ::Type{Complex{S}}) where
    {T<:AbstractFloat,S<:Real} = PComplex{promote_type(T,S)}
```

10. Functions and Operators

Functions process their arguments to produce and return a result when called.

10.1 Forms of function definitions

I. Block form: `function ... end`

```
function hyp(x,y)
    sqrt(x^2+y^2)
end
```

```
hyp (generic function with 1 method)
```

II. Single-line form

```
hyp(x, y) = sqrt(x^2 + y^2)
```

```
hyp (generic function with 1 method)
```

III. Anonymous functions

```
(x, y) -> sqrt(x^2 + y^2)
```

```
#2 (generic function with 1 method)
```

10.1.1 Block Form and `return` Statement

- With `return`, function execution terminates and control returns to the calling context.
- Without `return`, the value of the last expression is returned as the function value.

The two definitions

```
function xsinrecipx(x)
    if x == 0
        return 0.0
    end
    return x * sin(1/x)
end
```

and the equivalent version without explicit `return` in the last line:

```
function xsinrecipx(x)
    if x == 0
        return 0.0
    end
    x * sin(1/x)
end
```

are therefore equivalent.

- A function that returns `nothing` (*void functions* in C) returns a `nothing` value of type `Nothing`. (Just as a `Bool` object has two values, `true` and `false`, a `Nothing` object has only one: `nothing`.)
- An empty `return` statement is equivalent to `return nothing`.

```
function fn(x)
    println(x)
    return
end
```

```
a = fn(2)
```

```
2
```

```
a
```

```
@show a typeof(a);
```

```
a = nothing
typeof(a) = Nothing
```

10.1.2 Single-liner Form

The single-liner form looks like a simple assignment:

```
hyp(x, y) = sqrt(x^2 + y^2)
```

Julia provides two ways to combine multiple statements into a block that can stand in place of a single statement:

- `begin ... end` block
- Parenthesized statements separated by semicolons.

In both cases, the value of the block is the value of the last statement.

Thus, the following also works:

```
hyp(x, y) = (z = x^2; z += y^2; sqrt(z))
```

and

```
hyp(x, y) = begin
                z = x^2
                z += y^2
                sqrt(z)
            end
```

10.1.3 Anonymous Functions

Anonymous functions can be "rescued from anonymity" by assigning them a name:

```
hyp = (x,y) -> sqrt(x^2 + y^2)
```

Their actual application is in calling a *(higher order)* function that expects a function as an argument.

Typical applications include `map(f, collection)`, which applies a function to every element of a collection. Julia also supports `map(f, collection1, collection2)` with multiple collections:

```
map( (x,y) -> sqrt(x^2 + y^2), [3, 5, 8], [4, 12, 15])
```

```
3-element Vector{Float64}:
5.0
13.0
17.0
```

```
map( x->3x^3, 1:8 )
```

```
8-element Vector{Int64}:
3
24
81
192
375
648
1029
1536
```

Another example is `filter(test, collection)`, where a test is a function that returns a `Bool`.

```
filter(x -> ( x%3 == 0 && x%5 == 0), 1:100  )
```

```
6-element Vector{Int64}:
15
30
45
60
75
90
```

10.2 Argument Passing

- When calling a function, Julia does not copy objects passed as arguments. Function arguments refer to the original objects. Julia calls this concept *pass_by_sharing*.
- Consequently, functions can modify their arguments if they are mutable (e.g., `Vector` or `Array`).
- By convention, functions that modify their arguments end with an exclamation mark. The modified argument is typically the first argument and is also returned.

```
V = [1, 2, 3]

W = fill!(V, 17)
                          # '===' tests for identity
@show  V  W  V===W;       # V and W refer to the same object
```

```
V = [17, 17, 17]
W = [17, 17, 17]
V === W = true
```

```
function fill_first!(V, x)
        V[1] = x
        return V
end

U = fill_first!(V, 42)

@show V   U   V===U;
```

```
V = [42, 17, 17]
U = [42, 17, 17]
V === U = true
```

10.3 Function Argument Variants

- There are positional arguments (1st argument, 2nd argument, ...) and *keyword* arguments, which must be addressed by name when calling.
- Both positional and *keyword* arguments can have *default* values. These arguments can be omitted when calling.
- The order of declaration must be:
  1. Positional arguments without default values,
  2. Positional arguments with default values,
  3. — semicolon —,
  4. comma-separated list of keyword arguments (with or without default values)
- When calling, keyword arguments can appear in any order at any position. They can be separated from positional arguments with a semicolon, but this is optional.

```
fa(x, y=42; a) = println("x=$x, y=$y, a=$a")

fa(6, a=4, 7)
```

```
fa(6, 7; a=4)
fa(a=-2, 6)
```

```
x=6, y=7, a=4
x=6, y=7, a=4
x=6, y=42, a=-2
```

A function with only *keyword* arguments is declared as follows:

```
fkw(; x=10, y) = println("x=$x, y=$y")

fkw(y=2)
```

```
x=10, y=2
```

10.4 Functions are just Objects

- Functions can be assigned to variables

```
f2 = sqrt
f2(2)
```

```
1.4142135623730951
```

- Functions can be passed as arguments to other functions.

```
# naive Riemann integration example

function Riemann_integrate(f, a, b; NInter=1000)
    delta = (b-a)/NInter
    s = 0
    for i in 0:NInter-1
        s += delta * f(a + delta/2 + i * delta)
    end
    return s
end


Riemann_integrate(sin, 0, π)
```

```
2.0000008224672694
```

- They can be created by functions and returned as results.

```
function generate_add_func(x)
    function addx(y)
        return x+y
    end
    return addx
end
```

```
generate_add_func (generic function with 1 method)
```

```
h =  generate_add_func(4)
```

```
(::Main.Notebook.var"#addx#generate_add_func##0"{Int64}) (generic function with 1
method)
```

```
h(1)
```

```
5
```

```
h(2), h(10)
```

```
(6, 14)
```

The above function `generate_add_func()` can also be defined more briefly. The inner function name `addx` is local and inaccessible outside. An anonymous function can be used instead.

```
generate_add_func(x) = y -> x + y
```

```
generate_add_func (generic function with 1 method)
```

10.5 Function Composition: the Operators ∘ and ▷

- Function composition can also be written with the ∘ operator (`\circ + Tab`)

$$(f \circ g)(x) = f(g(x))$$

```
(sqrt ∘ + )(9, 16)
```

```
5.0
```

```
f = cos ∘ sin ∘ (x->2x)
f(.2)
```

```
0.9251300429004277
```

```
@show map(uppercase ∘ first, ["one", "a", "green", "leaves"]);
```

```
map(uppercase ∘ first, ["one", "a", "green", "leaves"]) = ['O', 'A', 'G', 'L']
```

- There is also an operator with which functions can act "from the right" and be composed (*piping*)

```
25 ▷ sqrt
```

```
5.0
```

```
1:10 ▷ sum ▷ sqrt
```

```
7.416198487095663
```

- These operators can also be broadcast (see Section 12.7). A vector of functions is applied element-wise to a vector of arguments:

```
["a", "list", "of", "strings"] .▷ [length, uppercase, reverse, titlecase]
```

```
4-element Vector{Any}:
1
"LIST"
"fo"
"Strings"
```

10.6 The do Notation

A syntactic peculiarity for defining anonymous functions as arguments of other functions is the `do` notation.

Let `higherfunc(f, a, ...)` be a function whose first argument is a function.

The function can be called without the first argument, with the function body defined in a following `do` block:

```
higherfunc(a, b) do x, y
   body of f(x,y)
end
```

Using `Riemann_integrate()` as an example, this looks like this:

```
# this is the same as Riemann_integrate(x->x^2, 0, 2)

Riemann_integrate(0, 2) do x x^2 end
```

```
2.6666659999999993
```

The `do` notation is especially useful for complex function bodies, such as this integrand defined in multiple steps:

```
r = Riemann_integrate(0, π) do x
        z1 = sin(x)
        z2 = log(1+x)
        if x > 1
            return z1^2
        else
            return 1/z2^2
        end
    end
```

```
1578.9022037353475
```

10.7 Function-like Objects

By defining a method for a type, objects become *callable* like functions.

```
# struct stores coefficients of a second-degree polynomial
struct Poly2Grad
    a0::Float64
    a1::Float64
    a2::Float64
end

p1 = Poly2Grad(2,5,1)
p2 = Poly2Grad(3,1,-0.4)
```

```
Poly2Grad(3.0, 1.0, -0.4)
```

The following method makes this structure callable:

```
function (p::Poly2Grad)(x)
    p.a2 * x^2 + p.a1 * x + p.a0
end
```

Objects can now be used like functions:

```
@show p2(5)  p1(-0.7)  p1;
```

```
p2(5) = -2.0
p1(-0.7) = -1.0100000000000002
p1 = Poly2Grad(2.0, 5.0, 1.0)
```

10.8 Operators and Special Forms

- Infix operators such as `+`, `*`, `>`, `∈` are functions.

```
+(3, 7)
```

```
10
```

```
f = +
```

```
+ (generic function with 191 methods)
```

```
f(3, 7)
```

```
10
```

- Constructions like `x[i]`, `a.x`, `[x; y]` are converted by the parser to function calls.

x[i]	getindex(x, i)
x[i] = z	setindex!(x, z, i)
a.x	getproperty(a, :x)
a.x = z	setproperty!(a, :x, z)
[x; y;...]	vcat(x, y, ...)

(The colon before a variable makes it into a symbol.)

> **i** Note
>
> For these functions, too, van be extended/overwritten by new methods. For example, for a custom type, setting a field (`setproperty!()`) could check the validity of the value or trigger further actions.
> In principle, `get/setproperty` can also do things that have nothing to do with an actually existing field of the structure.

10.9 Update Form

All arithmetic infix operators have an update form: The expression

```
x = x ⊙ y
```

can also be written as

```
x ⊙= y
```

Both forms are semantically equivalent: a new object created on the right is assigned to `x`.
Memory- and time-efficient *in-place updates* of arrays use explicit indexing:

```
for i in eachindex(x)
    x[i] += y[i]
end
```

or semantically equivalent broadcast form (see Section 12.7):

```
x .= x .+ y
```

10.10 Operator Precedence and Associativity

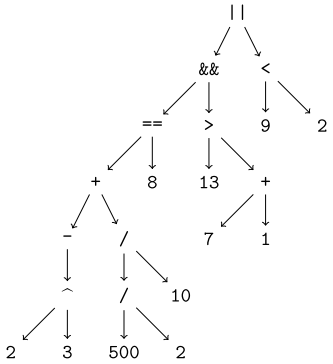
Expressions like

```
 -2^3+500/2/10==8 && 13 > 7 + 1 || 9 < 2
```

```
false
```

are converted by the parser into a tree structure:

```
using TreeView

walk_tree(Meta.parse("-2^3+500/2/10==8 && 13 > 7 + 1 || 9 < 2"))
```




```
Error showing value of type TreeView.LabelledTree
StackOverflowError:
Stacktrace:
 [1] show(io::IOContext{IOBuffer}, x::LabelledTree) (repeats 79983 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/9_functs.qmd:26
```

- Expression evaluation is governed by
  - precedence and
  - associativity.
- Precedence determines which operators bind more tightly, such as multiplication before addition.
- Associativity determines the evaluation order for operators of equal precedence.
- Complete documentation

10.10.1 Associativity

Addition/subtraction and multiplication/division have equal precedence and are left-associative (evaluated left-to-right):

```
200/5/2      # evaluated left to right as (200/5)/2
```

```
20.0
```

```
200/2*5      # evaluated left to right as (200/2)*5
```

```
500.0
```

Assignments like `=`, `+=`, `*=`,... are of equal rank and right-associative.

```
x = 1
y = 10

# evaluated right to left:  x += (y += (z = (a = 20)))

x += y +=  z = a = 20

@show x y z a;
```

```
x = 31
y = 30
z = 20
a = 20
```

Julia provides functions to query associativity. These functions are not exported from `Base`, so the module name must be specified.

```
for i in (:/, :+=, :(=), :^)
    a = Base.operator_associativity(i)
    println("Operation $i is  $(a)-associative")
end
```

```
Operation / is left-associative
Operation += is right-associative
Operation = is right-associative
Operation ^ is right-associative
```

Thus, the power operator is right-associative:

```
2^3^2    # right-associative,  = 2^(3^2)
```

```
512
```

10.10.2 Precedence

- Julia assigns operator precedence levels from 1 to 17:

```
for i in (:+, :-, :*, :/, :^, :(=))
    p = Base.operator_precedence(i)
    println("Precedence of  $i = $p")
end
```

```
Precedence of + = 11
Precedence of - = 11
Precedence of * = 12
Precedence of / = 12
Precedence of ^ = 15
Precedence of = = 1
```

- Precedence 11 < 12 explains why multiplication/division bind tighter than addition/subtraction.
- The power operator ^ has higher precedence.
- Assignments have the lowest precedence.

```
#   assignment has smallest precedence, therefore evaluation as  x = (3 < 4)

x = 3 < 4
x
```

```
true
```

```
(y = 3) < 4   # parentheses override any precedence
y
```

```
3
```

Returning to the example above:

```
-2^3+500/2/10==8 && 13 > 7 + 1 || 9 < 2
```

```
false
```

```
for i ∈ (:^, :+, :/, :(==), :&&, :>, :|| )
    print(i, " ")
    println(Base.operator_precedence(i))
end
```

```
^ 15
+ 11
/ 12
== 7
&& 6
> 7
|| 5
```

These rules evaluate the expression as:

```
((-(2^3)+((500/2)/10)==8) && (13 > (7 + 1))) || (9 < 2)
```

```
false
```

(as shown in the parse tree above).
So the precedence is:

Power > Multiplication/Division > Addition/Subtraction > Comparisons > logical && > logical || > assignment

Thus, an expression like

```
a = x <= y + z && x > z/2
```

is sensibly evaluated as `a = ((x <= (y+z)) && (x < (z/2)))`

- A special case is still
  - unary operators, in particular `+` and `-` as signs
  - *juxtaposition*, i.e., numbers directly before variables or parentheses without `*` symbol

  Both have precedence even before multiplication and division.

> **!** Important
>
> Therefore, the meaning of expressions changes when one applies *juxtaposition*:
>
> ```
> 1/2*π,  1/2π
> ```
>
> ```
> (1.5707963267948966, 0.15915494309189535)
> ```

- Compared to the power operator `^` (see https://discourse.julialang.org/t/confused-about-operator-precedence-for-2-3x/8214/7):

  Unary operators, including juxtaposition, bind tighter than ^ on the right but looser on the left.

Examples:

```
-2^2    # -(2^2)
```

```
-4
```

```
x = 5
2x^2    # 2(x^2)
```

```
50
```

```
2^-2    # 2^(-2)
```

```
0.25
```

```
2^2x  # 2^(2x)
```

```
1024
```

- Function application `f(...)` has precedence over all operators

```
sin(x)^2 === (sin(x))^2   # not sin(x^2)
```

```
true
```

10.10.3 Additional Operators

The Julia parser assigns precedence to numerous Unicode characters in advance, so that these characters can be used as operators by packages and self-written code.

Thus, for example,

```
∧ ⊗ ⊘ ⊙ ⊚ ⊛ ⊠ ⊡ ⊓ ∗ ∙ ∤ ⅋ ≀ ⊼ ⋄ ⋆ ⋇ ⋉ ⋊ ⋋ ⋌ ⋏ ⋒ ⟑ ⦸ ⦼ ⦾ ⦿ ⧶ ⧷ ⨇ ⨰ ⨱ ⨲ ⨳ ⨴ ⨵ ⨶ ⨷ ⨸ ⨻ ⨼
⨽ ⩀ ⩃ ⩄ ⩋ ⩍ ⩎ ⩑ ⩓ ⩕ ⩘ ⩚ ⩜ ⩞ ⩟ ⩠ ⫛
```

have precedence 12 like multiplication/division (and are left-associative like these) and for example

```
 ⊕ ⊖ ⊞ ⊟ |++| ∪ ∨ ⊔ ± ∓ ∔ ∸ ≏ ⊎ ⊻ ⊽ ⋎ ⋓ ⧺ ⧻ ⨈ ⨢ ⨣ ⨤ ⨥ ⨦ ⨧ ⨨ ⨩ ⨪ ⨫ ⨬ ⨭ ⨮ ⨹ ⨺ ⩁ ⩂ ⩅ ⩊ ⩌
⩏ ⩐ ⩒ ⩔ ⩖ ⩗
```

have precedence 11 like addition/subtraction.

11. Containers

Julia offers a wide selection of container types with largely similar interfaces. This chapter introduces `Tuple`, `Range`, and `Dict`; the next chapter covers `Array`, `Vector`, and `Matrix`.

These containers are:

- **iterable:** You can iterate over the elements of the container:

```
for x ∈ container ... end
```

- **indexable:** You can access elements via their index:

```
x = container[i]
```

and some are also

- **mutable:** You can add, remove, and modify elements.

Furthermore, there are several common functions, e.g.,

- `length(container)` — number of elements
- `eltype(container)` — element type
- `isempty(container)` — test if container is empty
- `empty!(container)` — empties the container (if mutable)

11.1 Tuples

A tuple is an immutable container of elements. You cannot add new elements or change existing values.

```
t = (33, 4.5, "Hello")

@show   t[2]              # indexable

for i ∈ t println(i) end   # iterable
```

```
t[2] = 4.5
33
4.5
Hello
```

A tuple is an **inhomogeneous** type. Each element has its own type, which is reflected in the tuple's type:

```
typeof(t)
```

```
Tuple{Int64, Float64, String}
```

Tuples are frequently used as function return values to return more than one object.

```
# Integer division and remainder:
# Assign quotient and remainder to variables `q` and `r`:

q, r = divrem(71, 6)
@show  q  r;
```

```
q = 11
r = 5
```

Parentheses can be omitted in certain constructs. This *implicit tuple packing/unpacking* is commonly used in multiple assignments:

```
x, y, z = 12, 17, 203
```

```
(12, 17, 203)
```

```
y
```

```
17
```

Some functions require tuples as arguments or always return tuples. A single-element tuple is written as:

```
x = (13,)         # a 1-element tuple
```

```
(13,)
```

The comma - not the parentheses - makes the tuple.

```
x= (13)         # not a tuple
```

```
13
```

11.2 Ranges

We have already used *range* objects in numerical `for` loops.

```
r = 1:1000
typeof(r)
```

```
UnitRange{Int64}
```

There are various *range* types. `UnitRange`, for example, is a *range* with step size 1. Their constructors are typically all named `range()`.

The colon is a special syntax.

- `a:b` is parsed as `range(a, b)`
- `a:b:c` is parsed as `range(a, c, step=b)`

Ranges are iterable, immutable, and indexable.

```
(3:100)[20]   # the 20th element
```

```
22
```

Recall the semantics of the `for` loop: `for i in 1:1000` does **not** mean 'increment the loop variable `i` by one each iteration'; **rather**, it means 'successively assign the values 1, 2, 3, ..., 1000 to the loop variable from the container'.

Creating this container explicitly would be very inefficient.

- *Ranges* are "lazy" vectors never stored as concrete lists. This makes them ideal as `for` loop iterators: memory-efficient and fast.
- They are "recipes" or generators that respond to the query "Give me your next element!".
- In fact, the supertype `AbstractRange` is a subtype of `AbstractVector`.

The macro `@allocated` outputs how many bytes of memory were allocated during the evaluation of an expression.

```
@allocated r = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

```
224
```

```
@allocated r = 1:20
```

```
32
```

The `collect()` function converts a range to a concrete vector.

```
collect(20:-3:1)
```

```
7-element Vector{Int64}:
20
17
14
11
8
5
2
```

Quite useful, e.g., when preparing data for plotting, is the *range* type `LinRange`.

```
LinRange(2, 50, 300)
```

```
300-element LinRange{Float64, Int64}:
2.0, 2.16054, 2.32107, 2.48161, 2.64214, …, 49.5184, 49.6789, 49.8395, 50.0
```

`LinRange(start, stop, n)` generates `n` equidistant values from start to stop. Use `collect()` to obtain the corresponding vector if needed.

11.3 Dictionaries

- *Dictionaries* (also known as associative arrays or lookup tables) are special containers.
- Whereas vector entries are addressed by integer indices: `v[i]`; dictionary entries are addressed by more general *keys*.
- A dictionary is a collection of *key-value* pairs with parameterized type `Dict{S,T}`, where `S` is the key type and `T` is the value type.

Create a dictionary explicitly:

```
# Population in 2020 in millions, source: wikipedia

Ppl = Dict("Berlin" => 3.66,  "Hamburg" => 1.85,
           "München" => 1.49, "Köln" => 1.08)
```

```
Dict{String, Float64} with 4 entries:
"München" => 1.49
"Köln" => 1.08
"Berlin" => 3.66
"Hamburg" => 1.85
```

```
typeof(Ppl)
```

```
Dict{String, Float64}
```

and indexed with the *keys*:

```
Ppl["Berlin"]
```

```
3.66
```

Querying a non-existent *key* throws an error.

```
Ppl["Leipzig"]
```

```
KeyError: key "Leipzig" not found
Stacktrace:
 [1] getindex(h::Dict{String, Float64}, key::String)
   @ Base ./dict.jl:477
 [2] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/6_ArraysEtcP1.qmd:179
```

Check beforehand with `haskey()`...

```
haskey(Ppl, "Leipzig")
```

```
false
```

Or use `get(dict, key, default)`, which returns the default value instead of throwing an error.

```
@show get(Ppl, "Leipzig", -1)   get(Ppl, "Berlin", -1);
```

```
get(Ppl, "Leipzig", -1) = -1
get(Ppl, "Berlin", -1) = 3.66
```

You can also request all `keys` and `values` as special containers.

```
keys(Ppl)
```

```
KeySet for a Dict{String, Float64} with 4 entries. Keys:
"München"
"Köln"
"Berlin"
"Hamburg"
```

```
values(Ppl)
```

```
ValueIterator for a Dict{String, Float64} with 4 entries. Values:
1.49
1.08
3.66
1.85
```

Iterate over the `keys`...

```
for i in keys(Ppl)
    n = Ppl[i]
    println("The city $i has $n million inhabitants.")
end
```

```
The city München has 1.49 million inhabitants.
The city Köln has 1.08 million inhabitants.
The city Berlin has 3.66 million inhabitants.
The city Hamburg has 1.85 million inhabitants.
```

Or iterate directly over `key-value` pairs.

```
for (city, pop) ∈ Ppl
    println("$city : $pop  Million.")
end
```

```
München : 1.49 Million.
Köln : 1.08 Million.
Berlin : 3.66 Million.
Hamburg : 1.85 Million.
```

11.3.1 Extending and Modifying

Add `key-value` pairs to a `Dict`...

```
Ppl["Leipzig"] = 0.52
Ppl["Dresden"] = 0.52
Ppl
```

```
Dict{String, Float64} with 6 entries:
"Dresden" => 0.52
"München" => 1.49
"Köln" => 1.08
"Berlin" => 3.66
"Leipzig" => 0.52
"Hamburg" => 1.85
```

Change a `value`:

```
# Update: Leipzig data was from 2010, not 2020
```

```
Ppl["Leipzig"] = 0.597
Ppl
```

```
Dict{String, Float64} with 6 entries:
"Dresden" ⇒ 0.52
"München" ⇒ 1.49
"Köln" ⇒ 1.08
"Berlin" ⇒ 3.66
"Leipzig" ⇒ 0.597
"Hamburg" ⇒ 1.85
```

Delete a pair by its `key`:

```
delete!(Ppl, "Dresden")
```

```
Dict{String, Float64} with 5 entries:
"München" ⇒ 1.49
"Köln" ⇒ 1.08
"Berlin" ⇒ 3.66
"Leipzig" ⇒ 0.597
"Hamburg" ⇒ 1.85
```

Many functions work with `Dicts` like other containers.

```
maximum(values(Ppl))
```

```
3.66
```

11.3.2 Creating an Empty Dictionary

Without explicit types:

```
d1 = Dict()
```

```
Dict{Any, Any}()
```

With explicit types:

```
d2 = Dict{String, Int}()
```

```
Dict{String, Int64}()
```

11.3.3 Conversion to Vectors: `collect()`

- `keys(dict)` and `values(dict)` return special container types.
- `collect()` converts them to `Vectors`.
- `collect(dict)` returns a `Vector{Pair{S,T}}`.

```
collect(Ppl)
```

```
5-element Vector{Pair{String, Float64}}:
"München" ⇒ 1.49
"Köln" ⇒ 1.08
"Berlin" ⇒ 3.66
"Leipzig" ⇒ 0.597
"Hamburg" ⇒ 1.85
```

```
collect(keys(Ppl)), collect(values(Ppl))
```

```
(["München", "Köln", "Berlin", "Leipzig", "Hamburg"], [1.49, 1.08, 3.66, 0.597, 1.85])
```

11.3.4 Ordered Iteration over a Dictionary

We sort the keys. As strings, they are sorted alphabetically. With the `rev` parameter, sorting is done in reverse order.

```
for k in sort(collect(keys(Ppl)), rev = true)
    n = Ppl[k]
    println("$k has $n million inhabitants  ")
end
```

```
München has 1.49 million inhabitants
Leipzig has 0.597 million inhabitants
Köln has 1.08 million inhabitants
Hamburg has 1.85 million inhabitants
Berlin has 3.66 million inhabitants
```

Let's sort `collect(dict)`, a vector of pairs. Use `by` to specify the sort key: the second element of each pair.

```
for (k,v) in sort(collect(Ppl), by = pair -> last(pair), rev=false)
    println("$k has $v million inhabitants")
end
```

```
Leipzig has 0.597 million inhabitants
Köln has 1.08 million inhabitants
München has 1.49 million inhabitants
Hamburg has 1.85 million inhabitants
Berlin has 3.66 million inhabitants
```

11.3.5 An Application of Dictionaries: Counting Frequencies

Let's do "experimental stochastics" with 2 dice:
Let `l` be a vector containing 100,000 sums of two dice rolls (numbers from 2 to 12).
How frequently does each number from 2 to 12 occur?
Roll the dice:

```
l = rand(1:6, 100_000) .+ rand(1:6, 100_000)
```

```
100000-element Vector{Int64}:
2
5
11
3
3
9
9
7
11
6
⋮
8
5
6
11
5
6
9
5
3
```

Count event frequencies using a dictionary. Use the event as the `key` and its frequency as the `value`.

```
# In this case, a simple vector would also work.
# A better use case for dictionaries is word frequency in texts,
# where keys are strings instead of integers.

d = Dict{Int,Int}()     # dictionary for counting

for i in l                    # for each i, increment d[i]
    d[i] = get(d, i, 0) + 1
```

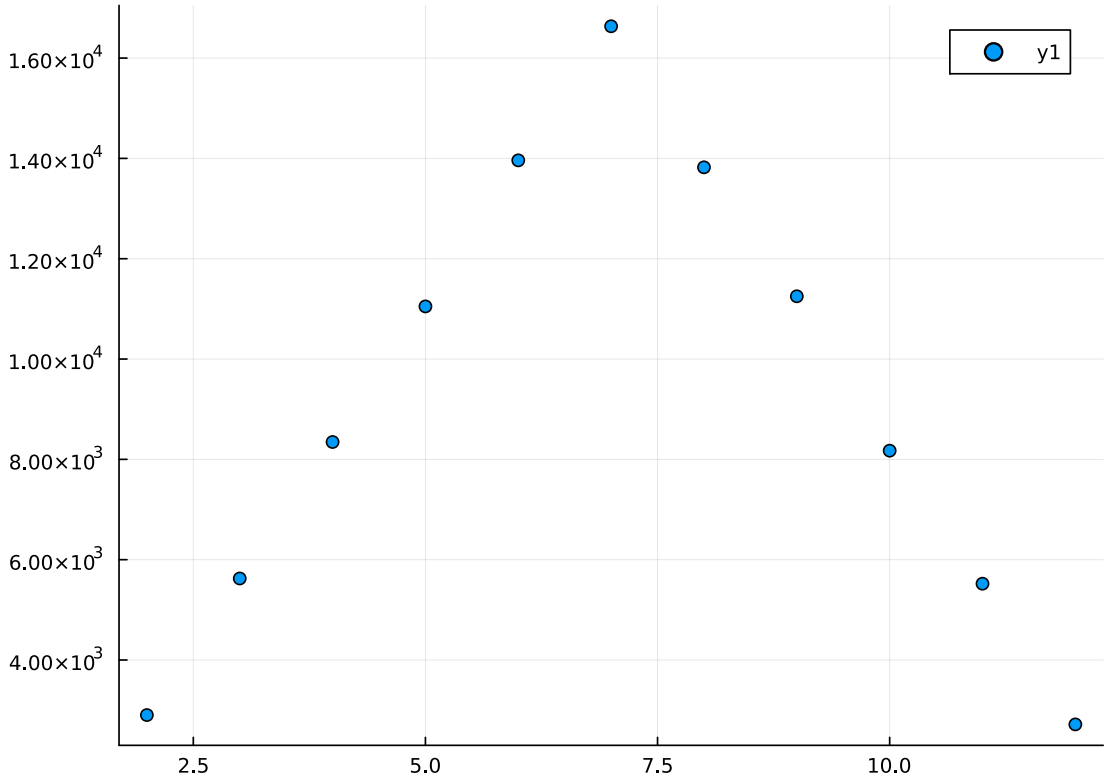
```
end
d
```

```
Dict{Int64, Int64} with 11 entries:
5 ⇒ 11049
12 ⇒ 2716
8 ⇒ 13822
6 ⇒ 13961
11 ⇒ 5521
9 ⇒ 11251
3 ⇒ 5625
7 ⇒ 16634
4 ⇒ 8346
2 ⇒ 2902
10 ⇒ 8173
```

Result:

```
using Plots

plot(collect(keys(d)), collect(values(d)), seriestype=:scatter)
```

```
Precompiling packages...
   1310.5 ms  ✓ QuartoNotebookWorkerPlotsExt (serial)
  1 dependency successfully precompiled in 1 seconds
```



Explanatory image:
https://math.stackexchange.com/questions/1204396/why-is-the-sum-of-the-rolls-of-two-dices-a-binomial-distribution-what-is-define

12. Vectors, Matrices, Arrays

12.1 General

We now turn to containers important for numerical computing:

- Vectors `Vector{T}`
- Matrices `Matrix{T}` with two indices
- N-dimensional arrays with N indices `Array{T,N}`

In fact, `Vector{T}` is an alias for `Array{T,1}` and `Matrix{T}` is an alias for `Array{T,2}`.

```
Vector{Float64} === Array{Float64,1} && Matrix{Float64} ===  Array{Float64,2}
```

```
true
```

When created from an explicit element list, Julia determines the greatest common type for the type parameter `T`.

```
v = [33, "33", 1.2]
```

```
3-element Vector{Any}:
33
"33"
1.2
```

If `T` is a numeric type, the elements are converted to this type.

```
v = [3//7, 4, 2im]
```

```
3-element Vector{Complex{Rational{Int64}}}:
3//7 + 0//1*im
4//1 + 0//1*im
0//1 + 2//1*im
```

The following functions provide information about an array:

- `length(A)` — number of elements
- `eltype(A)` — type of elements
- `ndims(A)` — number of dimensions (indices)
- `size(A)` — tuple with the dimensions of the array

```
v1 = [12, 13, 15]

m1 = [ 1    2.5
       6    -3 ]

for f ∈ (length, eltype, ndims, size)
    println("$(f)(v) = $(f(v)),          $(f)(m1) = $(f(m1))")
end
```

```
length(v) = 3, length(m1) = 4
eltype(v) = Complex{Rational{Int64}}, eltype(m1) = Float64
ndims(v) = 1, ndims(m1) = 2
size(v) = (3,), size(m1) = (2, 2)
```

- The strength of 'classical' arrays for scientific computing is that they are simply contiguous memory segments where same-length components (e.g., 64 bits) are stored sequentially. This minimizes memory requirements and maximizes access speed for both reading and modifying components. The location of `v[i]` can be calculated directly from `i`.

- Julia's `Array{T,N}` (including vectors and matrices) is implemented this way for standard numeric types `T`. Elements are stored *unboxed*. In contrast, `Vector{Any}` is implemented as a list of object addresses *(boxed)*, not as a list of the objects themselves.
- Julia's `Array{T,N}` stores its elements directly *(unboxed)*, when `isbitstype(T) == true`.

```
isbitstype(Float64),
isbitstype(Complex{Rational{Int64}}),
isbitstype(String)
```

```
(true, true, false)
```

12.2 Vectors

12.2.1 Stack-like Functions

- `push!(vector, items...)` — appends elements to the end
- `pushfirst!(vector, items...)` — prepends elements to the beginning
- `pop!(vector)` — removes and returns the last element
- `popfirst!(vector)` — removes and returns the first element

```
v = Float64[]           # empty Vector{Float64}

push!(v, 3, 7)
pushfirst!(v, 1)

a = pop!(v)
println("a= $a")

push!(v, 17)
```

```
a= 7.0
3-element Vector{Float64}:
1.0
3.0
17.0
```

A `push!()` operation can be expensive, as it may require allocating new memory and copying the existing vector. Julia optimizes memory by preallocating space, so subsequent `push!` operations are very fast, almost achieving O(1) speed.

Avoid `push!()` or `resize()` in time-critical code with very large arrays.

12.2.2 Further Constructors

You can create uninitialized vectors of a given length and type. This is the fastest method; elements contain random bit patterns.

```
# Uninitialized vector of length 1000

v = Vector{Float64}(undef, 1000)
v[345]
```

```
1.894679065e-315
```

- `zeros(n)` creates a `Vector{Float64}` of length `n`, initialized with zeros.

```
v = zeros(7)
```

```
7-element Vector{Float64}:
0.0
0.0
0.0
0.0
0.0
```

```
0.0
0.0
```

- `zeros(T,n)` creates a zero vector of type `T`:

```
v=zeros(Int, 4)
```

```
4-element Vector{Int64}:
0
0
0
0
```

- `fill(x, n)` creates a `Vector{typeof(x)}` of length `n`, filled with `x`:

```
v = fill(sqrt(2), 5)
```

```
5-element Vector{Float64}:
1.4142135623730951
1.4142135623730951
1.4142135623730951
1.4142135623730951
1.4142135623730951
```

- `similar(v)` creates an uninitialized vector of the same type and size as `v`:

```
w = similar(v)
```

```
5-element Vector{Float64}:
0.0
0.0
0.0
0.0
0.0
```

12.2.3 Construction by Implicit Loop *(list comprehension)*

Implicit `for` loops provide another method to create vectors.

```
v4 = [i for i in 1.0:8]
```

```
8-element Vector{Float64}:
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
```

```
v5 = [log(i^2) for i in 1:4 ]
```

```
4-element Vector{Float64}:
0.0
1.3862943611198906
2.1972245773362196
2.772588722239781
```

You can even include an `if` clause.

```
v6 = [i^2 for i in 1:8 if i%3 != 2]
```

```
5-element Vector{Int64}:
1
9
```

```
16
36
49
```

12.2.4 Bit Vectors

Besides `Vector{Bool}`, Julia provides the `BitVector` data type (and more generally `BitArray`) for storing boolean arrays.

While a `Bool` uses one byte, `BitVector` stores values bit by bit.

The constructor converts a `Vector{Bool}` to a `BitVector`:

```
vb = BitVector([true, false, true, true])
```

```
4-element BitVector:
1
0
1
1
```

Use `collect()` for the reverse conversion:

```
collect(vb)
```

```
4-element Vector{Bool}:
1
0
1
1
```

Bit vectors are produced, for example, by element-wise comparisons (see Section 12.7):

```
v4 .> 3.5
```

```
8-element BitVector:
0
0
0
1
1
1
1
1
```

12.2.5 Indexing

Indices are ordinal numbers, so **indexing starts at 1.**

Valid indices include:

- an integer
- an integer-valued range (same length or shorter)
- an integer vector (same length or shorter)
- a boolean vector or BitVector (same length)

Using indices, we can read and write array elements/parts.

```
v = [ 3i + 5.2 for i in 1:8]
```

```
8-element Vector{Float64}:
8.2
11.2
14.2
17.2
20.2
23.2
26.2
29.2
```

```
v[5]
```

```
20.2
```

In assignments, the right side is converted to the vector element type with `convert(T,x)` if necessary.

```
v[6] = 9999
v
```

```
8-element Vector{Float64}:
8.2
11.2
14.2
17.2
20.2
9999.0
26.2
29.2
```

Exceeding index limits throws a `BoundsError`.

```
v[77]
```

```
BoundsError: attempt to access 8-element Vector{Float64} at index [77]
Stacktrace:
 [1] throw_boundserror(A::Vector{Float64}, I::Tuple{Int64})
   @ Base ./essentials.jl:15
 [2] getindex(A::Vector{Float64}, i::Int64)
   @ Base ./essentials.jl:919
 [3] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/7_ArraysP2.qmd:227
Error showing value of type BoundsError
StackOverflowError:
Stacktrace:
 [1] show(io::IOContext{IOBuffer}, x::BoundsError) (repeats 79983 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/7_ArraysP2.qmd:24
```

A `range` object can be used to address a subvector.

```
vp = v[3:5]
vp
```

```
3-element Vector{Float64}:
14.2
17.2
20.2
```

```
vp = v[1:2:7]   # range with step size
vp
```

```
4-element Vector{Float64}:
8.2
14.2
20.2
26.2
```

- When used as an index, the special value `end` can be used in a range.
- When used as an index, the "empty" range `:` can be used as an abbreviation for `1:end`. This is useful for matrices: `A[:, 3]` addresses the entire 3rd column of `A`.

```
v[6:end] = [7, 7, 7]
v
```

```
8-element Vector{Float64}:
8.2
11.2
14.2
17.2
```

```
20.2
7.0
7.0
7.0
```

12.2.5.1 Indirect Indexing

Indirect indexing with a *vector of integers* follows the formula
$v[[i_1, i_2, i_3, ...]] = [\, v[i_1], v[i_2], v[i_3], ...]$

```
v[ [1, 3, 4] ]
```

```
3-element Vector{Float64}:
8.2
14.2
17.2
```

which is the same as

```
[ v[1], v[3], v[4] ]
```

```
3-element Vector{Float64}:
8.2
14.2
17.2
```

12.2.5.2 Indexing with a Vector of Truth Values

You can also use a `Vector{Bool}` or `BitVector` (see Section 12.2.4) **of the same length** as an index.

```
v[ [true, true, false, false, true, false, true, true] ]
```

```
5-element Vector{Float64}:
8.2
11.2
20.2
7.0
7.0
```

This is useful for:

- broadcast tests (see Section 12.7) which produce a `BitVector`, and
- combining `BitVector`s with bitwise operators `&` and `|`.

```
v[  (v .> 13) .& (v.<20) ]
```

```
2-element Vector{Float64}:
14.2
17.2
```

12.3 Matrices and Arrays

Most methods for vectors also apply to higher-dimensional arrays.
One can create them uninitialized:

```
A = Array{Float64,3}(undef, 6,9,3)
```

```
6×9×3 Array{Float64, 3}:
[:, :, 1] =
NaN 1.5e-323 3.0e-323 4.4e-323 … 9.0e-323 1.04e-322 1.2e-322
5.0e-324 1.5e-323 3.0e-323 4.4e-323 9.0e-323 1.04e-322 1.2e-322
5.0e-324 2.0e-323 3.5e-323 5.0e-323 9.4e-323 1.1e-322 1.24e-322
5.0e-324 2.0e-323 3.5e-323 5.0e-323 9.4e-323 1.1e-322 1.24e-322
1.0e-323 2.5e-323 4.0e-323 5.4e-323 1.0e-322 1.14e-322 1.3e-322
1.0e-323 2.5e-323 4.0e-323 5.4e-323 … 1.0e-322 1.14e-322 1.3e-322
```

```
[:, :, 2] =
1.33e-322 1.5e-322 1.63e-322 … 2.2e-322 2.37e-322 2.5e-322
1.33e-322 1.5e-322 1.63e-322 2.2e-322 2.37e-322 2.5e-322
1.4e-322 1.53e-322 1.0e-323 2.27e-322 5.0e-324 2.57e-322
1.4e-322 1.53e-322 1.0e-323 2.27e-322 5.0e-324 2.57e-322
1.43e-322 1.6e-322 1.73e-322 2.27e-322 2.47e-322 2.6e-322
1.43e-322 1.6e-322 1.73e-322 … 2.27e-322 2.47e-322 2.6e-322

[:, :, 3] =
2.67e-322 2.8e-322 2.96e-322 … 3.56e-322 3.56e-322 3.1e-322
2.67e-322 2.8e-322 2.96e-322 3.56e-322 3.56e-322 3.1e-322
2.7e-322 2.87e-322 3.0e-322 3.56e-322 3.75e-322 3.0e-322
2.7e-322 2.87e-322 3.0e-322 3.56e-322 3.56e-322 3.0e-322
2.77e-322 2.9e-322 3.06e-322 3.56e-322 3.75e-322 0.0
2.77e-322 2.9e-322 3.06e-322 … 3.56e-322 3.56e-322 0.0
```

In most functions, the dimensions can also be passed as a tuple; the above can be written as:

```
A = Array{Float64, 3}(undef, (6,9,3))
```

Functions like `zeros()` etc. also work.

```
m2 = zeros(3, 4, 2)  # or zeros((3,4,2))
```

```
3×4×2 Array{Float64, 3}:
[:, :, 1] =
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0

[:, :, 2] =
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
```

```
M = fill(5 , (3, 3))   # or fill(5, 3, 3)
```

```
3×3 Matrix{Int64}:
5 5 5
5 5 5
5 5 5
```

The function `similar()`, which creates an uninitialized array of the same size, can also take a type as a further argument.

```
M2 = similar(M, Float64)
```

```
3×3 Matrix{Float64}:
3.177e-321 6.91415e-310 6.91414e-310
NaN 6.91415e-310 3.177e-321
6.9141e-310 5.0e-324 NaN
```

12.3.1 Construction by Explicit Element List

While vectors are written in square brackets separated by commas, the notation for higher-dimensional objects is somewhat different.

- A matrix:

```
M2 = [2  3  -1
      4  5  -2]
```

```
2×3 Matrix{Int64}:
2 3 -1
4 5 -2
```

- the same matrix:

```
M2 = [2 3 -1; 4 5 -2]
```

```
2×3 Matrix{Int64}:
2 3 -1
4 5 -2
```

- An array with 3 indices:

```
M3 = [2   3  -1
      4   5   6 ;;;
      7   8   9
     11  12  13]
```

```
2×3×2 Array{Int64, 3}:
[:, :, 1] =
2 3 -1
4 5 6

[:, :, 2] =
7 8 9
11 12 13
```

- and once more the matrix `M2`:

```
M2 = [2;4;; 3;5;; -1;-2]
```

```
2×3 Matrix{Int64}:
2 3 -1
4 5 -2
```

Here, the following rules apply:

- The separator is the semicolon.
- A semicolon `;` increases the 1st index.
- Two semicolons `;;` increase the 2nd index.
- Three semicolons `;;;` increase the 3rd index etc.

In the previous examples, the following syntactic enhancement (*syntactic sugar*) was applied:

- Spaces separate like two semicolons – thus increasing the 2nd index: a_{12} a_{13} a_{14} ...
- Newline separates like a semicolon – thus increasing the 1st index.

!Important

- Vector notation with comma as separator only works for vectors – do not mix "semicolon, space, newline".
- Vectors, $1 \times n$-matrices, and $n \times 1$-matrices are three different things.

```
v1 = [2,3,4]
```

```
3-element Vector{Int64}:
2
3
4
```

```
v2 = [2;3;4]
```

```
3-element Vector{Int64}:
2
3
4
```

```
v3 = [2 3 4]
```

```
1×3 Matrix{Int64}:
2 3 4
```

```
v3 = [2;3;4;;]
```

```
3×1 Matrix{Int64}:
2
3
4
```

One can of course also construct a "vector of vectors" in the C/C++ style.

```
v = [[2,3,4], [5,6,7,8]]
```

```
2-element Vector{Vector{Int64}}:
[2, 3, 4]
[5, 6, 7, 8]
```

```
v[2][3]
```

```
7
```

You should only do this in special cases. The array notation in Julia is usually more convenient and faster.

12.3.2 Indices, Subarrays, Slices

```
# 6x6 matrix with random numbers uniformly distributed from [0,1) ∈ Float64
A = rand(6,6)
```

```
6×6 Matrix{Float64}:
0.463067 0.672554 0.56387 0.470856 0.958354 0.155192
0.998119 0.771962 0.79258 0.183362 0.0200682 0.773312
0.956628 0.258465 0.774233 0.768411 0.602852 0.0863743
0.211162 0.650677 0.286221 0.401343 0.482077 0.179016
0.539544 0.39127 0.356355 0.36664 0.315264 0.646141
0.189131 0.864078 0.41988 0.806717 0.0303571 0.794074
```

The usual index notation:

```
A[2, 3] = 77.77777
A
```

```
6×6 Matrix{Float64}:
0.463067 0.672554 0.56387 0.470856 0.958354 0.155192
0.998119 0.771962 77.7778 0.183362 0.0200682 0.773312
0.956628 0.258465 0.774233 0.768411 0.602852 0.0863743
0.211162 0.650677 0.286221 0.401343 0.482077 0.179016
0.539544 0.39127 0.356355 0.36664 0.315264 0.646141
0.189131 0.864078 0.41988 0.806717 0.0303571 0.794074
```

You can use ranges to address subarrays:

```
B = A[1:2, 1:3]
```

```
2×3 Matrix{Float64}:
0.463067 0.672554 0.56387
0.998119 0.771962 77.7778
```

Addressing parts with lower dimension is also called *slicing*.

```
# the 3rd column as a vector (slicing)

C = A[:, 3]
```

```
6-element Vector{Float64}:
0.5638695116046222
77.77777
0.774232845981998
0.2862211990082818
0.35635508273885474
0.41988037076788776
```

```
# the 3rd row as a vector (slicing)

E = A[3, :]
```

```
6-element Vector{Float64}:
0.9566275711275898
0.2584648669445483
0.774232845981998
0.7684109261091199
0.6028521581317805
0.0863743232119617
```

Slicing can also used on the right hand side for assignments:

```
# You can also assign to slices and subarrays

A[2, :] = [1,2,3,4,5,6]
A
```

```
6×6 Matrix{Float64}:
0.463067 0.672554 0.56387 0.470856 0.958354 0.155192
1.0 2.0 3.0 4.0 5.0 6.0
0.956628 0.258465 0.774233 0.768411 0.602852 0.0863743
0.211162 0.650677 0.286221 0.401343 0.482077 0.179016
0.539544 0.39127 0.356355 0.36664 0.315264 0.646141
0.189131 0.864078 0.41988 0.806717 0.0303571 0.794074
```

12.4 Behavior in Assignments, `copy()` and `deepcopy()`, Views

12.4.1 Assignments and Copies

- Variables are references to objects.
- *An assignment to a variable does not create a new object.*

```
A = [1, 2, 3]
B = A
```

```
3-element Vector{Int64}:
1
2
3
```

A and B are now names of the same object.

```
A[1] = 77
@show B;
```

```
B = [77, 2, 3]
```

```
B[3] = 300
@show A;
```

```
A = [77, 2, 300]
```

This behavior saves a lot of time and memory, but is not always desired. The function copy() creates a true copy of the object.

```
A = [1, 2, 3]
B = copy(A)
A[1] = 100
@show A B;
```

```
A = [100, 2, 3]
B = [1, 2, 3]
```

The function deepcopy(A) copies recursively. It also creates copies of the elements that A contains.

As long as an array only contains primitive objects (numbers), copy() and deepcopy() are equivalent. The following example shows the difference between copy() and deepcopy().

```
mutable struct Person
    name :: String
    age  :: Int
end

A = [Person("Meier", 20), Person("Müller", 21), Person("Schmidt", 23)]
B = A
C = copy(A)
D = deepcopy(A)
```

```
3-element Vector{Person}:
Person(Meier, 20)
Person(Müller, 21)
Person(Schmidt, 23)
```

```
A[1] = Person("Mustermann", 83)
A[3].age = 199

@show B C D;
```

```
B = Main.Notebook.Person[Person(Mustermann, 83), Person(Müller, 21), Person(Schmidt,
199)]
C = Main.Notebook.Person[Person(Meier, 20), Person(Müller, 21), Person(Schmidt, 199)]
D = Main.Notebook.Person[Person(Meier, 20), Person(Müller, 21), Person(Schmidt, 23)]
```

12.4.2 Views

When one assigns a piece of an array to a variable using *indices/ranges/slices*, Julia **constructs a new object**.

```
A = [1  2  3
     3  4  5]

v = A[:, 2]
@show  v

A[1, 2] = 77
@show  A  v;
```

```
v = [2, 4]
A = [1 77 3; 3 4 5]
v = [2, 4]
```

Sometimes, however, one wants reference semantics in the sense of: "Vector `v` should be the 2nd column vector of `A` and should also remain so (i.e., change if `A` changes)."

This is called a *view* in Julia: We want the variable `v` to represent only an 'alternative view' of the matrix `A`.

It can be achieved with the `@view` macro:

```
A = [1 2 3
     3 4 5]

v = @view A[:,2]
@show  v

A[1, 2] = 77
@show  v;
```

```
v = [2, 4]
v = [77, 4]
```

Julia uses this technique for efficiency reasons also in some functions of linear algebra. An example is the operator `'`, which delivers the adjoint matrix `A'` to a matrix `A`.

- The adjoint matrix `A'` is the transposed and element-wise complex-conjugated matrix to `A`.
- The parser converts this to the function call `adjoint(A)`.
- For real matrices, the adjoint is equal to the transposed matrix.
- Julia implements `adjoint()` as a *lazy function*, i.e., for efficiency reasons no new object is constructed. The method provides an alternative 'view' of the matrix (with swapped indices) and an alternative 'view' of the entries (with sign change in the imaginary part).
- The adjoint of a vector produces a $1 \times n$ matrix (row vector).

```
A = [ 1.  2.
      3.  4.]
B = A'
```

```
2×2 adjoint(::Matrix{Float64}) with eltype Float64:
1.0 3.0
2.0 4.0
```

The matrix `B` is just a modified 'view' of `A`:

```
A[1, 2] =10
B
```

```
2×2 adjoint(::Matrix{Float64}) with eltype Float64:
1.0 3.0
10.0 4.0
```

From vectors, `adjoint()` makes a $1 \times n$-matrix (a row vector).

```
v = [1, 2, 3]
v'
```

```
1×3 adjoint(::Vector{Int64}) with eltype Int64:
1 2 3
```

Another such function, which provides an alternative 'view', a different indexing of the same data is `reshape()`.

Here, a vector with 12 entries is transformed into a 3x4 matrix:

```
A = [1,2,3,4,5,6,7,8,9,10,11,12]

B = reshape(A, 3, 4)
```

```
3×4 Matrix{Int64}:
1 4 7 10
2 5 8 11
3 6 9 12
```

12.5 Storage of an Array

- Memory is addressed linearly. A matrix can be stored in memory row-wise *(row major)* or column-wise *(column major)*.
- C/C++/Python(NumPy) use row-major storage: The 4 elements of a 2x2 matrix are stored in the order $a_{11}, a_{12}, a_{21}, a_{22}$.
- Julia, Fortran, Matlab store column-wise: $a_{11}, a_{21}, a_{12}, a_{22}$.

This information is important to iterate efficiently over matrices:

```
function column_major_add(A, B)
    (n,m) = size(A)
    for j = 1:m
        for i = 1:n      # inner loop traverses a column
            A[i,j] += B[i,j]
        end
    end
end

function row_major_add(A, B)
    (n,m) = size(A)
    for i = 1:n
        for j = 1:m      # inner loop traverses a row
            A[i,j] += B[i,j]
        end
    end
end
```

```
row_major_add (generic function with 1 method)
```

```
A = rand(10000, 10000);
B = rand(10000, 10000);
```

```
using BenchmarkTools

@benchmark row_major_add($A, $B)
```

```
BenchmarkTools.Trial: 6 samples with 1 evaluation per sample.
 Range (min … max):  905.524 ms … 985.877 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     979.043 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   956.631 ms ±  39.531 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

 █
 █▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁█▁█▁██ ▁
 906 ms           Histogram: frequency by time          986 ms <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
@benchmark column_major_add($A, $B)
```

```
BenchmarkTools.Trial: 74 samples with 1 evaluation per sample.
 Range (min … max):  67.467 ms … 76.844 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     67.967 ms              ┊ GC (median):    0.00%
 Time  (mean ± σ):   68.172 ms ±  1.087 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

  67.5 ms         Histogram: frequency by time        69.5 ms <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

12.5.1 Locality of Memory Access and *Caching*

We have observed that the order of inner and outer loops significantly affects computational efficiency:

It is more efficient when the innermost loop iterates over the left index, i.e., a column rather than a row. This is due to the architecture of modern processors.

- Memory access involves multiple cache levels.
- A *cache miss*, which necessitates reloading from slower caches, slows down processing.
- To minimize the frequency of *cache misses*, processors reload larger memory blocks.
- Consequently, it is crucial to organize memory access as locally as possible.

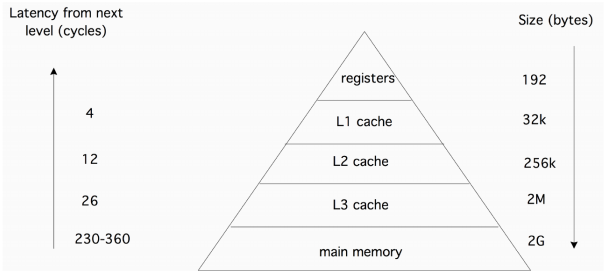


Figure 1.5: Memory hierarchy of an Intel Sandy Bridge, characterized by speed and size.

Figure 12.1: Memory hierarchy of Intel processors, from: Victor Eijkhout, *Introduction to High-Performance Scientific Computing*, https://theartofhpc.com/

12.6 Mathematical Operations with Arrays

Arrays of the same dimension (e.g., all 7×3-matrices) form a linear space.

- They can be multiplied by scalars and
- they can be added and subtracted.

```
0.5 * [2, 3, 4, 5]
```

```
4-element Vector{Float64}:
1.0
1.5
2.0
2.5
```

```
0.5 * [ 1  3
        2  7] - [ 2  3; 1 2]
```

```
2×2 Matrix{Float64}:
-1.5 -1.5
0.0 1.5
```

12.6.1 Matrix Product

The matrix product is defined for

1st factor	2nd factor	Product
$(n \times m)$-matrix	$(m \times k)$-matrix	$(n \times k)$-matrix
$(n \times m)$-matrix	m-vector	n-vector
$(1 \times m)$-row vector	$(m \times n)$-matrix	n-vector
$(1 \times m)$-row vector	m-vector	scalar product
m-vector	$(1 \times n)$-row vector	$(m \times n)$-matrix

Examples:

```
A = [1 2 3
     4 5 6]
v = [2, 3]
w = [1, 3, 4];
```

- (2,3)-matrix ∗ (3,2)-matrix

```
A * A'
```

```
2×2 Matrix{Int64}:
14 32
32 77
```

- (3,2)-matrix ∗ (2,3)-matrix

```
A' * A
```

```
3×3 Matrix{Int64}:
17 22 27
22 29 36
27 36 45
```

- (2,3)-matrix ∗ 3-vector

```
A * w
```

```
2-element Vector{Int64}:
19
43
```

- (1,2)-vector ∗ (2,3)-matrix

```
v' * A
```

```
1×3 adjoint(::Vector{Int64}) with eltype Int64:
14 19 24
```

- (3,2)-matrix ∗ 2-vector

```
A' * v
```

```
3-element Vector{Int64}:
14
19
24
```

- (1,2)-vector ∗ 2-vector (scalar product)

```
v' * v
```

```
13
```

2-vector ∗ (1,3)-vector (outer product)

```
v * w'
```

```
2×3 Matrix{Int64}:
2 6 8
3 9 12
```

12.7 Broadcasting

- With *broadcasting*, operations or functions are applied element-wise to arrays.
- Broadcasting is indicated by a dot preceding an operator or following a function name.
- The parser translates `f.(x,y)` into `broadcast(f, x, y)`, and similarly, `x .⊙ y` into `broadcast(⊙, x, y)`.
- Operands lacking one or more dimensions are virtually replicated in those dimensions.
- Broadcasting assignments such as `.=`, `.+=`, etc., alter the semantics by modifying values directly within the left-side object (which must have the appropriate dimensions) without creating a new object.
- With *broadcasting*, operations or functions are applied element-wise to arrays.

Some examples:

- Element-wise application of a function

```
sin.([1, 2, 3])
```

```
3-element Vector{Float64}:
0.8414709848078965
0.9092974268256817
0.1411200080598672
```

- The following does not give the algebraic square root of a matrix, but the element-wise square root of each entry.

```
A = [8 2
     3 4]

sqrt.(A)
```

```
2×2 Matrix{Float64}:
2.82843 1.41421
1.73205 2.0
```

- The following does not give A^2, but the entries are squared.

```
A.^2
```

```
2×2 Matrix{Int64}:
64 4
9 16
```

- For comparison, here the results of the algebraic operations:

```
@show A^2 A^(1/2);
```

```
A ^ 2 = [70 24; 36 22]
A ^ (1 / 2) = [2.780234855920959 0.42449510866609885; 0.6367426629991483
1.9312446385887614]
```

- Broadcasting also works with functions of several variables.

```
hyp(a,b) = sqrt(a^2+b^2)
```

```
B = [3 4
     5 7]

hyp.(A, B)
```

```
2×2 Matrix{Float64}:
8.544 4.47214
5.83095 8.06226
```

When operands possess differing dimensions, the operand with fewer dimensions is effectively expanded through replication along those dimensions.

Adding a scalar to a matrix:

```
A = [ 1 2 3
      4 5 6]
```

```
2×3 Matrix{Int64}:
1 2 3
4 5 6
```

```
A .+ 300
```

```
2×3 Matrix{Int64}:
301 302 303
304 305 306
```

The scalar was replicated to match the matrix dimensions. Let `broadcast()` illustrate the form of the second operand after broadcasting:

```
broadcast( (x,y) → y, A, 300)
```

(This replication occurs solely in a virtual sense; the object is not actually instantiated in memory.)

```
A .+ [10, 20]
```

```
2×3 Matrix{Int64}:
11 12 13
24 25 26
```

The vector is expanded by repeating columns:

```
broadcast((x,y)→y, A, [10,20])
```

```
2×3 Matrix{Int64}:
10 10 10
20 20 20
```

Matrix and row vector: The row vector is repeated row-wise:

```
A .* [1,2,3]'      # adjoint vector
```

```
2×3 Matrix{Int64}:
1 4 9
4 10 18
```

The 2nd operand is grown by `broadcast()` through replication of rows.

```
broadcast((x,y)→y, A, [1,2,3]')
```

```
2×3 Matrix{Int64}:
1 2 3
1 2 3
```

12.7.0.1 *Broadcasting* in Assignments

Assignments such as `=`, `+=`, `/=`,... in Julia involve constructing an object on the right-hand side and assigning it to a variable on the left-hand side.

When working with arrays, efficiency often requires reusing existing array objects. The values computed on the right-hand side are then stored directly into the pre-existing object on the left-hand side.

This is accomplished using broadcast variants of assignment operators: `.=`, `.+=`,....

```
A .= 3
```

```
2×3 Matrix{Int64}:
3 3 3
3 3 3
```

```
A .+= [1, 4]
```

```
2×3 Matrix{Int64}:
4 4 4
7 7 7
```

12.8 Further Array Functions - a Selection

Julia provides a large number of functions that work with arrays.

```
A = [22 -17 8 ; 4 6 9]
```

```
2×3 Matrix{Int64}:
22 -17 8
4 6 9
```

- Find the maximum

```
maximum(A)
```

```
22
```

- Find the maximum of each column

```
maximum(A, dims=1)
```

```
1×3 Matrix{Int64}:
22 6 9
```

- Find the maximum of each row

```
maximum(A, dims=2)
```

```
2×1 Matrix{Int64}:
22
9
```

- Find the minimum and its position

```
amin, i = findmin(A)
```

```
(-17, CartesianIndex(1, 2))
```

- What is a `CartesianIndex`?

```
dump(i)
```

```
CartesianIndex{2}
I: Tuple{Int64, Int64}
1: Int64 1
2: Int64 2
```

- Extract the indices of the minimum as a tuple

```
i.I
```

```
(1, 2)
```

- Sum and product of all entries

```
sum(A), prod(A)
```

```
(32, -646272)
```

- Column sum (1st index is reduced)

```
sum(A, dims=1)
```

```
1×3 Matrix{Int64}:
26 -11 17
```

- Row sums (2nd index is reduced)

```
sum(A, dims=2)
```

```
2×1 Matrix{Int64}:
13
19
```

- Sum after applying a function element-wise

```
sum(x->sqrt(abs(x)), A)   #   sum_ij sqrt(|a_ij|)
```

```
19.09143825297046
```

- Reduce (fold) the array with a function

```
reduce(+, A)    #  equivalent to sum(A)
```

```
32
```

- `mapreduce(f, op, array)`: Apply `f` to all entries, then reduce with `op`

```

mapreduce(x -> x^2, +, A )     # Sum of the squares of all entries
```

```
970
```

- Are there elements in A that are > 5?

```
any(x -> x>5, A)
```

```
true
```

- How many elements in A are > 5?

```
count(x-> x>5, A)
```

```
4
```

- are all entries positive?

```
all(x-> x>0, A)
```

```
false
```

13. Linear Algebra in Julia

```
using LinearAlgebra
```

The `LinearAlgebra` package provides, among other things:

- additional subtypes of `AbstractMatrix` which are usable like other matrices, among them:
  - `Tridiagonal`
  - `SymTridiagonal`
  - `Symmetric`
  - `UpperTriangular`
- additional/extended functions: `norm`, `opnorm`, `cond`, `inv`, `det`, `exp`, `tr`, `dot`, `cross`, ...
- a universal solver for systems of linear equations: `\`
  - `x = A \ b` solves $A\mathbf{x} = \mathbf{b}$ by appropriate matrix factorization and forward/backward substitution
- Matrix factorizations
  - `LU`
  - `QR`
  - `Cholesky`
  - `SVD`
  - ...
- Computation of eigenvalues/eigenvectors
  - `eigen`, `eigvals`, `eigvecs`
- Access to BLAS/LAPACK functions

13.1 Special Matrix Types

```
A = SymTridiagonal(fill(1.0, 4), fill(-0.3, 3))
```

```
4×4 SymTridiagonal{Float64, Vector{Float64}}:
1.0 -0.3  ⋅   ⋅
-0.3 1.0 -0.3  ⋅
 ⋅  -0.3 1.0 -0.3
 ⋅   ⋅  -0.3 1.0
```

```
B = UpperTriangular(A)
```

```
4×4 UpperTriangular{Float64, SymTridiagonal{Float64, Vector{Float64}}}:
1.0 -0.3  0.0  0.0
 ⋅   1.0 -0.3  0.0
 ⋅    ⋅   1.0 -0.3
 ⋅    ⋅    ⋅   1.0
```

These types are stored space-efficiently. The usual arithmetic operations are implemented:

```
A + B
```

```
4×4 Matrix{Float64}:
2.0 -0.6  0.0  0.0
-0.3 2.0 -0.6  0.0
0.0 -0.3 2.0 -0.6
0.0  0.0 -0.3 2.0
```

Read-only index access is possible,

```
A[1,4]
```

```
0.0
```

Write operations are not necessarily possible:

```
A[1,3] = 17
```

```
ArgumentError: cannot set off-diagonal entry (1, 3)
Stacktrace:
 [1] setindex!(A::SymTridiagonal{Float64, Vector{Float64}}, x::Int64, i::Int64, j::Int64)
      @ LinearAlgebra ~/.julia/juliaup/julia-1.12.5+0.x64.linux.gnu/share/julia/stdlib/v1.12/
LinearAlgebra/src/tridiag.jl:486
 [2] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/11_LinAlg.qmd:91
Error showing value of type ArgumentError
StackOverflowError:
Stacktrace:
 [1] show(io::IOContext{IOBuffer}, x::ArgumentError) (repeats 79983 times)
   @ Main.Notebook ~/Julia/Book26/JuliaBook/chapters/11_LinAlg.qmd:24
```

Conversion to a 'normal' matrix is possible using `collect()`:

```
A2 = collect(A)
```

```
4×4 Matrix{Float64}:
1.0 -0.3  0.0  0.0
-0.3  1.0 -0.3  0.0
0.0 -0.3  1.0 -0.3
0.0  0.0 -0.3  1.0
```

13.1.1 The Identity Matrix `I`

`I` denotes an identity matrix (square, diagonal elements = 1, all others = 0) of the required size

```
A + 4I
```

```
4×4 SymTridiagonal{Float64, Vector{Float64}}:
5.0 -0.3  ⋅   ⋅
-0.3 5.0 -0.3  ⋅
 ⋅  -0.3 5.0 -0.3
 ⋅   ⋅  -0.3 5.0
```

13.2 Norms

To study questions such as conditioning or convergence of an algorithm, we need a metric. For linear spaces, it is appropriate to define the metric via a norm:

$$d(x, y) := \|x - y\|$$

13.2.1 p-Norms

A simple class of norms in $\mathbb{R}^n$ are the p-norms

$$\|\mathbf{x}\|_p = \left(\sum |\, x_i \,|^p\right)^{\frac{1}{p}},$$

which generalize the Euclidean norm $p = 2$.

The Max-Norm $p = \infty$

Let $x_{\max}$ be the component of $\mathbf{x} \in \mathbb{R}^n$ with the largest absolute value. Then always

$$| x_{\max} | \leq \|\mathbf{x}\|_p \leq n^{\frac{1}{p}} | x_{\max} |$$

(Consider a vector whose components are all equal to $x_{\max}$ respectively a vector whose components are all equal to zero except $x_{\max}$.)

It follows that

$$\lim_{p\to\infty} \|\mathbf{x}\|_p = | x_{\max} | =: \|\mathbf{x}\|_\infty .$$

In Julia, the `LinearAlgebra` package defines a function `norm(v, p)`.

```
v = [3, 4]
w = [-1, 2, 33.2]

@show norm(v)  norm(v, 2)  norm(v, 1) norm(v, 4) norm(w, Inf);
```

```
norm(v) = 5.0
norm(v, 2) = 5.0
norm(v, 1) = 7.0
norm(v, 4) = 4.284572294953817
norm(w, Inf) = 33.2
```

- If the 2nd argument `p` is missing, `p=2` is set.
- The 2nd argument can also be `Inf` (i.e., $+\infty$).
- The 1st argument can be any numerical container. The sum $\sum | x_i |^p$ extends over *all* elements of the container.
- Thus, for a matrix `norm(A)` is equal to the *Frobenius norm* of the matrix `A`.

```
A = [1 2 3
     4 5 6
     7 8 9]
norm(A)      # Frobenius norm
```

```
16.881943016134134
```

Since norms are homogeneous under multiplication with scalars, $\|\lambda\mathbf{x}\| = | \lambda | \cdot \|\mathbf{x}\|$, they are completely determined by the specification of the unit ball. Subadditivity of the norm (triangle inequality) is equivalent to the convexity of the unit ball (code visible by clicking).

```
using Plots

colors=[:purple, :green, :red, :blue,:aqua, :black]
x=LinRange(-1, 1, 1000)
y=LinRange(-1, 1, 1000)

fig1=plot()
for p ∈ (0.8, 1, 1.5, 2, 3.001, 1000)
    contour!(x,y, (x,y) -> p * norm([x, y], p), levels=[p], aspect_ratio=1,
        cbar=false, color=[pop!(colors)], contour_labels=true, ylim=[-1.1, 1.1])
end
fig1
```

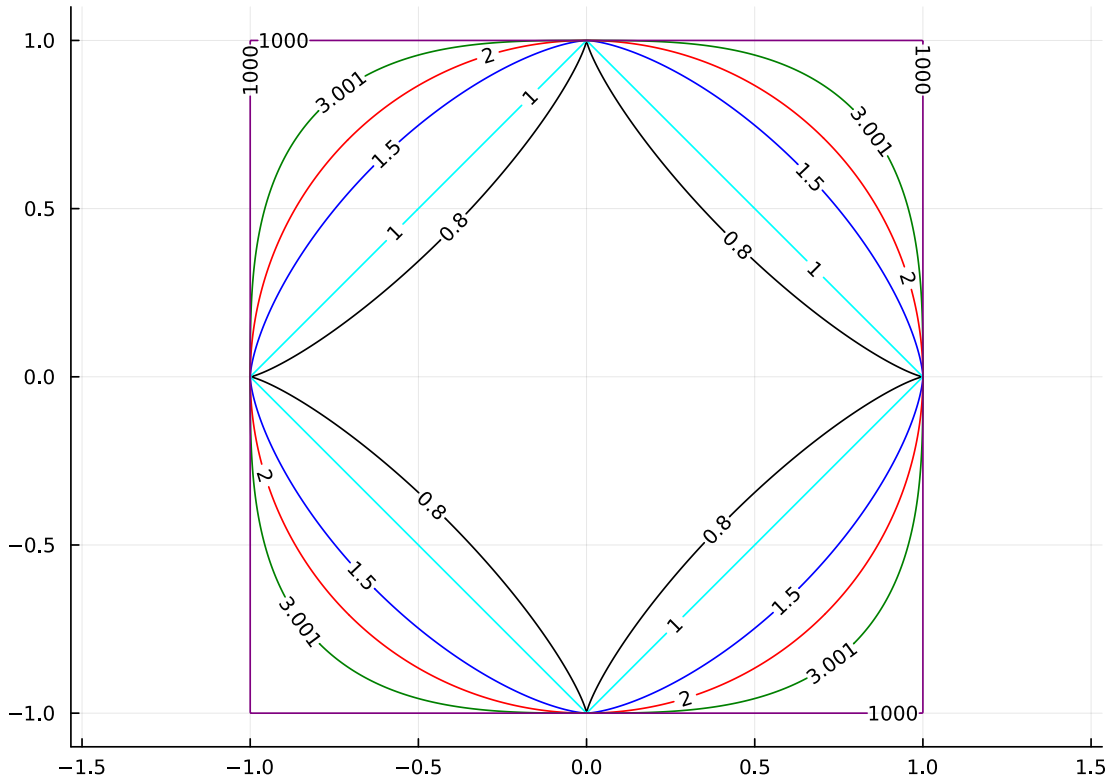


Figure 13.1: Unit balls in $\mathbb{R}^2$ for different p-norms: p=0.8; 1; 1.5; 2; 3.001 and 1000

We see that, $p \geq 1$ must hold so that the unit ball is convex and $\|.\|_p$ is a norm.
However, the Julia function `norm(v, p)` also works for parameters `p<1`.

13.2.2 Induced Norms (Operator Norms)

Matrices A represent linear mappings $\mathbf{v} \mapsto A\mathbf{v}$. The matrix norm induced by a vector norm answers the question:

"By what factor can a vector be maximally stretched by the transformation A?"

Due to the homogeneity of the norm under multiplication with scalars, it is sufficient to consider the image of the unit ball under the transformation A.

Definition

Let V be a vector space with dimension $0 < n < \infty$ and A an $n \times n$-matrix. Then

$$\|A\|_p = \max_{\|\mathbf{v}\|_p=1} \|A\mathbf{v}\|_p$$

Induced norms are difficult to calculate for general p. Exceptions are the cases

- $p = 1$: column sum norm
- $p = 2$: spectral norm and
- $p = \infty$: row sum norm

These 3 cases are implemented in Julia in the function `opnorm(A, p)` from the `LinearAlgebra` package, where again `opnorm(A) = opnorm(A, 2)` holds.

```
A = [ 0    1
      1.2  1.5 ]

@show opnorm(A, 1) opnorm(A, Inf) opnorm(A, 2) opnorm(A);
```

```
opnorm(A, 1) = 2.5
opnorm(A, Inf) = 2.7
opnorm(A, 2) = 2.0879899930905124
opnorm(A) = 2.0879899930905124
```

The following picture shows the effect of A on unit vectors. Vectors of the same color are mapped onto each other. (Code visible by clicking):

```
using CairoMakie

A = [ 0    1
      1.2  1.5 ]

t = LinRange(0, 1, 100)
xs = sin.(2π*t)
ys = cos.(2π*t)
Axys = A * [xs, ys]

u = [sin(n*π/6) for n=0:11]
v = [cos(n*π/6) for n=0:11]
x = y = zeros(12)

Auv = A * [u,v]

fig2 = Figure(size=(800, 400))
lines(fig2[1, 1], xs, ys, color=t, linewidth=5, colormap=:hsv, axis=(; aspect = 1,
limits=(-2,2, -2,2),
      title=L"$\mathbf{v}$", titlesize=30))
arrows2d!(fig2[1,1], x, y, u, v, tipwidth=10, colormap=:hsv, shaftcolor=range(0,11),
shaftwidth=3)

Legend(fig2[1,2], MarkerElement[], String[], L"⇒", width=40, height=30, titlesize=30,
framevisible=false)

lines(fig2[1,3], Axys[1], Axys[2], color=t, linewidth=5, colormap=:hsv, axis=(;
aspect=1, limits=(-2,2, -2,2),
      title=L"$A\mathbf{v}$", titlesize=30))
arrows2d!(fig2[1,3], x, y, Auv[1], Auv[2], tipwidth=10, colormap=:hsv,
shaftcolor=range(0,11),
            shaftwidth=3)

fig2
```

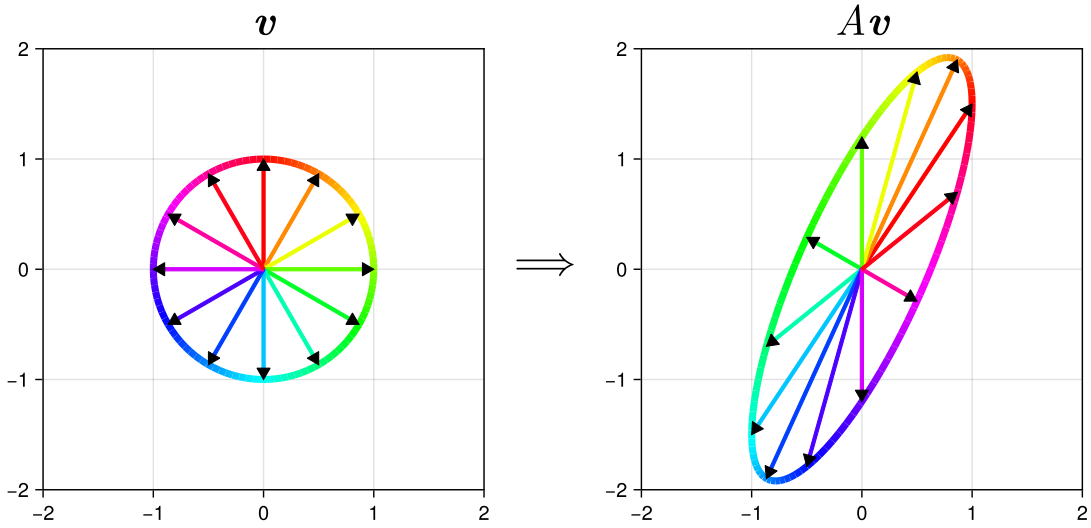


Figure 13.2: Image of the unit ball under $v \mapsto Av$ with $\|A\| \approx 2.088$

13.2.3 Condition Number

For p = 1, p = 2 (default) or p = Inf, `cond(A,p)` returns the condition number in the p-norm

$$\operatorname{cond}_p(A) = \|A\|_p \cdot \left\|A^{-1}\right\|_p$$

```
@show cond(A, 1)  cond(A, 2) cond(A)  cond(A, Inf);
```

```
cond(A, 1) = 5.625
cond(A, 2) = 3.633085176038432
cond(A) = 3.633085176038432
cond(A, Inf) = 5.625000000000001
```

13.3 Matrix Factorizations

The basic tasks of numerical linear algebra:

- Solve a system of linear equations $A\mathbf{x} = \mathbf{b}$.
- If no solution exists, find the best approximation, i.e., the vector $\mathbf{x}$ that minimizes $\|A\mathbf{x} - \mathbf{b}\|$.
- Find eigenvalues and eigenvectors $A\mathbf{x} = \lambda\mathbf{x}$ of A.

These tasks can be solved using matrix factorizations. Some fundamental matrix factorizations:

- **LU decomposition** $A = L \cdot U$
  - factorizes a matrix as a product of a *lower* and an *upper* triangular matrix
  - always works (possibly after row exchanges - pivoting)
- **Cholesky decomposition** $A = L \cdot L^*$
  - the upper triangular matrix is the conjugate of the lower,
  - half the effort compared to LU
  - only works if A is Hermitian and positive definite
- **QR decomposition** $A = Q \cdot R$
  - decomposes A as a product of an orthogonal (or unitary in the complex case) matrix and an upper triangular matrix
  - Q is length-preserving (rotations and/or reflections); the scalings are described by R
  - always works
- **SVD** *(Singular value decomposition)*: $A = U \cdot D \cdot V^*$
  - U and V are orthogonal (or unitary), D is a diagonal matrix with entries on the diagonal $\sigma_i \geq 0$, the so-called *singular values* of A.

– Every linear transformation $\mathbf{v} \mapsto A\mathbf{v}$ can thus be represented as a rotation (and/or reflection) V^*, followed by a pure scaling $v_i \mapsto \sigma_i v_i$ and another rotation U.

13.3.1 LU Factorization

LU factorization is Gaussian elimination. The result of Gaussian elimination is the upper triangular matrix U. The lower triangular matrix L contains ones on the diagonal and the non-diagonal entries l_{ij} are equal to minus the coefficients by which row Z_j is multiplied and added to row Z_i in Gaussian elimination:

$$A = \begin{bmatrix} 1 & 2 & 2 \\ 3 & -4 & 4 \\ -2 & 1 & 5 \end{bmatrix}_{\substack{Z_2 \mapsto Z_2 - 3Z_1 \\ Z_3 \mapsto Z_3 + 2Z_1}} \Longrightarrow \begin{bmatrix} 1 & 2 & 2 \\ & -10 & -2 \\ & 5 & 9 \end{bmatrix}_{Z_3 \mapsto Z_3 + \frac{1}{2} Z_2} \Longrightarrow \begin{bmatrix} 1 & 2 & 2 \\ & -10 & -2 \\ & & 8 \end{bmatrix} = U$$

$$A = \begin{bmatrix} 1 & & \\ +3 & 1 & \\ -2 & -\frac{1}{2} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ & -10 & -2 \\ & & 8 \end{bmatrix}$$

- In practice, it is often necessary to solve $A\mathbf{x} = \mathbf{b}$ for a fixed matrix A and multiple right-hand sides $\mathbf{b}$.
- The factorization of A, which has a cubic complexity $\sim n^3$ relative to the size n of the matrix, needs to be performed only once.
- For each subsequent right-hand side $\mathbf{b}$, the forward and backward substitution steps have quadratic complexity $\sim n^2$.

The `LinearAlgebra` package of Julia contains the function `lu(A, options)` for calculating an LU decomposition:

```
A = [ 1   2  2
      3  -4  4
     -2   1  5]

L, U, _ = lu(A, NoPivot())
display(L)
display(U)
```

```
3×3 Matrix{Float64}:
1.0 0.0 0.0
3.0 1.0 0.0
-2.0 -0.5 1.0
3×3 Matrix{Float64}:
1.0 2.0 2.0
0.0 -10.0 -2.0
0.0 0.0 8.0
```

13.3.1.1 Pivoting

Let's look at one step of Gaussian elimination:

$$\begin{bmatrix} * & \cdots & * & * & \cdots & * \\ & \ddots & \vdots & \vdots & & \vdots \\ & & * & * & \cdots & * \\ & & & a_{ij} & \cdots & a_{in} \\ & & & a_{i+1,j} & \cdots & a_{i+1,n} \\ & & & \vdots & & \vdots \\ & & & a_{mj} & \cdots & a_{mn} \end{bmatrix}$$

The goal is to make the entry $a_{i+1,j}$ disappear by adding an appropriate multiple of row Z_i to row Z_{i+1}. This only works if the *pivot element* a_{ij} is not zero. If $a_{ij} = 0$, we must exchange rows to fix this.

Furthermore, the conditioning of the algorithm is best if we arrange the matrix at each step so that the pivot element is the largest in absolute value in the corresponding column of the remaining submatrix. In (row) pivoting, rows are exchanged to ensure that

$$|a_{ij}| = \max_{k=1,\ldots,m} | a_{kj} |$$

13.3.1.2 LU in Julia

- The factorizations in Julia return a special object that contains the matrix factors and additional information.
- The Julia function `lu(A)` performs an LU factorization with pivoting.

```
F = lu(A)
typeof(F)
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
```

Elements of the object:

```
@show F.L F.U F.p;
```

```
F.L = [1.0 0.0 0.0; 0.3333333333333333 1.0 0.0; -0.6666666666666666 -0.5 1.0]
F.U = [3.0 -4.0 4.0; 0.0 3.333333333333333 0.6666666666666667; 0.0 0.0 8.0]
F.p = [2, 1, 3]
```

One can also use an appropriate tuple on the left side:

```
L, U, p = lu(A);
p
```

```
3-element Vector{Int64}:
2
1
3
```

The permutation vector indicates how the rows of the matrix have been permuted. It holds:

$$L \cdot U = PA$$

Julia's syntax of indirect indexing allows applying the row permutation with the notation `A[p,:]`:

```
display(A)
display(A[p,:])
display(L*U)
```

```
3×3 Matrix{Int64}:
1 2 2
3 -4 4
-2 1 5
3×3 Matrix{Int64}:
3 -4 4
1 2 2
-2 1 5
3×3 Matrix{Float64}:
3.0 -4.0 4.0
1.0 2.0 2.0
-2.0 1.0 5.0
```

Forward/backward substitution with an `LU`-object is performed by the `\` operator:

```
b = [1, 2, 3]
x = F \ b
```

```
3-element Vector{Float64}:
-0.10000000000000002
-0.012500000000000006
0.5625
```

Verification:

```
A * x - b
```

```
3-element Vector{Float64}:
0.0
0.0
0.0
```

In Julia, the `\` operator hides a quite universal "matrix solver" which perfoms implicitely an appropriate matrix factorization:

```
A \ b
```

```
3-element Vector{Float64}:
-0.10000000000000002
-0.012500000000000006
0.5625
```

13.3.2 QR Decomposition

The function `qr()` returns a special QR object that contains the components Q and R. The orthogonal (or unitary) matrix Q is in an optimized form. Conversion to a "normal" matrix is, as always, possible with `collect()` if needed.

```
F = qr(A)
@show typeof(F) typeof(F.Q)
display(collect(F.Q))
display(F.R)
```

```
typeof(F) = LinearAlgebra.QRCompactWY{Float64, Matrix{Float64}, Matrix{Float64}}
typeof(F.Q) = LinearAlgebra.QRCompactWYQ{Float64, Matrix{Float64}, Matrix{Float64}}
3×3 Matrix{Float64}:
-0.267261 0.872872 0.408248
-0.801784 -0.436436 0.408248
0.534522 -0.218218 0.816497
3×3 Matrix{Float64}:
-3.74166 3.20713 -1.06904
0.0 3.27327 -1.09109
0.0 0.0 6.53197
```

13.3.3 Appropriate Factorization

The function `factorize()` returns a factorization appropriate for the given matrix type; see documentation for details. This factorization can subsequently be utilized for multiple right-hand sides $\mathbf{b_1}, \mathbf{b_2}, \ldots$.

```
Af = factorize(A)
```

```
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3×3 Matrix{Float64}:
1.0 0.0 0.0
0.333333 1.0 0.0
-0.666667 -0.5 1.0
U factor:
3×3 Matrix{Float64}:
3.0 -4.0 4.0
0.0 3.33333 0.666667
0.0 0.0 8.0
```

```
Af \ [1, 2, 3]
```

```
3-element Vector{Float64}:
-0.10000000000000002
-0.012500000000000006
0.5625
```

```
Af \ [5, 7, 9]
```

```
3-element Vector{Float64}:
0.4000000000000001
0.42500000000000004
1.875
```

14. Characters, Strings, and Unicode

14.1 Character Encodings (Early History)

There were - depending on manufacturer, country, programming language, operating system, etc. - a large variety of encodings.

Still relevant today are:

14.1.1 ASCII

The American Standard Code for Information Interchange (ASCII) was published as a standard in the USA in 1963.

- It defines $2^7 = 128$ characters, namely:
  - 33 control characters, such as `newline`, `escape`, `end of transmission/file`, `delete`
  - 95 graphically printable characters:
    - 52 Latin letters `a-z, A-Z`
    - 10 digits `0-9`
    - 7 punctuation marks `.,:;?!"`
    - 1 space
    - 6 parentheses `[{()}]`
    - 7 mathematical operations `+-*/<>=`
    - 12 special characters `#$%&'\^_|~`@`
- ASCII is still the "lowest common denominator" in the encoding chaos.
- The first 128 Unicode characters are identical to ASCII.

14.1.2 ISO 8859 Character Sets

- ASCII uses only 7 bits.
- In a byte, you can fit another 128 characters by setting the 8th bit.
- In 1987/88, various 1-byte encodings were standardized in ISO 8859, all ASCII-compatible, including:

Encoding	Region	Languages
ISO 8859-1 (Latin-1)	Western Europe	German, French,..., Icelandic
ISO 8859-2 (Latin-2)	Eastern Europe	Slavic languages with Latin script
ISO 8859-3 (Latin-3)	Southern Europe	Turkish, Maltese,...
ISO 8859-4 (Latin-4)	Northern Europe	Estonian, Latvian, Lithuanian, Greenlandic, Sami
ISO 8859-5 (Latin/Cyrillic)	Eastern Europe	Slavic languages with Cyrillic script
ISO 8859-6 (Latin/Arabic)		
ISO 8859-7 (Latin/Greek)		
...		

Encoding	Region	Languages
ISO 8859-15 (Latin-9)		1999: Revision of Latin-1: now including Euro sign

14.2 Unicode

The goal of the Unicode Consortium is a uniform encoding for all scripts worldwide.

- Unicode version 1 was published in 1991
- Unicode version 17 was published in 2025 with 159,801 characters, including:
  - 172 scripts
  - mathematical and technical symbols
  - Emojis and other symbols, control and formatting characters
- Over 90,000 characters are assigned to the CJK scripts (Chinese/Japanese/Korean)

14.2.1 Technical Details

- Each character is assigned a `codepoint`, which is simply a sequential number written hexadecimally
  - either with 4 digit as `U+XXXX` (zeroth plane)
  - or with 6 digit as `U+XXXXXX` (further planes)
- Each plane ranges from `U+XY0000` to `U+XYFFFF`, thus containing $2^{16} = 65534$ characters.
- 17 planes `XY=00` to `XY=10` are provided, giving a value range from `U+0000` to `U+10FFFF`.
- Thus, a maximum of 21 bits per character are needed.
- The total number of possible codepoints is slightly less than 0x10FFFF, as certain areas are not used for technical reasons. It is about 1.1 million, so there is still much room.
- So far, codepoints have been assigned only from these planes:
  - Plane 0 = BMP (Basic Multilingual Plane) `U+0000 - U+FFFF`,
  - Plane 1 = SMP (Supplementary Multilingual Plane) `U+010000 - U+01FFFF`,
  - Plane 2 = SIP (Supplementary Ideographic Plane) `U+020000 - U+02FFFF`,
  - Plane 3 = TIP (Tertiary Ideographic Plane) `U+030000 - U+03FFFF`, and
  - Plane 14 = SSP (Supplementary Special-purpose Plane) `U+0E0000 - U+0EFFFF`.
- `U+0000` to `U+007F` is identical to ASCII
- `U+0000` to `U+00FF` is identical to ISO 8859-1 (Latin-1)

14.2.2 Properties of Unicode Characters

In the standard, each character is described by

- its codepoint (number)
- a name (which consists only of ASCII uppercase letters, digits, and hyphens) and
- various attributes such as
  - script direction
  - category: uppercase letter, lowercase letter, modifier letter, digit, punctuation, symbol, separator,....

In the Unicode standard, this looks like (simplified, only codepoint and name):

```
…
U+0041 LATIN CAPITAL LETTER A
U+0042 LATIN CAPITAL LETTER B
U+0043 LATIN CAPITAL LETTER C
U+0044 LATIN CAPITAL LETTER D
…
U+00E9 LATIN SMALL LETTER E WITH ACUTE
U+00EA LATIN SMALL LETTER E WITH CIRCUMFLEX
…
U+0641 ARABIC LETTER FEH
U+0642 ARABIC LETTER QAF
…
U+21B4 RIGHTWARDS ARROW WITH CORNER DOWNWARDS
…
```

What does 'RIGHTWARDS ARROW WITH CORNER DOWNWARDS' look like?
Julia uses `\U...` for input of Unicode codepoints.

```
'\U21b4'
```

```
'↴': Unicode U+21B4 (category So: Symbol, other)
```

14.2.3 A Selection of Scripts

A small helper function:

```
function printuc(c, n)
    for i in 1:n
        print(c + i -1)
        if i%70 == 0 print("\n") end
    end
end
```

```
printuc (generic function with 1 method)
```

Cyrillic

```
printuc('\U0400', 100)
```

```
ЀЁЂЃЄЅІЇЈЉЊЋЌЍЎЏАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфх
цчшщъыьэюяѐёђѓєѕіїјљњћќѝўџѠѡѢѣ
```

Tamil

```
printuc('\U0be7',20)
```

```
௧௨௩௪௫௬௭௮௯௰௱௲௳௴௵௶௷௸௹௺
```

Chess

```
printuc('\U2654', 12)
```

```
♔♕♖♗♘♙♚♛♜♝♞♟
```

Mathematical Operators

```
printuc('\U2200', 255)
```

```
∀∁∂∃∄∅∆∇∈∉∊∋∌∍∎∏∐∑−∓∔∕∖∗∘∙√∛∜∝∞∟∠∡∢∣∤∥∦∧∨∩∪∫∬∭∮∯∰∱∲∳∴∵∶∷∸∹∺∻∼∽∾∿≀≁≂≃≄≅
≆≇≈≉≊≋≌≍≎≏≐≑≒≓≔≕≖≗≘≙≚≛≜≝≞≟≠≡≢≣≤≥≦≧≨≩≪≫≬≭≮≯≰≱≲≳≴≵≶≷≸≹≺≻≼≽≾≿⊀⊁⊂⊃⊄⊅⊆⊇⊈⊉
⊊⊋⊌⊍⊎⊏⊐⊑⊒⊓⊔⊕⊖⊗⊘⊙⊚⊛⊜⊝⊞⊟⊠⊡⊢⊣⊤⊥⊦⊧⊨⊩⊪⊫⊬⊭⊮⊯⊰⊱⊲⊳⊴⊵⊶⊷⊸⊹⊺⊻⊼⊽⊾⊿⋀⋁⋂⋃⋄⋅⋆⋇⋈⋉⋊⋋⋌⋍⋎⋏
⋐⋑⋒⋓⋔⋕⋖⋗⋘⋙⋚⋛⋜⋝⋞⋟⋠⋡⋢⋣⋤⋥⋦⋧⋨⋩⋪⋫⋬⋭⋮⋯⋰⋱⋲⋳⋴⋵⋶⋷⋸⋹⋺⋻⋼⋽⋾⋿
```

Runes

```
printuc('\U16a0', 40)
```

```
ᚠᚡᚢᚣᚤᚥᚦᚧᚨᚩᚪᚫᚬᚭᚮᚯᚰᚱᚲᚳᚴᚵᚶᚷᚸᚹᚺᚻᚼᚽᚾᚿᛀᛁᛂᛃᛄᛅᛆᛇᛈᛉᛊᛋᛌᛍᛎᛏ
```

Phaistos Disc

- This script is not deciphered.
- It is unclear what language it represents.
- There is only one single document in this script: the Phaistos Disc from the Bronze Age.

```
printuc('\U101D0', 46 )
```

```
𐇐𐇑𐇒𐇓𐇔𐇕𐇖𐇗𐇘𐇙𐇚𐇛𐇜𐇝𐇞𐇟𐇠𐇡𐇢𐇣𐇤𐇥𐇦𐇧𐇨𐇩𐇪𐇫𐇬𐇭𐇮𐇯𐇰𐇱𐇲𐇳𐇴𐇵
```

14.2.4 Unicode Transformation Formats: UTF-8, UTF-16, UTF-32

Unicode transformation formats define how a sequence of codepoints is represented as a sequence of bytes.

Since codepoints are of different lengths, they cannot simply be written down one after the other. Where does one end and the next begin?

- **UTF-32**: The simplest but also most memory-intensive is to make them all the same length. Each codepoint is encoded in 4 bytes = 32 bits.
- In **UTF-16**, a codepoint is represented either with 2 bytes or with 4 bytes.
- In **UTF-8**, a codepoint is represented with 1, 2, 3, or 4 bytes.
- **UTF-8** is the format with the highest prevalence. Julia also uses it.

14.2.5 UTF-8

- For each codepoint, 1, 2, 3, or 4 full bytes are used.
- With variable-length encoding, you must be able to recognize which byte sequences belong together:
  - A byte of the form 0xxxxxxx represents an ASCII codepoint of length 1.
  - A byte of the form 110xxxxx starts a 2-byte code.
  - A byte of the form 1110xxxx starts a 3-byte code.
  - A byte of the form 11110xxx starts a 4-byte code.
  - All further bytes of a 2-, 3-, or 4-byte code have the form 10xxxxxx.
- Thus, the space available for the codepoint (number of x) is:
  - One-byte code: 7 bits
  - Two-byte code: 5 + 6 = 11 bits
  - Three-byte code: 4 + 6 + 6 = 16 bits
  - Four-byte code: 3 + 6 + 6 + 6 = 21 bits
- Thus, every ASCII text is automatically also a correctly encoded UTF-8 text.
- If the 17 planes (equivalent to 21 bits, resulting in approximately 1.1 million possible characters) currently defined in Unicode are ever depleted, UTF-8 can be extended to include 5- and 6-byte code sequences.

14.3 Characters and Strings in Julia

14.3.1 Characters

The `Char` type encodes a single Unicode character.

- Julia uses single quotes for characters: `'a'`.
- A `Char` occupies 4 bytes of memory and
- represents a Unicode codepoint.
- `Char`s can be converted to/from `UInt`s and
- the integer value is equal to the Unicode codepoint.

`Char`s can be converted to/from `UInt`s:

```
UInt('a')
```

```
0x0000000000000061
```

```
b = Char(0x2656)
```

```
'♖': Unicode U+2656 (category So: Symbol, other)
```

14.3.2 Strings

- In Julia, strings are denoted with double quotes: `"a"`.
- These strings are encoded in UTF-8, where a single character may consist of 1 to 4 bytes.

```
@show typeof('a') sizeof('a') typeof("a") sizeof("a");
```

```
typeof('a') = Char
sizeof('a') = 4
```

```
typeof("a") = String
sizeof("a") = 1
```

For a non-ASCII string, the number of bytes and the number of characters differ:

```
asciistr = "Hello World!"
@show length(asciistr) ncodeunits(asciistr);
```

```
length(asciistr) = 12
ncodeunits(asciistr) = 12
```

```
str = "😄 Hellö 🎶"
@show length(str) ncodeunits(str);
```

```
length(str) = 9
ncodeunits(str) = 16
```

Iterating over a string iterates over the characters:

```
for i in str
    println(i, "  ", typeof(i))
end
```

```
😄 Char
Char
H Char
e Char
l Char
l Char
ö Char
Char
🎶 Char
```

14.3.3 Concatenation of Strings

Strings with concatenation form a non-commutative monoid.
Therefore, Julia writes concatenation multiplicatively.

```
 str * asciistr * str
```

```
"😄 Hellö 🎶Hello World!😄 Hellö 🎶"
```

Powers with natural exponents are thus also defined.

```
str^3,  str^0
```

```
("😄 Hellö 🎶😄 Hellö 🎶😄 Hellö 🎶", "")
```

14.3.4 String Interpolation

The dollar sign serves a special purpose in strings, frequently utilized within `print()` statements. It enables the interpolation of variables or expressions.

```
a = 33.4
b = "x"

s = "The result for $b is equal to $a and the doubled square root of it is $(2sqrt(a))\n"
```

```
"The result for x is equal to 33.4 and the doubled square root of it is
11.55854662143991\n"
```

14.3.5 Backslash Escape Sequences

The backslash `\` also has a special function in string constants. Julia uses the backslash codings known from C and other languages for special characters, dollar signs, and backslashes themselves:

```
s = "This is how one gets \'quotes\" and a \$ sign and a\nline break and a \\ etc... "
print(s)
```

```
This is how one gets ‘quotes" and a $ sign and a
line break and a \ etc…
```

14.3.6 Triple Quotes

Strings may also be enclosed in triple quotes, preserving line breaks and embedded quotes:

```
s = """
 This should
be a "longer"
  'text'.
"""

print(s)
```

```
This should
be a "longer"
‘text’.
```

14.3.7 Raw Strings

In a `raw string`, all backslash escape sequences except for `\"` are disabled:

```
s = raw"A $ and a \ and two \\ and a 'bla'..."
print(s)
```

```
A $ and a \ and two \\ and a ‘bla’…
```

14.4 Further Functions for Characters and Strings (Selection)

14.4.1 Tests for Characters

```
@show isdigit('0') isletter('Ψ') isascii('\U2655') islowercase('α')
@show isnumeric('½') iscntrl('\n') ispunct(';');
```

```
isdigit(‘0’) = true
isletter(‘Ψ’) = true
isascii(‘♕’) = false
islowercase(‘α’) = true
isnumeric(‘½’) = true
iscntrl(‘\n’) = true
ispunct(‘;’) = true
```

14.4.2 Application to Strings

These tests can be used on strings with `all()`, `any()`, or `count()`:

```
all(ispunct, ";.:")
```

```
true
```

```
any(isdigit, "It is 3 o'clock! 🕒" )
```

```
true
```

```
count(islowercase, "Hello, du!!")
```

```
6
```

14.4.3 Other String Functions

```
@show startswith("Lampenschirm", "Lamp")  occursin("pensch", "Lampenschirm")
@show endswith("Lampenschirm", "irm");
```

```
startswith("Lampenschirm", "Lamp") = true
occursin("pensch", "Lampenschirm") = true
endswith("Lampenschirm", "irm") = true
```

```
@show uppercase("Eis") lowercase("Eis")  titlecase("eiSen");
```

```
uppercase("Eis") = "EIS"
lowercase("Eis") = "eis"
titlecase("eiSen") = "Eisen"
```

```
# remove newline from end of string

@show chomp("Eis\n")  chomp("Eis");
```

```
chomp("Eis\n") = "Eis"
chomp("Eis") = "Eis"
```

```
split("π is irrational.")
```

```
3-element Vector{SubString{String}}:
"π"
"is"
"irrational."
```

```
replace("π is irrational.", "is" ⇒ "is allegedly")
```

```
"π is allegedly irrational."
```

14.5 Indexing of Strings

Strings are immutable but indexable, with a few special features:

- The index numbers the bytes of the string.
- For a non-ASCII string, not all indices are valid because a valid index always addresses a Unicode character.

Our example string:

```
str
```

```
"😄 Hellö 🎶"
```

The first character

```
str[1]
```

```
'😄': Unicode U+1F604 (category So: Symbol, other)
```

This character is 4 bytes long in UTF-8 encoding. Thus, 2, 3, and 4 are invalid indices.

```
str[2]
```

```
StringIndexError: invalid index [2], valid nearby indices [1]⇒'😄', [5]⇒' '
Stacktrace:
 [1] string_index_err(s::AbstractString, i::Int64)
   @ Base ./strings/string.jl:12
 [2] getindex_continued(s::String, i::Int64, u::UInt32)
   @ Base ./strings/string.jl:473
 [3] getindex(s::String, i::Int64)
   @ Base ./strings/string.jl:465
```

```
 [4] top-level scope
   @ ~/Julia/Book26/JuliaBook/chapters/10_Strings.qmd:461
```

Only the 5th byte is a new character:

```
str[5]
```

```
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Even when addressing substrings, start and end must always be valid indices; i.e., the end index must also index the first byte of a character, and that character is the last of the substring.

```
str[1:7]
```

```
"😄 He"
```

The function `eachindex()` returns an iterator over the valid indices:

```
for i in eachindex(str)
    c = str[i]
    println("$i: $c")
end
```

```
1: 😄
5:
6: H
7: e
8: l
9: l
10: ö
12:
13: 🎶
```

As usual, `collect()` makes an iterator into a vector.

```
collect(eachindex(str))
```

```
9-element Vector{Int64}:
1
5
6
7
8
9
10
12
13
```

The function `nextind()` returns the next valid index.

```
@show nextind(str, 1) nextind(str, 2);
```

```
nextind(str, 1) = 5
nextind(str, 2) = 5
```

Why does Julia use a byte index instead of a character index? The main reason is the efficiency of indexing.

- In a long string (e.g., book text), the position `s[123455]` can be found quickly with a byte index.
- A character index would have to traverse the entire string in UTF-8 encoding to find the n-th character, since characters can be 1, 2, 3, or 4 bytes long.

Some functions return indices or ranges as results. They always return valid indices:

```
findfirst('l', str)
```

```
8
```

```
findfirst("Hel", str)
```

```
6:8
```

```
str2 = "αβγδϵ"^3
```

```
"αβγδϵαβγδϵαβγδϵ"
```

```
n = findfirst('γ', str2)
```

```
5
```

So you can continue searching from the next valid index after `n=5`:

```
findnext('γ', str2, nextind(str2, n))
```

```
15
```

15. Input and Output

15.1 Console

The operating system typically provides three channels (*streams*) for a program:

- Standard input (`stdin`)
- Standard output (`stdout`)
- Standard error (`stderr`)

When executed in a terminal, console, or shell, the program reads keyboard input through `stdin` and outputs to the terminal via `stdout` and `stderr`.

- Writing to `stdout`: `print()`, `println()`, `printstyled()`
- Writing to `stderr`: `print(stderr,...)`, `println(stderr,...)`, `printstyled(stderr,...)`
- Reading from `stdin`: `readline()`

15.1.1 Input

The *Python* language provides an `input()` function:

```
ans = input("Please enter a positive number!")
```

It prints the prompt, waits for input, and returns a `string`.

In Julia, you can implement this function as follows:

```
function input(prompt = "Input:")
    println(prompt)
    flush(stdout)
    return chomp(readline())
end
```

```
input (generic function with 2 methods)
```

Comments

- Write operations are buffered by modern operating systems. `flush(stdout)` empties the buffer and forces the write operation to complete immediately.
- `readline()` returns a string ending with a newline (`\n`). The function `chomp()` removes a trailing line break from the string.

```
a = input("Please enter two numbers!")
```

```
"34 56"
```

15.1.2 Processing the Input

`split(str)` splits a string into "words", returning a string array:

```
av = split(a)
```

```
2-element Vector{SubString{String}}:
"34"
"56"
```

`parse(T, str)` tries to convert `str` to type `T`:

```
v = parse.(Int, av)
```

```
2-element Vector{Int64}:
34
56
```

`parse()` throws an error if the string cannot be parsed as type `T`. You can catch the error with `try/catch`, or use `tryparse(T, str)`, which returns `nothing` in such cases. Test the result with `isnothing()`.

15.2 Formatted Output with the `Printf` Macro

You often need to output numbers or strings with strict formatting: total length, decimal places, alignment, etc.

For this purpose, the `Printf` package defines the macros `@sprintf` and `@printf`, which work similarly to the corresponding C functions.

```
using Printf

x = 123.7876355638734

@printf("Output right-aligned with max. 10 character width and 3 decimal places: x =
%10.3f", x)
```

```
Output right-aligned with max. 10 character width and 3 decimal places: x =  123.788
```

The first argument is a string containing placeholders (here: `%10.3f`) for the variables, followed by the variables themselves.

Placeholders have the form:

```
%[flags][width][.precision]type
```

where entries in square brackets are optional.

Type specifications in placeholders

`%s`	`string`
`%i`	`integer`
`%o`	`integer, octal (base 8)`
`%x, %X`	`integer, hexadecimal (base 16), digits 0-9a-f or 0-9A-F`
`%f`	`floating point`
`%e`	`floating point, scientific notation`
`%g`	`floating point, %f or %e as appropriate`

Flags

Plus sign	right-aligned (default)
Minus sign	left-aligned
Zero	adds leading zeros

Width

```
Minimum number of characters used (more will be taken if necessary)
```

15.2.1 Examples

```
using Printf    # Load the package first
```

```
@printf("|%s|", "Hello")     # string with placeholder for string
```

```
|Hello|
```

The vertical bars are not part of the placeholder; they indicate the output field boundaries.

```
@printf("|%10s|", "Hello")   # Minimum length, right-aligned
```

```
| Hello|
```

```
@printf("|%-10s|", "Hello")     # left-aligned
```

```
|Hello |
```

```
@printf("|%3s|", "Hello")     # Length specification can be exceeded
                              # Better a badly formatted table than wrong values!
```

```
|Hello|
```

```
j = 123
k = 90019001
l = 3342678

@printf("j = %012i, k = %-12i, l = %12i", j, k, l)   # 0-flag for leading zeros
```

```
j = 000000000123, k = 90019001 , l = 3342678
```

`@printf` and `@sprintf` can be called like functions:

```
@printf("%i %i", 22, j)
```

```
22 123
```

or as macros, i.e., without parentheses or commas:

```
@printf "%i %i" 22 j
```

```
22 123
```

`@printf` can take a stream as its first argument; otherwise, the argument list consists of:

- format string with placeholders
- variables matching the placeholders in number and type

```
@printf(stderr, "First result: %i %s\nSecond result %i",
                                  j, "(estimated)", k)
```

```
First result: 123 (estimated)
Second result 90019001
```

The macro `@sprintf` does not print; it returns the formatted string:

```
str = @sprintf("x = %10.6f", π );
```

```
str
```

```
"x = 3.141593"
```

15.2.2 Formatting Floating-Point Numbers

The *precision* value specifies:

- `%f` and `%e` formats: maximum decimal places
- `%g` format: maximum total digits (integer part + decimal places)

```
x = 123456.7890123456

@printf("%20.4f   %20.4e", x, x)     # 4 decimal places
```

```
123456.7890 1.2346e+05
```

```
@printf("%20.7f %20.7e", x, x)     # 7 decimal places
```

```
123456.7890123 1.2345679e+05
```

```
@printf("%20.7g %20.4g", x, x)    # 7 and 4 digits total, respectively
```

```
123456.8 1.235e+05
```

15.3 File Operations

Files are handled by:

- Opening ⇒ creation of a new *stream* object (in addition to `stdin`, `stdout`, `stderr`)
- Reading from and writing to this *stream*
- Closing ⇒ detachment of the *stream* object from the file

```
stream = open(path, mode)
```

- path: filename or path
- mode:

```
"r" read, opens at file beginning
"w" write, opens at file beginning (file is created or overwritten)
"a" append, opens to continue writing at file end
```

Let's write a file:

```
file = open("myfile.txt", "w")
```

```
IOStream(<file myfile.txt>)
```

```
@printf(file, "%10i\n", k)
```

```
println(file, " second line")
```

```
close(file)
```

Let's look at the file:

```
;cat myfile.txt
```

```
90019001
second line
```

...and now we open it again for reading:

```
stream = open("myfile.txt", "r")
```

```
IOStream(<file myfile.txt>)
```

`readlines(stream)` returns all lines of a text file as a string vector.
`eachline(stream)` returns an iterator over the file lines.

```
n = 0
for line in eachline(stream)    # Read line by line
    n += 1
    println(n, line)            # Print with line number
end
close(stream)
```

```
1 90019001
2 second line
```

15.4 Packages for File Formats

Julia packages for various file formats include:

- PrettyTables.jl Output formatted tables
- DelimitedFiles.jl Read and write matrices
- CSV.jl Read and write CSV files
- XLSX.jl Read and write Excel files

and many more...

15.4.1 DelimitedFiles.jl

This package offers convenient functions for saving and reading matrices using `writedlm()` and `readdlm()`.

```
using DelimitedFiles
```

Generate a 200×3 matrix of random numbers:

```
A = rand(200,3)
```

```
200×3 Matrix{Float64}:
0.242052 0.139977 0.0213735
0.860452 0.921534 0.135449
0.780965 0.6372 0.416209
0.284369 0.76668 0.0325718
0.901622 0.0636347 0.833439
0.796214 0.984548 0.0332568
0.81308 0.568994 0.414981
0.188305 0.865349 0.908197
0.166486 0.65975 0.901737
0.204107 0.141201 0.0720358
⋮
0.31036 0.40433 0.513737
0.762837 0.412597 0.0890357
0.148856 0.314174 0.843368
0.563406 0.537396 0.694132
0.929565 0.264838 0.265931
0.232175 0.742833 0.409047
0.816274 0.642768 0.23883
0.586098 0.973815 0.240567
0.302521 0.683097 0.368841
```

and save it:

```
f = open("data2.txt", "w")
writedlm(f, A)
close(f)
```

The written file starts like this:

```
;head data2.txt
```

```
0.24205199766594254 0.1399771615290385 0.021373493695757695
0.860451688227497 0.9215337499514452 0.13544883014638742
0.780964976014078 0.6371995170230044 0.4162091214722937
0.28436852057305706 0.7666802395700884 0.03257182903393552
0.901622476189251 0.06363468493606483 0.8334389353153023
0.7962141348779413 0.9845483637051188 0.03325683070660668
0.8130801601016981 0.5689943195747339 0.41498081083566574
0.1883049395764308 0.8653492832736521 0.9081965329945879
0.16648615248459198 0.6597504707341652 0.9017373388815852
0.20410724461059393 0.14120130273969866 0.07203584538037122
```

Reading it back is simple:

```
B = readdlm("data2.txt")
```

```
200×3 Matrix{Float64}:
0.242052 0.139977 0.0213735
0.860452 0.921534 0.135449
0.780965 0.6372 0.416209
0.284369 0.76668 0.0325718
0.901622 0.0636347 0.833439
0.796214 0.984548 0.0332568
0.81308 0.568994 0.414981
0.188305 0.865349 0.908197
0.166486 0.65975 0.901737
0.204107 0.141201 0.0720358
⋮
0.31036 0.40433 0.513737
0.762837 0.412597 0.0890357
0.148856 0.314174 0.843368
0.563406 0.537396 0.694132
0.929565 0.264838 0.265931
0.232175 0.742833 0.409047
0.816274 0.642768 0.23883
0.586098 0.973815 0.240567
0.302521 0.683097 0.368841
```

In Julia, the `do` notation is frequently utilized for file handling (see Section 10.6). The `open()` function includes methods where the first argument is a `function(iostream)`. This function is applied to the stream, which is automatically closed afterward. The `do` notation allows to define this function anonymously:

```
open("data2.txt", "w") do io
    writedlm(io, A)
end
```

15.4.2 CSV and DataFrames

- The CSV format provides tables readable by MS Excel and other applications.
- Example: the weather and climate database *Meteostat*.
- The DataFrames.jl package handles tabular data conveniently.

```
using CSV, DataFrames, Downloads
# Weather data from Westerland (see https://dev.meteostat.net/bulk/hourly.html)

url = "https://bulk.meteostat.net/v2/hourly/10018.csv.gz"
http_response = Downloads.download(url)
file = CSV.File(http_response, header=false);
```

The data looks like this:

```
# https://dev.meteostat.net/bulk/hourly.html#endpoints
#
# Column 1  Date
#        2  Time (hour)
#        3  Temperature
#        5  Humidity
#        6  Precipitation
#        8 Wind direction
#        9 Wind speed

df =  DataFrame(file)
```

Row	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11	Column12	Column13
	Date	Int64	Float64?	Float64?	Int64?	Float64?	Missing	Int64?	Float64?	Float64?	Float64?	Missing	Int64?
1	1989-03-187	6.0	*missing*	*missing*	*missing*	*missing*	270	11.2	*missing*	*missing*	*missing*	*missing*	

Row	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11	Column12	Column13
	Date	Int64	Float64?	Float64?	Int64?	Float64?	Missing	Int64?	Float64?	Float64?	Float64?	Missing	Int64?
2	1989-03-18	8	6.0	−1.9	57	*missing*	*missing*	250	11.2	*missing*	*missing*	*missing*	*missing*
3	1989-03-18	9	7.0	−3.0	49	*missing*	*missing*	250	14.8	*missing*	*missing*	*missing*	*missing*
4	1989-03-18	10	7.0	−1.9	53	*missing*	*missing*	250	18.4	*missing*	*missing*	*missing*	*missing*
5	1989-03-18	11	8.0	−1.0	53	*missing*	*missing*	220	24.1	*missing*	*missing*	*missing*	*missing*
6	1989-03-18	12	8.0	0.0	57	*missing*	*missing*	240	27.7	*missing*	*missing*	*missing*	*missing*
7	1989-03-18	13	8.0	−1.0	53	*missing*	*missing*	240	25.9	*missing*	*missing*	*missing*	*missing*
8	1989-03-18	14	8.0	0.0	57	*missing*	*missing*	230	29.5	*missing*	*missing*	*missing*	*missing*
9	1989-03-18	15	8.0	0.0	57	*missing*	*missing*	230	31.7	*missing*	*missing*	*missing*	*missing*
10	1989-03-23	9	7.0	−0.9	57	*missing*	*missing*	280	37.1	*missing*	*missing*	*missing*	*missing*
11	1989-03-23	10	7.0	−0.9	57	*missing*	*missing*	270	64.8	*missing*	*missing*	*missing*	*missing*
12	1989-03-23	11	7.0	−0.9	57	*missing*	*missing*	280	33.5	*missing*	*missing*	*missing*	*missing*
13	1989-03-27	7	6.0	4.0	87	*missing*	*missing*	160	11.2	*missing*	*missing*	*missing*	*missing*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
189269	2025-08-23	7	19.8	14.2	70	*missing*	*missing*	332	22.2	31.5	1011.7	*missing*	2
189270	2025-08-23	8	19.2	14.2	73	*missing*	*missing*	342	22.2	31.5	1011.7	*missing*	2
189271	2025-08-23	9	18.6	14.3	76	*missing*	*missing*	343	20.4	29.6	1011.9	*missing*	2
189272	2025-08-23	10	18.7	14.0	74	*missing*	*missing*	351	20.4	27.8	1013.2	*missing*	2
189273	2025-08-23	11	18.3	14.0	76	*missing*	*missing*	354	20.4	27.8	1013.4	*missing*	2
189274	2025-08-23	12	18.0	13.9	77	*missing*	*missing*	2	20.4	27.8	1013.5	*missing*	2
189275	2025-08-23	13	17.9	13.6	76	*missing*	*missing*	3	20.4	29.6	1012.4	*missing*	2
189276	2025-08-24	0	17.9	13.6	76	*missing*	*missing*	7	20.4	31.5	1012.4	*missing*	2

Row	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11	Column12	Column13
	Date	Int64	Float64?	Float64?	Int64?	Float64?	Missing	Int64?	Float64?	Float64?	Float64?	Missing	Int64?
189277	2025-08-241	17.7	13.6	77	*missing*	*missing*	13	18.5	31.5	1012.2	*missing*	2	
189278	2025-08-242	17.0	13.7	81	*missing*	*missing*	14	18.5	31.5	1012.1	*missing*	2	
189279	2025-08-243	16.9	13.8	82	*missing*	*missing*	18	18.5	31.5	1012.2	*missing*	2	
189280	2025-08-244	17.0	13.7	81	*missing*	*missing*	9	20.4	31.5	1012.1	*missing*	*missing*	

For convenient plotting and date/time handling, we load two packages:

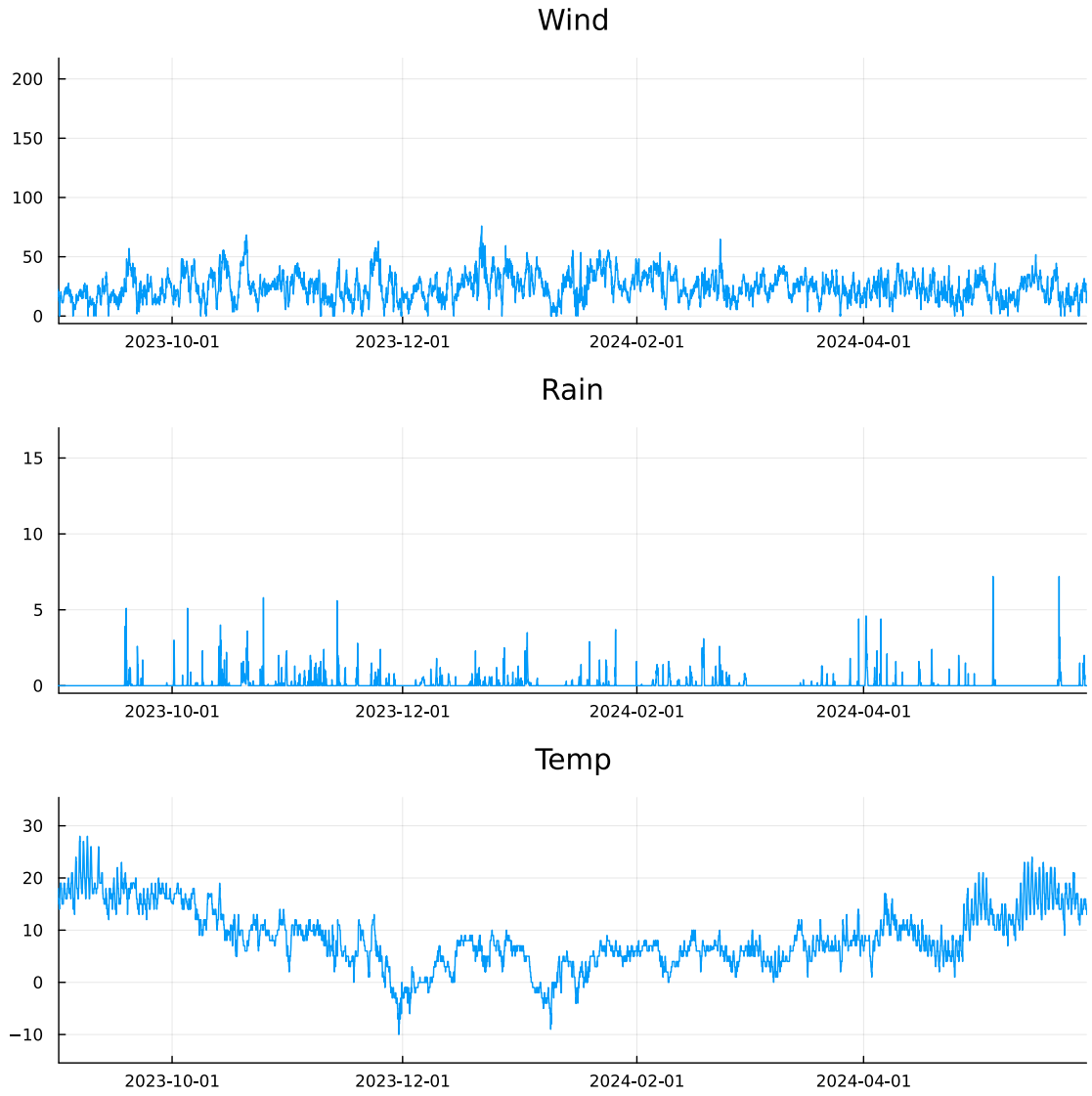
```
using StatsPlots, Dates
```

We create a new column that combines date (from column 1) and time (from column 2):

```
# new column combining col. 1 and 2 (date & time)

df[!, :datetime] = DateTime.(df.Column1) .+ Hour.(df.Column2);
```

The resulting plot:

```
@df df plot(:datetime, [:Column9, :Column6, :Column3],
            xlims = (DateTime(2023,9,1), DateTime(2024,5,30)),
            layout=(3,1), title=["Wind" "Rain" "Temp"],
            legend=:none, size=(800,800))
```


Wind
200
150
100
50
0
2023-10-01
2023-12-01
2024-02-01
2024-04-01
Rain
15
10
5
0
2023-10-01
2023-12-01
2024-02-01
2024-04-01
Temp
30
20
10
0
−10
2023-10-01
2023-12-01
2024-02-01
2024-04-01

16. Plots and Data Visualization in Julia: *Plots.jl*

Julia has numerous graphics packages. Two frequently used ones are Makie.jl and Plots.jl. Before presenting `Plots.jl` in detail, we list some others.

16.1 Brief Overview: Some Graphics Packages

Package/Documentation	Tutorial	Examples	Remarks
Plots.jl	Tutorial	Gallery	designed as a unified interface to various *backends* (graphics libraries)
Makie.jl	Basic tutorial	Beautiful Makie	"data visualization ecosystem for Julia", backends: Cairo (vector graphics), OpenGL, WebGL
PlotlyJS.jl	Getting started	Examples	Interface to the Plotly JavaScript graphics library
Gadfly.jl	Tutorial	Gallery	"a plotting and data visualization system written in Julia, influenced by R's ggplot2"
Bokeh.jl		Gallery	Julia frontend for Bokeh
VegaLite.jl	Tutorial	Examples	Julia frontend for Vega-Lite
Luxor.jl	Tutorial	Examples	General vector graphics/illustrations
Javis.jl	Tutorials	Examples	*Animated* vector graphics
TidierPlots.jl	Reference		"is a 100% Julia implementation of the R package ggplot2 powered by Makie.jl"
PythonPlot.jl		Examples (in Python)	Interface to Matplotlib (Python), 1:1 transfer of the Python API, therefore see Matplotlib documentation

16.2 Plots.jl

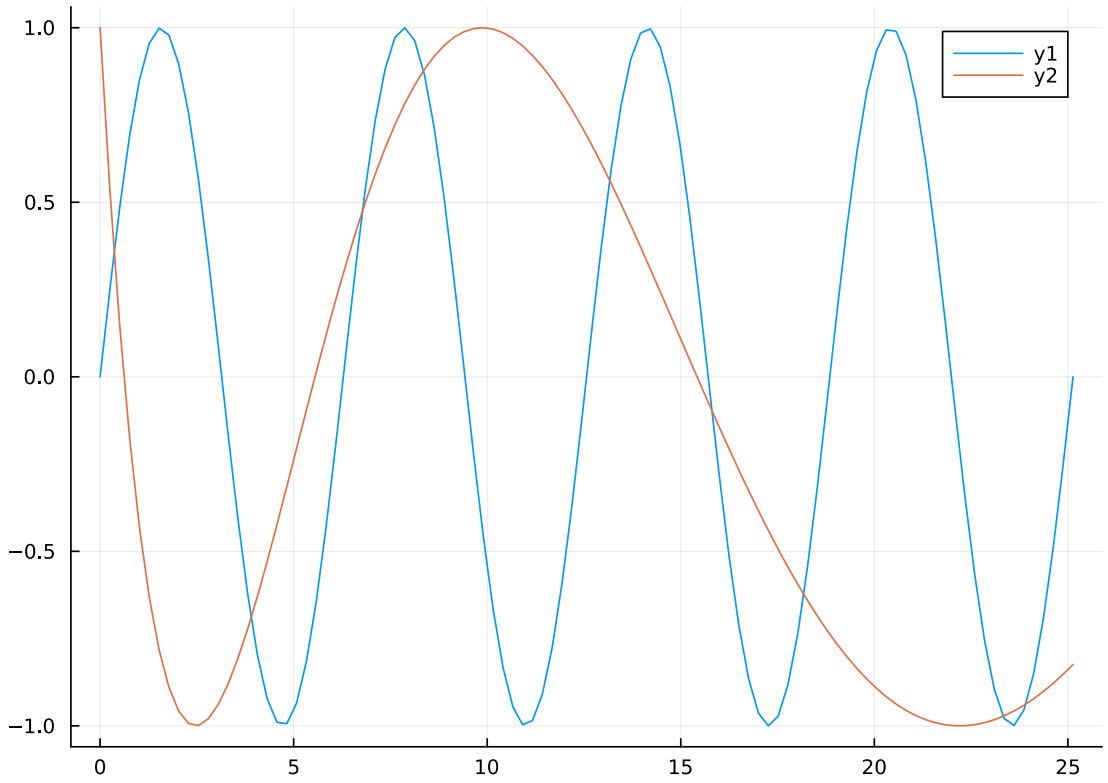
16.2.1 Simple Plots

The `plot()` function expects, in the simplest case:

- as the first argument a vector of x-values of length n and
- as the second argument a vector of the same length with the corresponding y-values.
- The second argument can also be an $n \times m$ matrix. Each column is treated as a separate graph (called a `series` in the documentation), plotting m curves:

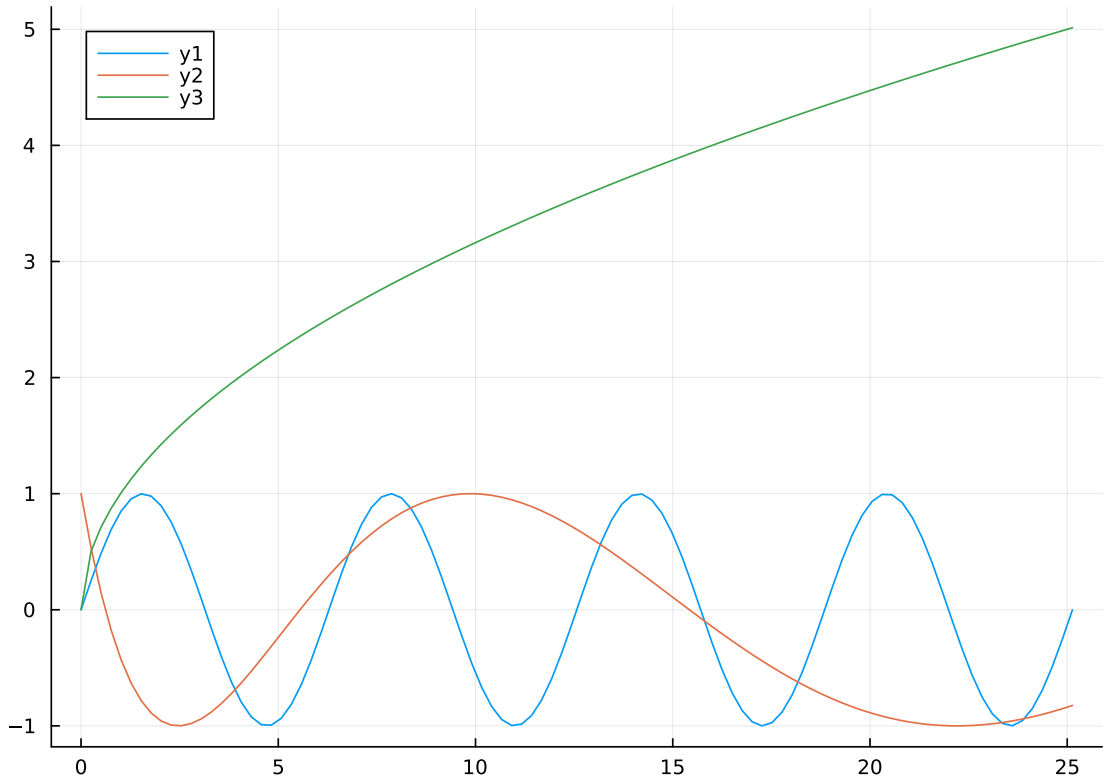
```
using Plots

x  = range(0, 8π; length = 100)
sx = @. sin(x)          # the @. macro broadcasts (vectorizes) every operation
cx = @. cos(2x^(1/2))
```

```
plot(x, [sx cx])
```



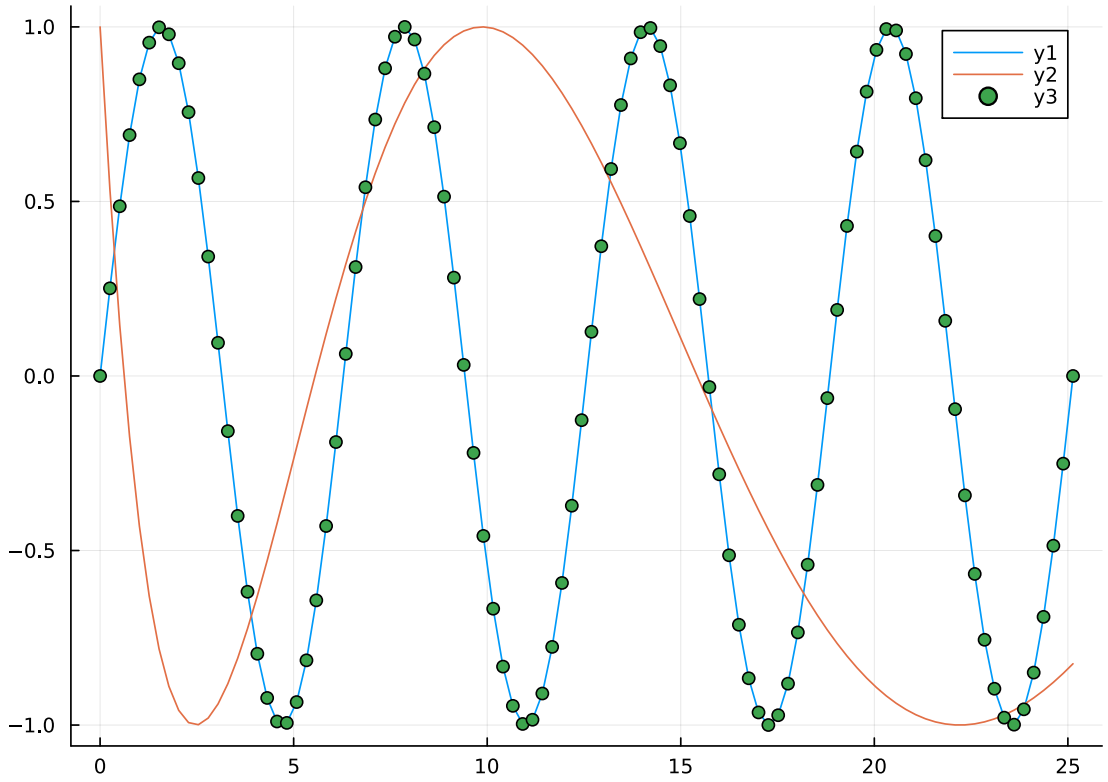
- Functions like `plot()`, `scatter()`, `contour()`, `heatmap()`, `histogram()`, `bar()`, etc. from *Plots.jl* all start a new plot.
- The versions `plot!()`, `scatter!()`, `contour!()`, `heatmap!()`, `histogram!()`, `bar!()`, etc. extend an existing plot:

```
plot(x, sx)          # plot only sin(x)
plot!(x, cx)         # add second graph
plot!(x, sqrt.(x))   # add a third one
```



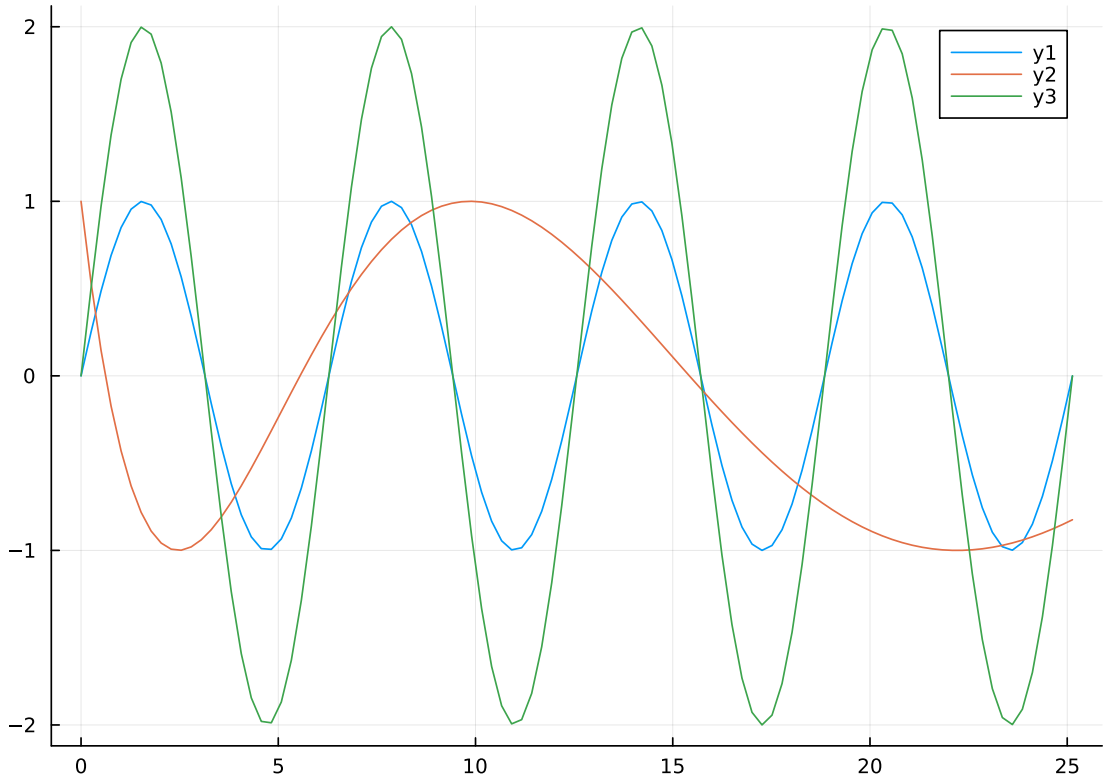
Plots are objects that can be assigned. Then they can be used later, copied, and in particular extended with the `!` functions:

```
plot1 = plot(x, [sx cx])
plot1a = deepcopy(plot1)    # plot objects are quite deep structures
scatter!(plot1, x, sx)      # add scatter plot, i.e. unconnected data points
```



The copied version `plot1a` remains unchanged by the `scatter!` call and can be used independently:

```
plot!(plot1a, x, 2 .* sx)
```



Plot objects can be saved as graphics files (PDF, SVG, PNG, etc.):

```
savefig(plot1, "plot.png")
```

```
"/home/hellmund/Julia/Book26/JuliaBook/chapters/plot.png"
```

```
;ls -l plot.png
```

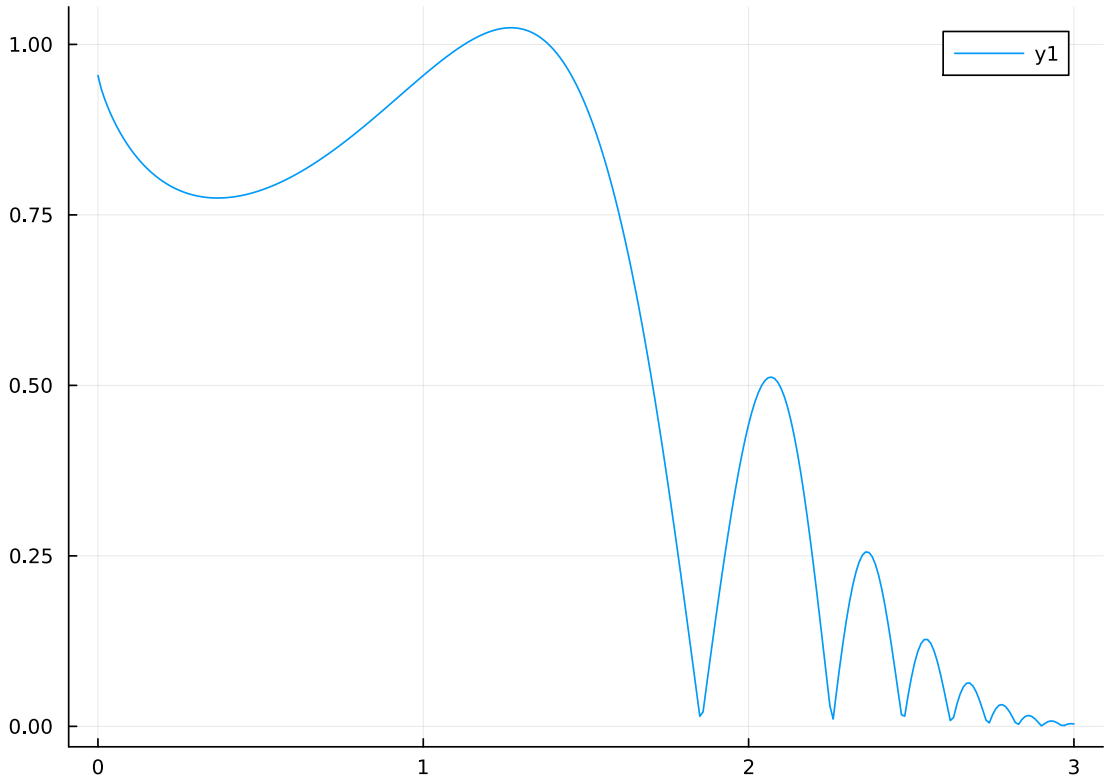
```
-rw-r-r- 1 hellmund hellmund 55780 Mar 11 15:51 plot.png
```

Plot objects can also be inserted as subplots into other plots, see section Section 16.2.7.

16.2.2 Function Plots

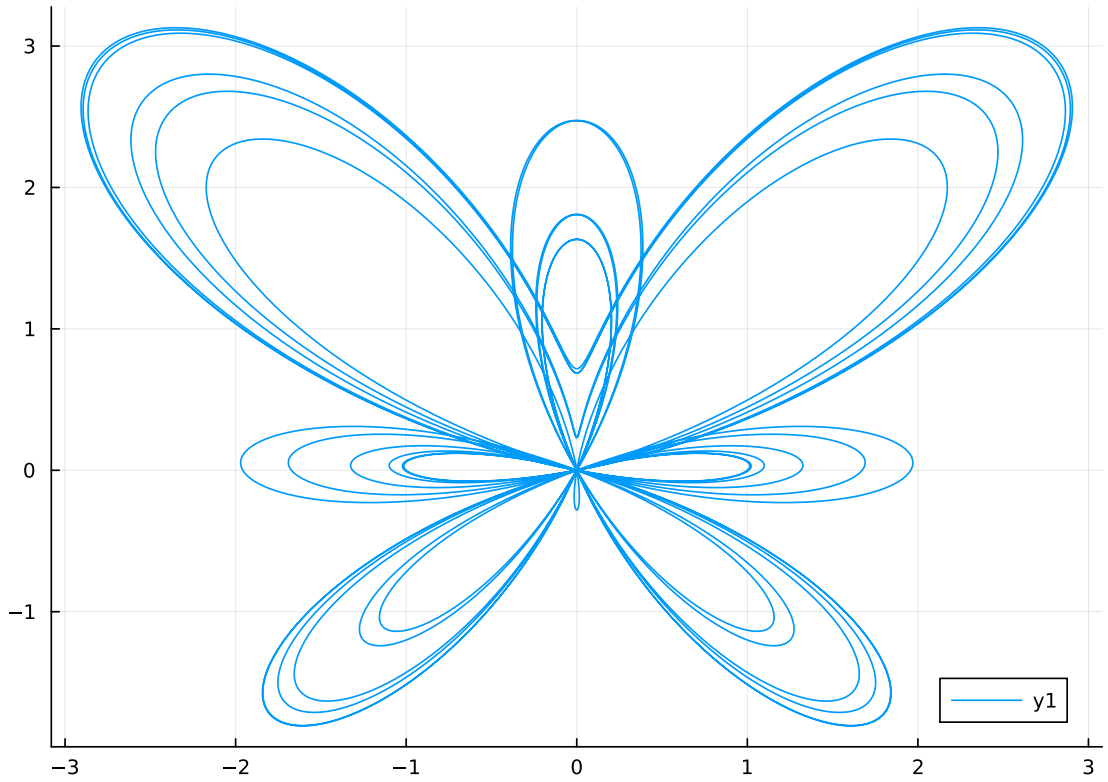
`plot()` can also be passed a function and a vector of x-values:

```
# https://mzrg.com/math/graphs.shtml

f(x) = abs(sin(x^x)/2^((x^x-π/2)/π))

plot(f, 0:0.01:3)
```



The parametric form $x = x(t), y = y(t)$ is plotted by passing two functions and a vector of t-values to `plot()`.

```
# https://en.wikipedia.org/wiki/Butterfly_curve_(transcendental)

xt(t) = sin(t) * (exp(cos(t))-2cos(4t)-sin(t/12)^5)
yt(t) = cos(t) * (exp(cos(t))-2cos(4t)-sin(t/12)^5)

plot(xt, yt, 0:0.01:12π)
```



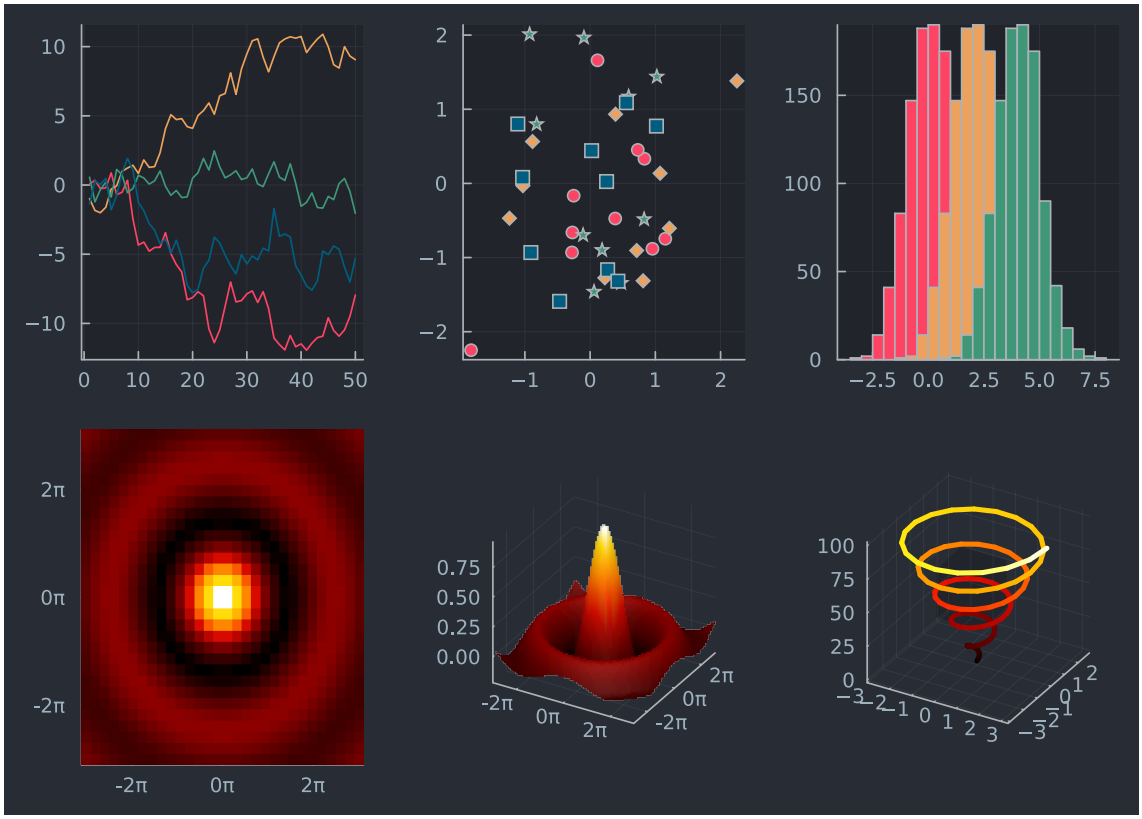
16.2.3 Plot Themes

"PlotThemes is a package to spice up plots made with Plots.jl."
See the illustrated list of themes

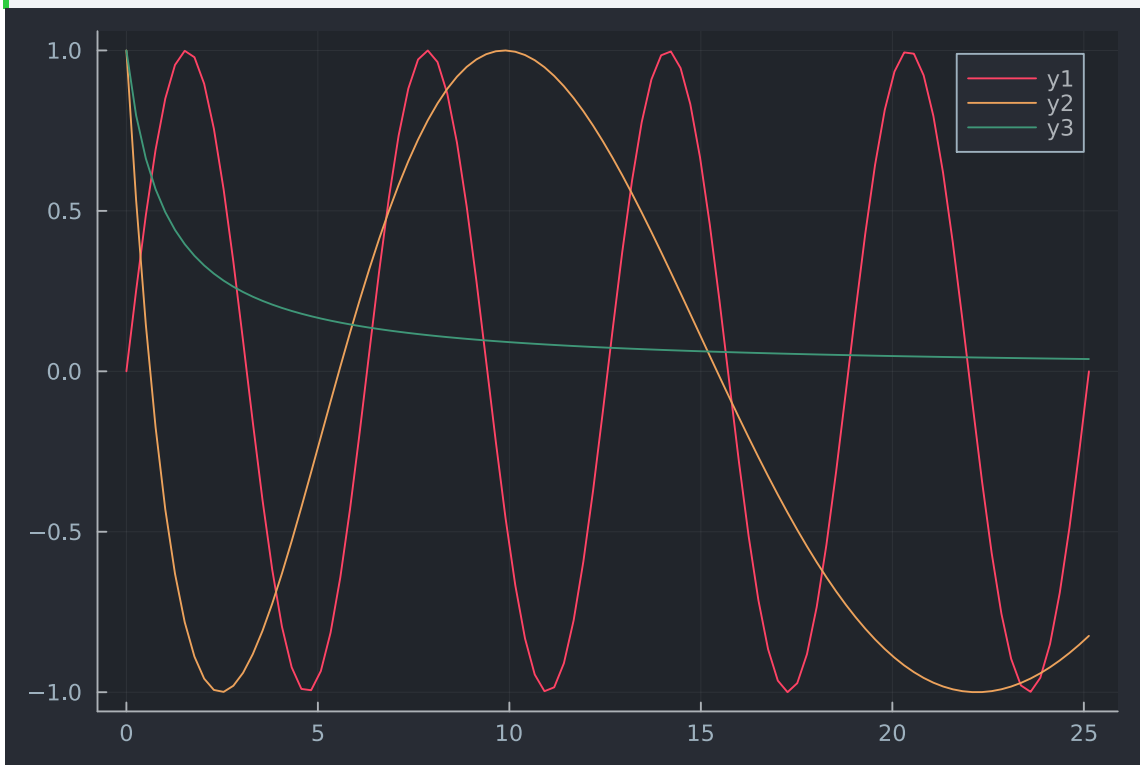
```
using PlotThemes

# list of themes
keys(PlotThemes._themes)
```

```
KeySet for a Dict{Symbol, PlotTheme} with 21 entries. Keys:
:juno
:default
:dao
:ggplot2
:gruvbox_dark
:dark
:gruvbox_light
:solarized
:wong
:dracula
:solarized_light
:mute
:boxed
:rose_pine
:wong2
:sand
:lime
:rose_pine_dawn
:bright
⋮
```

```
Plots.showtheme(:juno)
```




```
using PlotThemes

theme(:juno)   # set a theme for all further plots

plot(x, [sx  cx  1 ./ (1 .+ x)])
```



16.2.4 Plot Attributes

The `Plots.jl` functions have numerous options. Attributes are divided into 4 groups:

```
plotattr(:Plot)  # attributes for the overall plot
```

Defined Plot attributes are: background_color, background_color_outside, display_type, dpi, extra_kwargs, extra_plot_kwargs, fontfamily, foreground_color, html_output_format, inset_subplots, layout, link, overwrite_figure, plot_title, plot_titlefontcolor, plot_titlefontfamily, plot_titlefonthalign, plot_titlefontrotation, plot_titlefontsize, plot_titlefontvalign, plot_titleindex, plot_titlelocation, plot_titlevspan, pos, show, size, tex_output_standalone, thickness_scaling, warn_on_unsupported, window_title

```
plotattr(:Subplot) # attributes for a subplot
```

Defined Subplot attributes are: annotationcolor, annotationfontfamily, annotationfontsize, annotationhalign, annotationrotation, annotations, annotationvalign, aspect_ratio, background_color_inside, background_color_subplot, bottom_margin, camera, clims, color_palette, colorbar, colorbar_continuous_values, colorbar_discrete_values, colorbar_fontfamily, colorbar_formatter, colorbar_scale, colorbar_tickfontcolor, colorbar_tickfontfamily, colorbar_tickfonthalign, colorbar_tickfontrotation, colorbar_tickfontsize, colorbar_tickfontvalign, colorbar_ticks, colorbar_title, colorbar_title_location, colorbar_titlefontcolor, colorbar_titlefontfamily, colorbar_titlefonthalign, colorbar_titlefontrotation, colorbar_titlefontsize, colorbar_titlefontvalign, extra_kwargs, fontfamily_subplot, foreground_color_subplot, foreground_color_title, framestyle, left_margin, legend_background_color, legend_column, legend_font, legend_font_color, legend_font_family, legend_font_halign, legend_font_pointsize, legend_font_rotation, legend_font_valign, legend_foreground_color, legend_position, legend_title, legend_title_font, legend_title_font_color, legend_title_font_family, legend_title_font_halign, legend_title_font_pointsize, legend_title_font_rotation, legend_title_font_valign, margin, projection, projection_type, right_margin, subplot_index, title, titlefontcolor, titlefontfamily, titlefonthalign, titlefontrotation, titlefontsize, titlefontvalign, titlelocation, top_margin

```
plotattr(:Axis)  # attributes for an axis
```

Defined Axis attributes are: discrete_values, draw_arrow, flip, foreground_color_axis, foreground_color_border, foreground_color_grid, foreground_color_guide, foreground_color_minor_grid, foreground_color_text, formatter, grid, gridalpha, gridlinewidth, gridstyle, guide, guide_position, guidefontcolor, guidefontfamily, guidefonthalign, guidefontrotation, guidefontsize, guidefontvalign, lims, link, minorgrid, minorgridalpha, minorgridlinewidth, minorgridstyle, minorticks, mirror, rotation, scale, showaxis, tick_direction, tickfontcolor, tickfontfamily, tickfonthalign, tickfontrotation, tickfontsize, tickfontvalign, ticks, unit, unitformat, widen

```
plotattr(:Series) # attributes for a series, e.g., a line in the plot
```

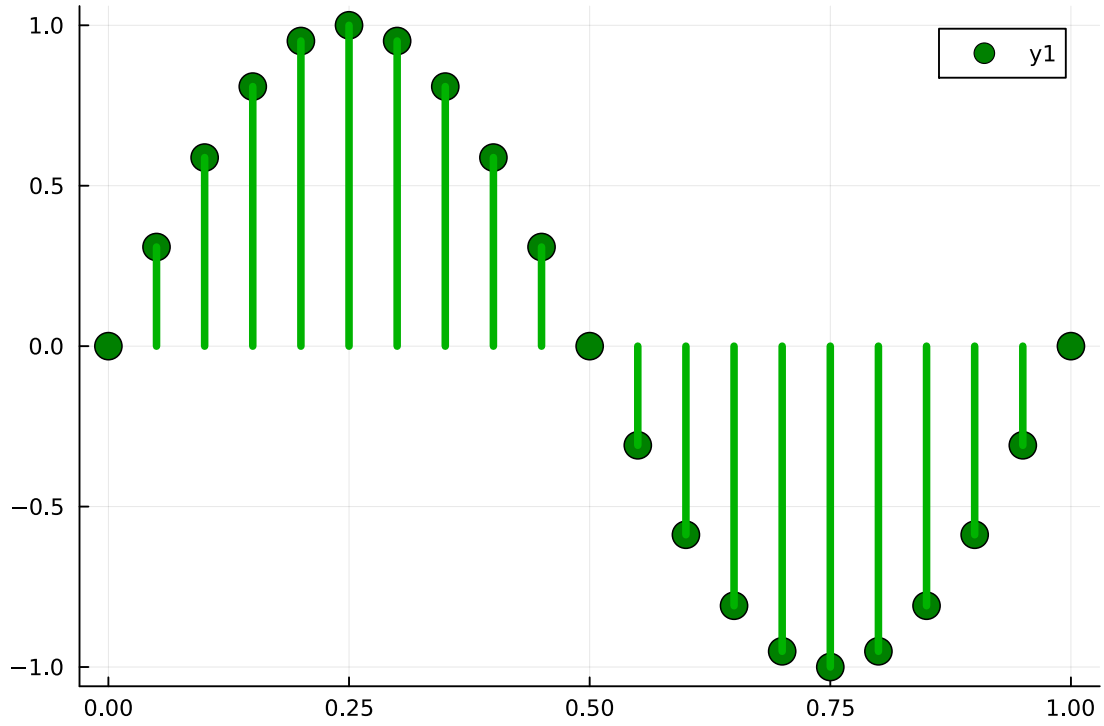
Defined Series attributes are: arrow, bar_edges, bar_position, bar_width, bins, colorbar_entry, connections, contour_labels, contours, extra_kwargs, fill_z, fillalpha, fillcolor, fillrange, fillstyle, group, hover, label, levels, line_z, linealpha, linecolor, linestyle, linewidth, marker_z, markeralpha, markercolor, markershape, markersize, markerstrokealpha, markerstrokecolor, markerstrokestyle, markerstrokewidth, normalize, orientation, permute, primary, quiver, ribbon, series_annotations, seriesalpha, seriescolor, seriestype, show_empty_bins, smooth, stride, subplot, weights, x, xerror, y, yerror, z, z_order, zerror
You can also query what individual attributes mean and which values are allowed:

```
plotattr("linestyle")
```

```
:linestyle

Style of the line (for path and bar stroke). Choose from
[:auto, :solid, :dash, :dot, :dashdot, :dashdotdot]

Aliases: (:linestyles, :ls, :s, :style).

Type: Symbol.

Series attribute, defaults to solid.
```

An example:

```
theme(:default)  # return to default theme

x = 0:0.05:1
y = sin.(2π*x)

plot(x, y, seriestype = :sticks, linewidth = 4, seriescolor = "#00b300",
        marker = :circle, markersize = 8, markercolor = :green,
)
```



Many specifications can be abbreviated significantly; see, e.g., the `Aliases:` in the output of `plotattr("linestyle")`.

The following `plot()` command is equivalent to the previous one:

```
plot(x, y, t = :sticks, w = 4, c = "#00b300", m = (:circle, 8, :green ))
```

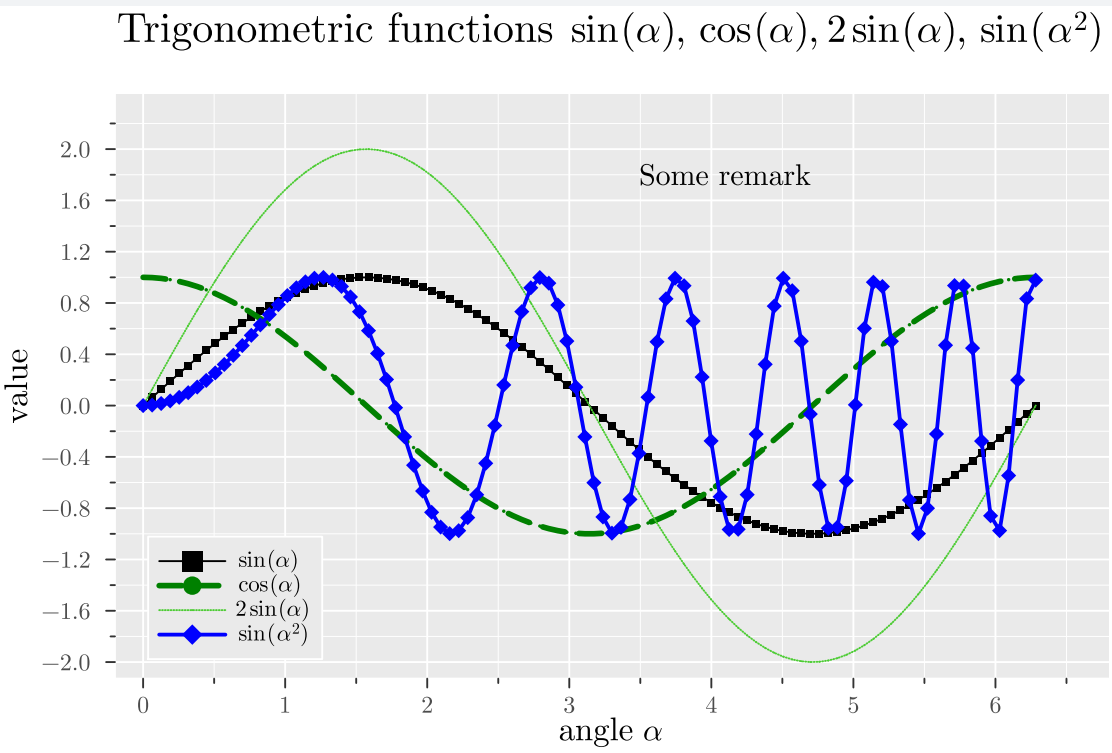
16.2.5 Additional Extras

```
using Plots                # no harm in repeating
using Plots.PlotMeasures   # for measurements in mm, cm,...
using LaTeXStrings         # for LaTeX constructs in plot labels
using PlotThemes           # predefined themes
```

The `LaTeXStrings.jl` package provides the `L"..."` string constructor. These strings can contain LaTeX constructs, especially formulas. Without explicit dollar signs, they are automatically interpreted in LaTeX math mode.

```
xs = range(0, 2π, length = 100)

data = [sin.(xs) cos.(xs) 2sin.(xs) (x->sin(x^2)).(xs)]   # 4 functions

theme(:ggplot2)

plot10 = plot(xs, data,
           fontfamily="Computer Modern",
```

```
            # LaTeX string L"..."
           title = L"Trigonometric functions $\sin(\alpha), \cos(\alpha), 2\sin(\alpha),
\sin(\alpha^2)$",
            xlabel = L"angle $\alpha$",
            ylabel = "value",

            # 1x4-matrices with colors, markers,...  for the 4 'series'
            color=[:black :green  RGB(0.3, 0.8, 0.2) :blue ],
            markers = [:rect :circle :utriangle :diamond],
            markersize = [2 1 0 4],
            linewidth = [1 3 1 2],
            linestyle = [:solid :dash :dot :solid ],

            # axes
            xlim = (0, 6.6),
            ylim = (-2, 2.3),
            yticks = -2:.4:2.3,       # with step size

            # legend
            legend = :bottomleft,
          label = [ L"\sin(\alpha)" L"\cos(\alpha)" L"2\sin(\alpha)"  L"\sin(\alpha^2)"],

            top_margin = 5mm,     # here Plots.PlotMeasures is needed
)

# additional text: annotate!(x-pos, y-pos, text("...", font, fontsize))

annotate!(plot10, 4.1, 1.8, text("Some remark","Computer Modern", 10) )
```

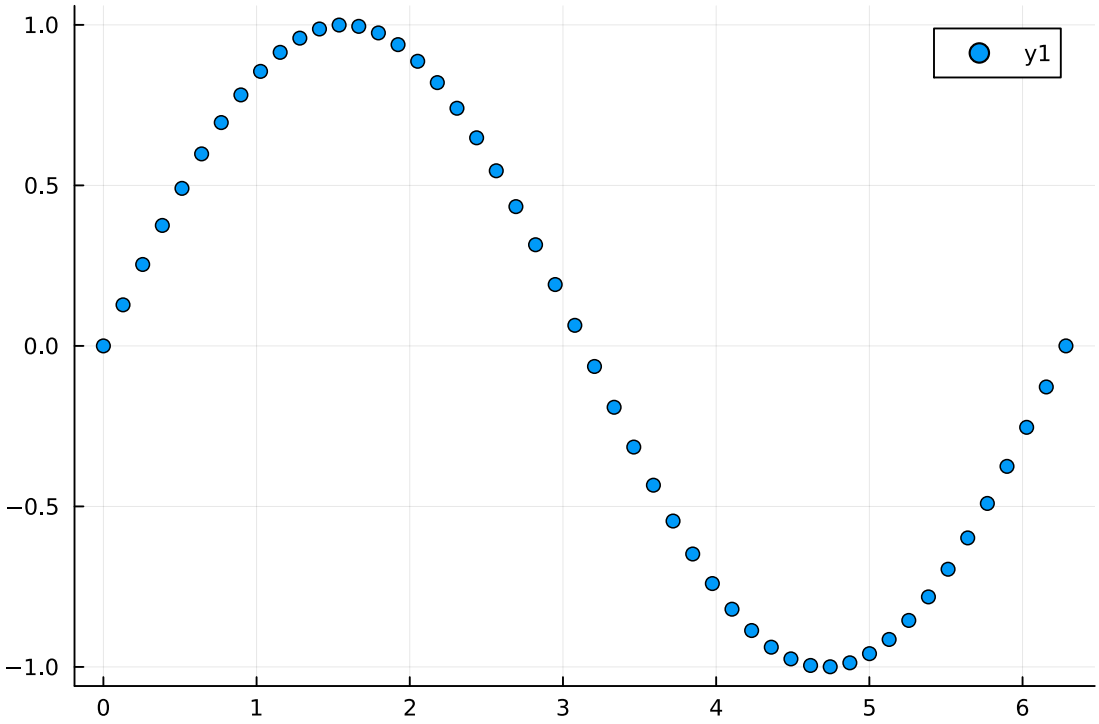


16.2.6 Other Plot Functions

So far, we have mainly plotted lines. Many other types exist, such as *scatter plots, contours, heatmaps, histograms, sticks*, etc.

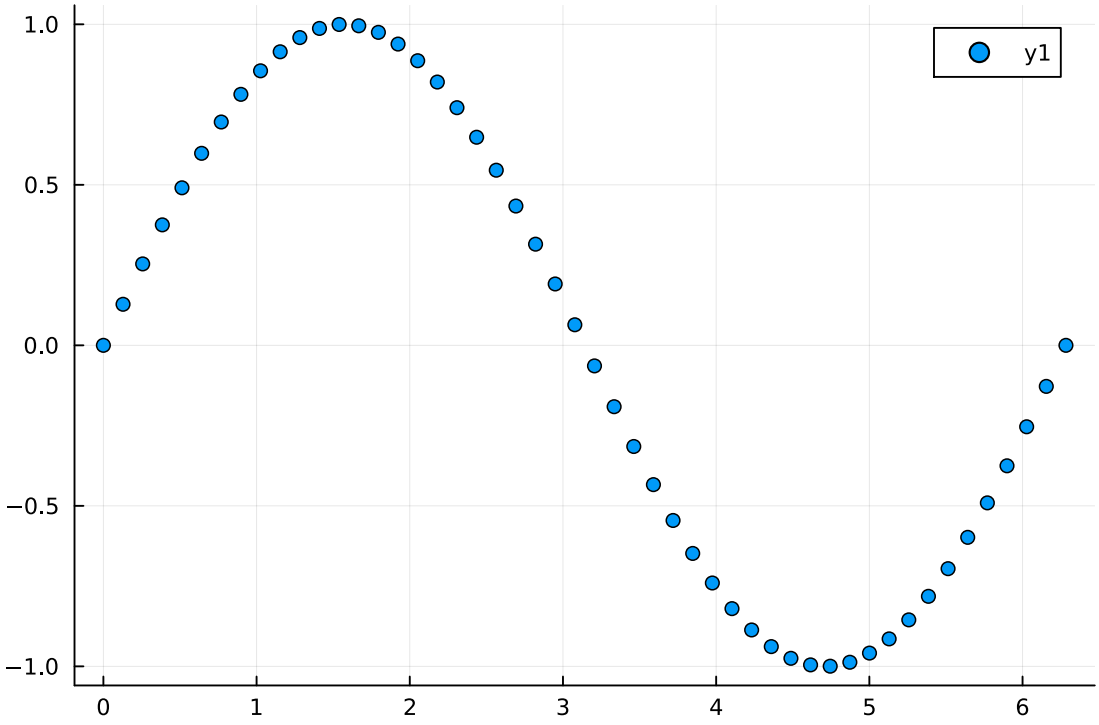
This can be controlled with the `seriestype` attribute:

```
theme(:default)

x = range(0, 2π; length = 50)
plot(x, sin.(x), seriestype=:scatter)
```



or by using the specific function named after the `seriestype`:

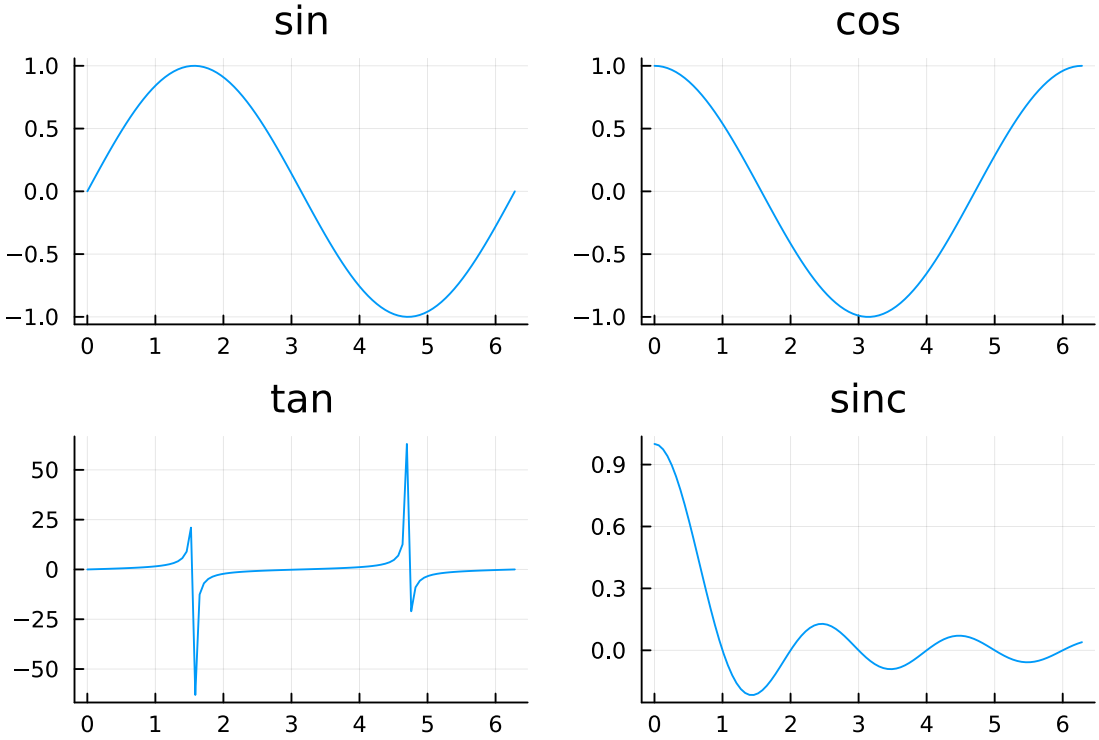
```
x = range(0, 2π; length = 50)
scatter(x, sin.(x))
```



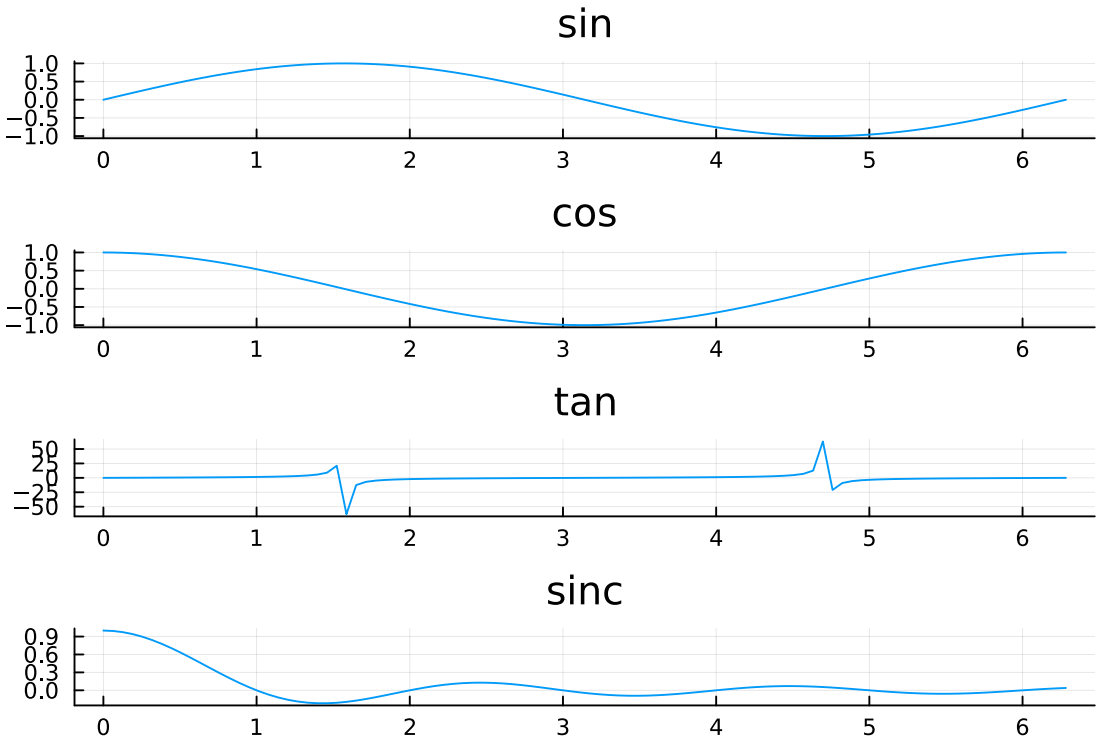
16.2.7 Subplots and Layout

Multiple plots can be combined into one figure. The arrangement is determined by the `layout` parameter: `layout=(m,n)` arranges plots in an $m \times n$ grid:

```
x = range(0, 2π; length = 100)
plots = []                      # vector of plot objects
for f in [sin, cos, tan, sinc]
   p = plot(x, f.(x))
   push!(plots, p)
end
plot(plots..., layout=(2,2), legend=false, title=["sin" "cos" "tan" "sinc"])
```

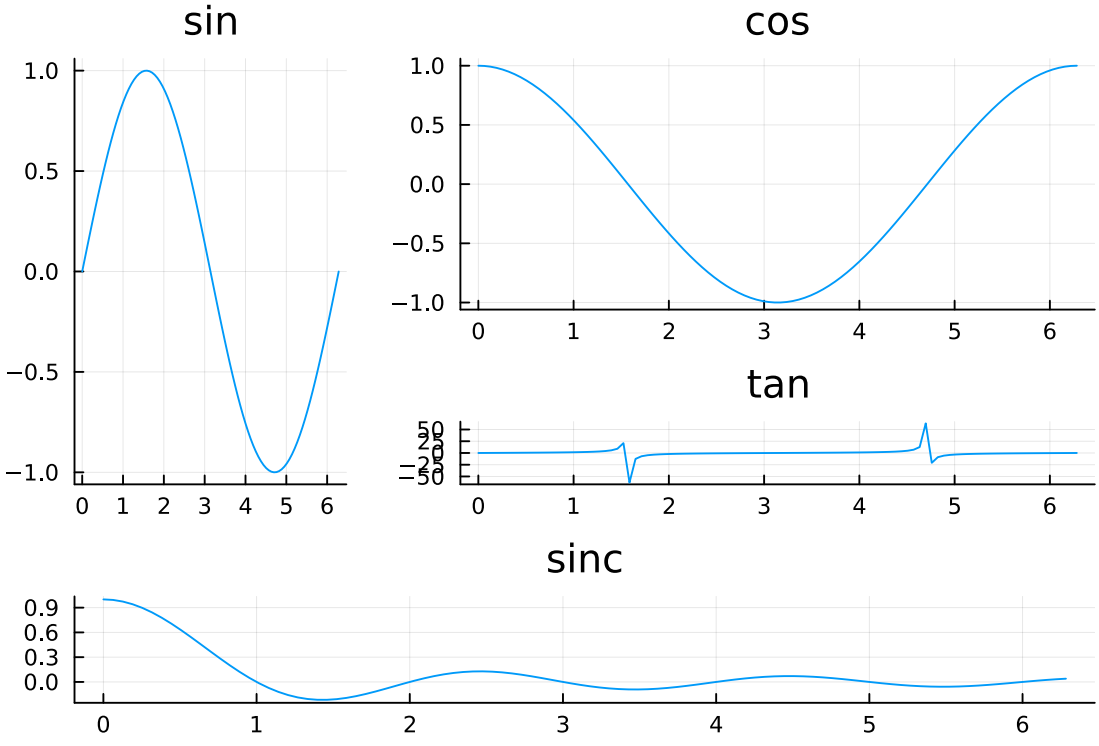



```
plot(plots..., layout=(4,1), legend=false, title=["sin" "cos" "tan" "sinc"])
```



Layouts can also be nested and explicit width/height proportions can be specified using the `@layout` macro:

```
mylayout = @layout [
               a{0.3w} [ b
                         c{0.2h}  ]
               d{0.2h}
           ]

plot(plots..., layout=mylayout, legend=false, title=["sin" "cos" "tan" "sinc"])
```



16.2.8 Backends

`Plots.jl` provides a unified interface to various *backends* (graphics engines). You can switch backends and use the same plot commands and attributes.

However, not all *backends* support all plot types and attributes. An overview is available here.

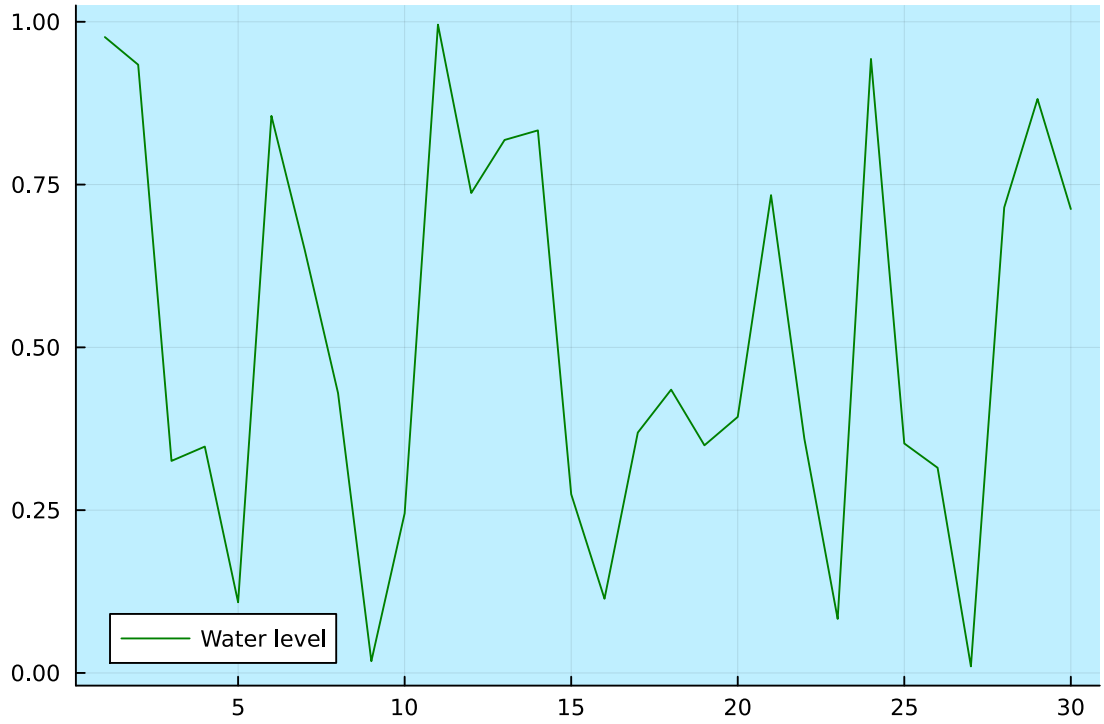
So far, we have used the default backend: GR, a graphics engine developed at the Jülich Research Center, primarily in C.

```
using Plots
backend()      # display the selected backend, GR is the default
```

```
Plots.GRBackend()
```

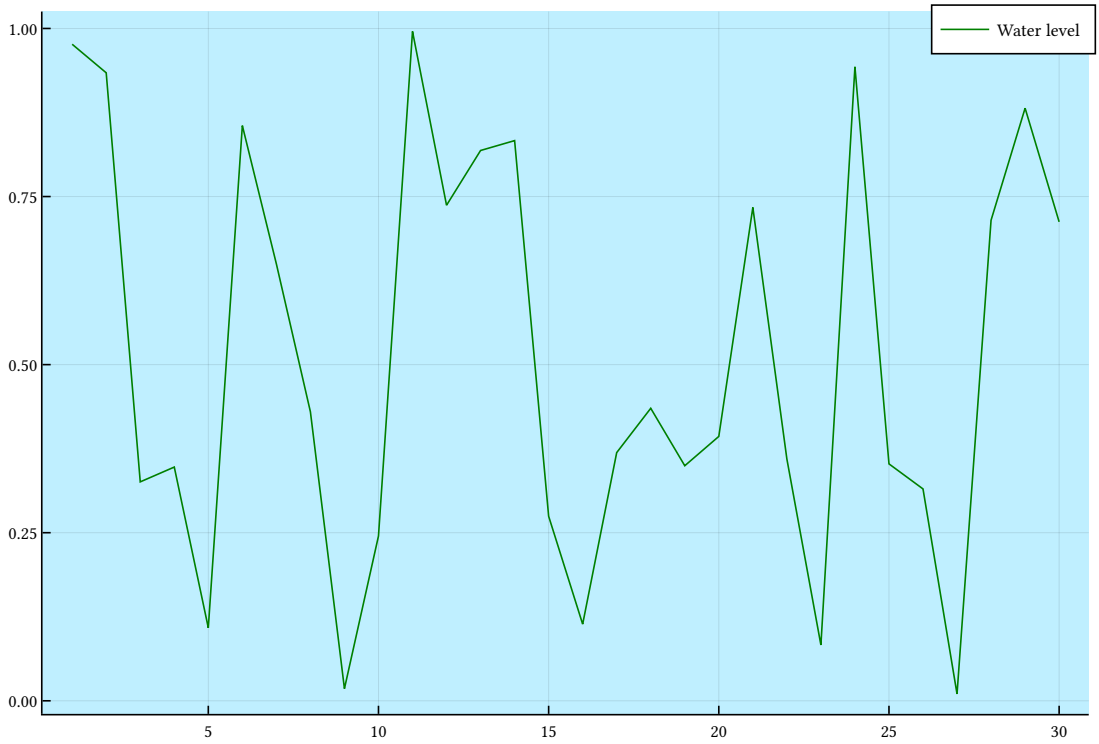
Another example:

```
x = 1:30
y = rand(30)
plot(x, y, linecolor =:green, bg_inside =:lightblue1, line =:solid, label = "Water level")
```



The same plot with the `PlotlyJS` backend:

```
plotlyjs()       # change plots backend
plot(x, y, linecolor =:green, bg_inside =:lightblue1, line =:solid, label = "Water level")
```



This backend enables interactivity via JavaScript. Hovering over the image allows zooming and panning; 3D plots can also be rotated.

```
gr() # return to GR as backend
```

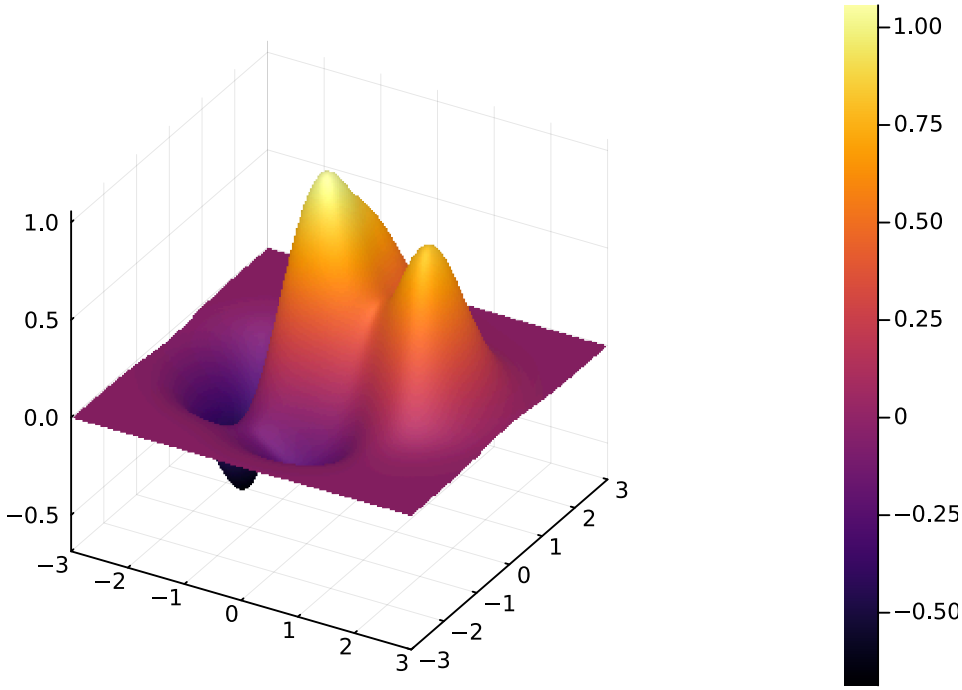
```
Plots.GRBackend()
```

16.2.9 3D Plots

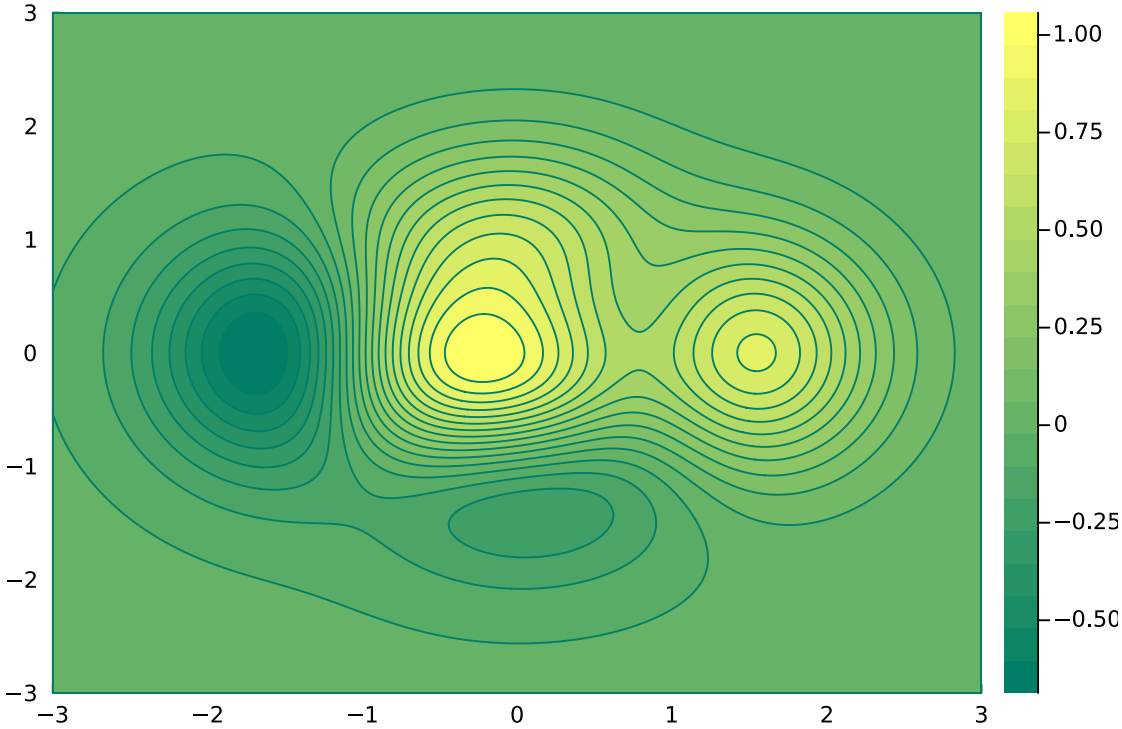
The `surface()` and `contour()` functions plot a function $f(x, y)$. Required arguments are:

- a set (vector) X of x-values,
- a set (vector) Y of y-values and
- a function of two variables that is then evaluated on $X \times Y$ and plotted.

```
f(x,y) = (1 - x/2 + x^5 + y^3) * exp(-x^2 - y^2)

surface( -3:0.02:3, -3:0.02:3,  f)
```



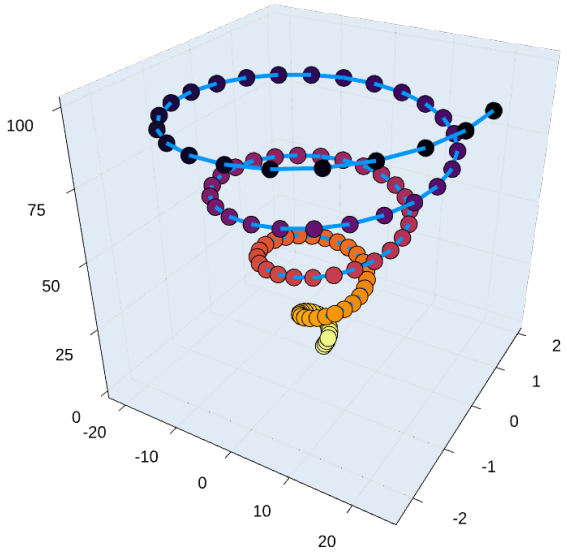

```
contour(  -3:0.02:3,  -3:0.02:3,    f,  fill=true,  colormap=:summer,  levels=20,
contour_labels=false)
```



Curves (or point sets) in three dimensions are plotted by calling `plot()` with three vectors containing the x, y, and z coordinates of the data points.

```
plotlyjs()

t = range(0, stop=8π, length=100)    # parameter t
```

```
x = @. t * cos(t)                    # x(t), y(t), z(t)
y = @. 0.1 * t * sin(t)
z = @. 100 * t/8π
plot(x, y, z, zcolor=reverse(z), markersize=3, markershape= :circle,
             linewidth=5, legend=false, colorbar=false)
```



The `plotlyjs` backend makes the plot interactive: it can be rotated and zoomed with the mouse.

16.2.10 Plots.jl and *recipes*

Other packages can extend `Plots.jl` by defining so-called *recipes* for special plots and data structures (see https://docs.juliaplots.org/latest/ecosystem/), e.g.:

- `StatsPlots.jl`: direct plotting of *DataFrames*, special statistical plots, etc.
- `GraphRecipes.jl`: plotting of graph structures

16.2.11 A Bar Chart

For the last example, we load a package providing over 700 free (*"public domain"*) datasets, including:

- the *Titanic* passenger list,
- fuel consumption data for American cars from the 70s, or
- historical exchange rates:

```
using RDatasets
```

The dataset "Motor Trend Car Road Tests"

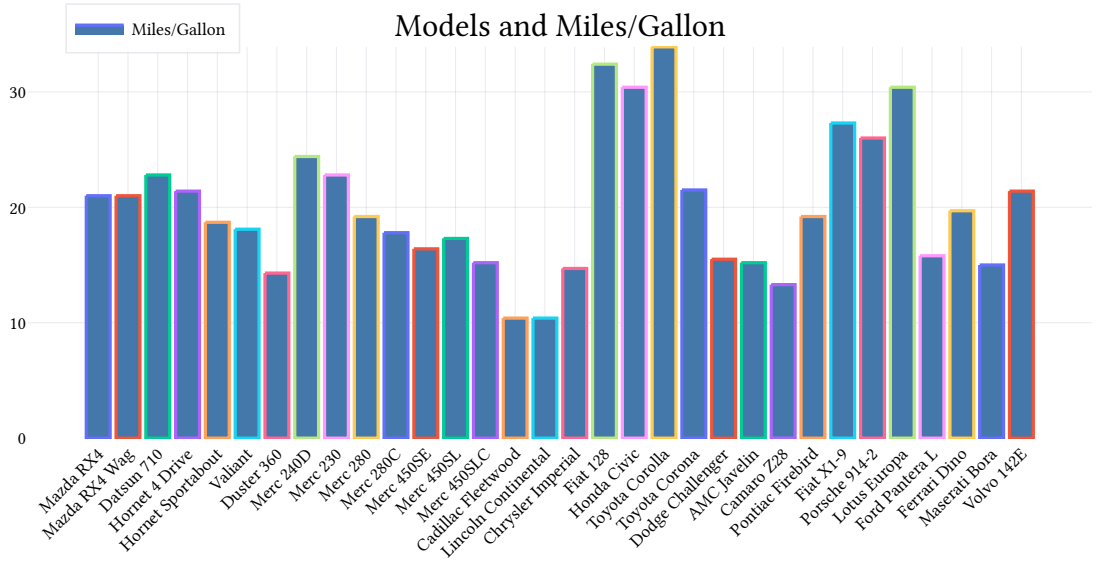
```
cars = dataset("datasets", "mtcars")
```

Row	Model	MPG	Cyl	Disp	HP	DRat	WT	QSec	VS	AM	Gear	Carb
	String31	Float64	Int64	Float64	Int64	Float64	Float64	Float64	Int64	Int64	Int64	Int64
1	Mazda RX4	21.0	6	160.0	110	3.9	2.62	16.46	0	1	4	4
2	Mazda RX4 Wag	21.0	6	160.0	110	3.9	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108.0	93	3.85	2.32	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360.0	175	3.15	3.44	17.02	0	0	3	2
6	Valiant	18.1	6	225.0	105	2.76	3.46	20.22	1	0	3	1
7	Duster 360	14.3	8	360.0	245	3.21	3.57	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.19	20.0	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4
11	Merc 280C	17.8	6	167.6	123	3.92	3.44	18.9	1	0	4	4
12	Merc 450SE	16.4	8	275.8	180	3.07	4.07	17.4	0	0	3	3
13	Merc 450SL	17.3	8	275.8	180	3.07	3.73	17.6	0	0	3	3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
21	Toyota Corona	21.5	4	120.1	97	3.7	2.465	20.01	1	0	3	1
22	Dodge Challenger	15.5	8	318.0	150	2.76	3.52	16.87	0	0	3	2
23	AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.3	0	0	3	2
24	Camaro Z28	13.3	8	350.0	245	3.73	3.84	15.41	0	0	3	4
25	Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
26	Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.9	1	1	4	1

Row	Model	MPG	Cyl	Disp	HP	DRat	WT	QSec	VS	AM	Gear	Carb
	String31	Float64	Int64	Float64	Int64	Float64	Float64	Float64	Int64	Int64	Int64	Int64
27	Porsche 914-2	26.0	4	120.3	91	4.43	2.14	16.7	0	1	5	2
28	Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.9	1	1	5	2
29	Ford Pantera L	15.8	8	351.0	264	4.22	3.17	14.5	0	1	5	4
30	Ferrari Dino	19.7	6	145.0	175	3.62	2.77	15.5	0	1	5	6
31	Maserati Bora	15.0	8	301.0	335	3.54	3.57	14.6	0	1	5	8
32	Volvo 142E	21.4	4	121.0	109	4.11	2.78	18.6	1	1	4	2

We only need two columns for the plot: `cars.Model` and `cars.MPG`, the fuel consumption in *miles per gallon* (higher is more economical).

```
theme(:bright)

bar(cars.Model, cars.MPG,
           label = "Miles/Gallon",
           title = "Models and Miles/Gallon",
           xticks =:all,
           xrotation = 45,
           size = [600, 400],
           legend =:topleft,
           bottom_margin = 10mm
   )
```

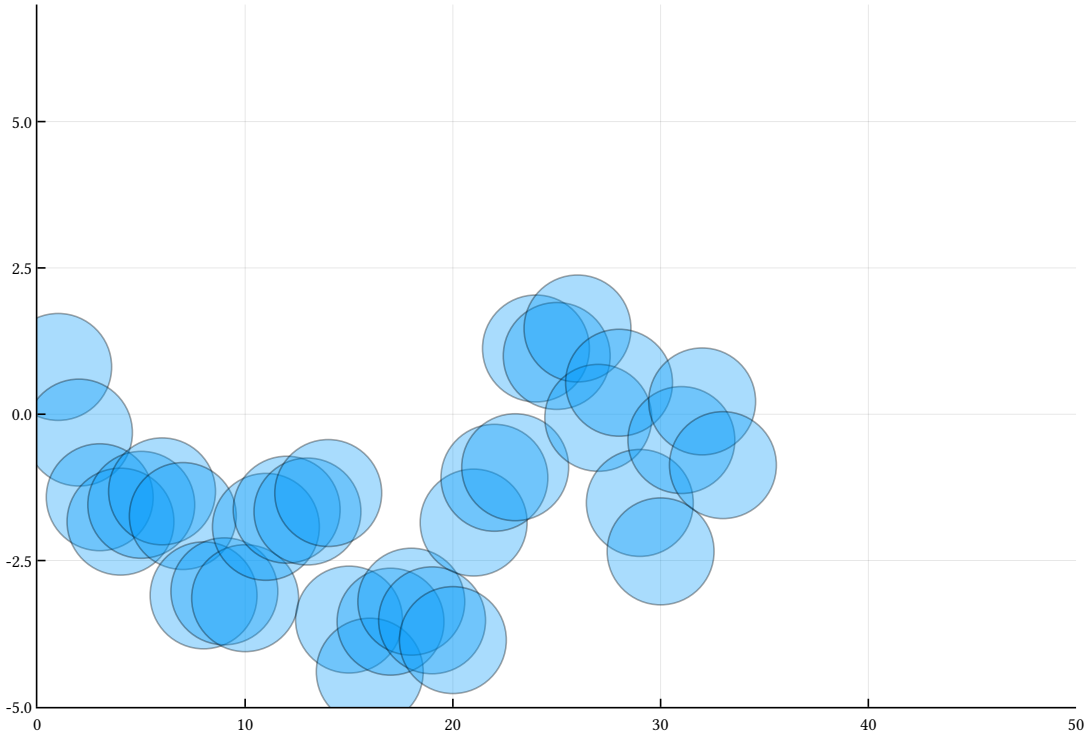


16.2.12 What is Missing: Animation

See the documentation; here is an example (from https://www.juliafordatascience.com/animations-with-plots-jl/):

```
using Plots, Random
theme(:default)

anim = @animate for i in 1:50
    Random.seed!(123)
    scatter(cumsum(randn(i)), ms=i, lab="", alpha = 1 - i/50,
        xlim=(0,50), ylim=(-5, 7))
end

gif(anim, fps=50)
```

```
Plots.AnimatedGif("/tmp/jl_NmBa18fERQ.gif")
```


5.0
2.5
0.0
-2.5
-5.0
0
10
20
30
40
50