

Scientific Computing for Physicist

Yu-Sheng Zhao

Yi-Dai Zhang

Jin-Guo Liu

Yu-Sheng Zhao
Hong-Kong University of Science and Technology (Guangzhou)

Yi-Dai Zhang
Hong-Kong University of Science and Technology (Guangzhou)

Jin-Guo Liu
Hong-Kong University of Science and Technology (Guangzhou)

<https://book.jinguo-group.science>

Version: 2024-02-07

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

Contents

1	Becoming an Open-Source Developer	3
1.1	Get a Terminal!	3
1.2	Maintainability - Version Control	6
1.3	Correctness - Unit Tests	11
2	Julia	13
2.1	Setup Julia	13
2.2	Why Julia is fast?	17
2.3	Type and Multiple-dispatch	27
2.4	Tuple, Array and broadcasting	36
2.5	Publishing a Package	40
	Appendix	43
3	References	45

1 *Becoming an Open-Source Developer*

This section focuses on understanding the open source workflow, which is the foundation of scientific computing. Along the way, we will introduce to you our recommended tools for accomplishing each task.

1.1 *Get a Terminal!*

You need to get a working terminal to follow the instructions in this book, because every cool guy uses a terminal.

1.1.1 *Linux operating system*

Using Linux is the most straight-forward way to get a terminal. Just like Windows, IOS, and macOS, Linux is an operating system. In fact, Android, one of the most popular platforms on the planet, is powered by the Linux operating system. It is free to use, open source¹, widely used on clusters and good at automating your works. Linux kernel and Linux distribution are different concepts. - The **Linux kernel** is started by Linus Torvalds² in 1991. - A **Linux distribution** is an operating system³ made from a software collection that includes the Linux kernel⁴ and, often, a package management system⁵. The Linux distribution used in this course is Ubuntu⁶.

¹ <https://opensource.com/resources/what-open-source>

² https://en.wikipedia.org/wiki/Linus_Torvalds

³ https://en.wikipedia.org/wiki/Operating_system

⁴ https://en.wikipedia.org/wiki/Linux_kernel

⁵ https://en.wikipedia.org/wiki/Package_management_system

⁶ <https://ubuntu.com/desktop>

1.1.2 *Shell (or Terminal)*


Although you can use a **graphical user interface** (GUI) to interact with your Linux distribution, you will find that the **command line interface** (CLI) is more efficient and powerful. The CLI is also known as the **shell** or **terminal**.

The shell is a program that takes commands from the keyboard and gives them to the operating system to perform. Zsh⁷ and Bash⁸ are two popular shell interpreters used in the Linux operating systems. - **Bash** is the default shell on most Linux distributions. - **Zsh** (with oh-my-zsh⁹ extension) is an extended version of the shell, with a more powerful command-line editing and completion system. It includes features like spelling correction and tab-completion, and it also supports plugins and themes.

⁷ <https://zsh.org/>

⁸ <https://gnu.org/software/bash/>

⁹ <https://github.com/ohmyzsh/ohmyzsh>

In Ubuntu, one can use `Ctrl + Alt + T` to open a shell. In a shell, we use `- man`  `command_name` to get help information related to a command, `- CTRL-C` to break a program and `- CTRL-D` to exit a shell or an REPL.

The following is a short list of bash commands that will be used frequently in this book.

```
man      # an interface to the system reference manuals

ls       # list directory contents
cd       # change directory
mkdir    # make directories
rm       # remove files or directories
pwd      # print name of current/working directory

echo     # display a line of text
cat      # concatenate files and print on the standard output

alias    # create an alias for a command

lscpu    # display information about the CPU architecture
lsmem    # list the ranges of available memory with their online status

top      # display Linux processes
ssh      # the OpenSSH remote login client
vim      # Vi IMproved, a programmer's text editor
git      # the stupid content tracker

tar      # an archiving utility
```

Git Resources

- MIT Open course: Missing semester¹⁰
- Get started with the Linux command line and the Shell¹¹

¹⁰ <https://missing.csail.mit.edu/2020/shell-tools/>

¹¹ <https://learn.microsoft.com/en-us/training/paths/shell/>

1.1.3 Vim Editor

To edit files in the terminal, you can use `Vim` - the default text editor in most Linux distributions. `Vim` has three primary modes, each tailored for specific tasks. The primary modes include - **Normal Mode**, where users can navigate through the file and perform tasks like deleting lines or copying text; One can enter the normal mode by typing `ESC`; - **Insert Mode**, where users can insert text as in conventional text editors; One can enter the insert mode by typing `i` in the normal mode; - **Command Mode**, where users input commands for tasks like saving files or searching; One can enter the command mode by typing `:` in the normal mode.

A few commands are listed below to get you started with Vim.

```
i      # input
:w     # write
:q     # quit
:q!    # force quit without saving

u      # undo
CTRL-R # redo
```

All the commands must be executed in the **normal mode**. To learn more about Vim, please check this lecture¹².

¹² <https://missing.csail.mit.edu/2020/editors/>

1.1.4 SSH

The Secure Shell (SSH) protocol is a method for securely sending commands to a computer over an unsecured network. SSH uses cryptography to authenticate and encrypt connections between devices. It is widely used to: - push code to a remote git repository, - log into a remote machine and execute commands.

With a host name (the IP of the target machine to login) and a user name, one can use the following command to login,

```
ssh <username>@<hostname>
```

where <username> is the user's account name and <hostname> is the host name or IP of the target machine. You will get logged in after inputting the password.

Tips to make your life easier

It will be tedious to type the host name and user name everytime you want to login to the remote machine. You can setup the ~/.ssh/config file to make your life easier. The following is an example of the ~/.ssh/config file.

```
Host amat5315
  HostName <hostname>
  User <username>
```

where amat5315 is the alias of the host. After setting up the ~/.ssh/config, you can login to the remote machine by typing

```
ssh amat5315
```

If you want to avoid typing the password everytime you login, you can use the command

```
ssh-keygen
```

to generate a pair of public and private keys, which will be stored in the `~/.ssh` folder on the local machine. After setting up the keys, you can copy the public key to the remote machine by typing

```
ssh-copy-id amat5315
```

Try connecting to the remote machine again, you will find that you don't need to type the password anymore.

How does SSH key pair work? The SSH key pair is a pair of asymmetric keys, one is the public key and the other is the private key. In the above example, the public key is uploaded to the remote machine and the private key is stored on the local machine. The public key can be shared with anyone, but the private key must be kept secret.

To connect to a server, the server needs to know that you are the one who with the right to access it. To do so, the server will need to check if you have the private key that corresponds to the public key stored on the server. If you have the private key, you will be granted access to the server.

The secret of the SSH key pair is that **the public key can be used to encrypt a message that can only be decrypted by the private key**, i.e. the public key is more like a lock and the private key is the key to unlock the lock. This is the foundation of the SSH protocol. So server can send you a message encrypted by your public key, and only you can decrypt it with your private key. This is how the server knows that you are the one who has the private key without actually sending the private key to the server.

1.2 Maintainability - Version Control

Maintaining a software project is not easy, especially when it comes to multiple developers working on the same piece of code. When adding a new feature to the project, maintainers may encounter the following problems:

- Multiple developers modify the same file at the same time, works can not be merged easily.
- New code breaks an existing feature, downstream users are affected.

The solution to the above problems is **version-control**. Among all version control software, **git** is the most popular one.

1.2.1 Create a git repository

A git repository, also known as a repo, is basically a directory where your project lives and git keeps track of your file's history. To get started, you start with a terminal and type

```
cd path/to/working/directory
git init
echo "Hello, World" > README.md
git add -A
git commit -m 'this is my initial commit'
git status
```

- Line 1: changes the directory to the working directory, which can be either an existing directory or a new directory.
- Line 2: initializes a git repository in the working directory. A `.git` directory is created in the working directory, which contains all the necessary metadata for the repo.
- Line 3: creates a file `README.md` with the content `Hello, World`. The file `README.md` is a **markdown** file, which is a lightweight markup language with plain-text-formatting syntax. You can learn more about markdown from the markdown tutorial¹³. This line can be omitted if the working directory already contains files.
- Line 4: line add files to the **staging area** (area that caches changes that to be committed).
- Line 5: commits the changes to the repository, which will create a **snapshot** of your current work.
- Line 6: shows the status of the working directory, staging area, and repository. If the above commands are executed correctly, the output should be nothing to commit, working tree clean.

¹³ <https://www.markdowntutorial.com/>

1.2.2 Track the changes

Git enables developers to track changes in their codebase. Continuing the previous example, we can analyze the repository with the following commands:

```
echo "Bye Bye, World" > README.md
git diff
git add -A
git commit -m 'a second commit'
git log
git checkout HEAD~1
git checkout main
```

- Line 1: makes changes to the file `README.md`.
- Line 2: shows the changes made to the file `README.md`.
- Line 3-4: adds the changes to the staging area and commits the changes to the repository.
- Line 5: shows the history of commits. The output should be something like this:

```
commit 02cd535b6d78fca1713784c61eec86e67ce9010c (HEAD -> main)
Author: GiggLeLiu <cacate0129@gmail.com>
Date: Mon Feb 5 14:34:20 2024 +0800
```

```
    a second commit
```

```
commit 570e390759617a7021b0e069a3fbe612841b3e50
Author: GiggLeLiu <cacate0129@gmail.com>
Date: Mon Feb 5 14:23:41 2024 +0800
```

```
    this is my initial commit
```

- Line 6: Checkout the previous snapshot. Note `HEAD` is your current snapshot and `HEAD~n` is the n th snapshot counting back from the current snapshot.
- Line 7: Return to the `main` **branch**, which points to the latest snapshot. We will discuss more about **branch** later in this tutorial.

You can use `git reset` to reset the current `HEAD` to the specified snapshot, which can be useful when you committed something bad by accident.

1.2.3 Work with remote repositories

A server to store git repository, or **remote** in git terminology, is required for the collaboration purpose. Remote repositories can be hosted on git hosting services like GitHub, GitLab, or Bitbucket. After creating a new empty repository (no `README` files) on a git hosting service (How to create a new github repo?¹⁴), a URL for cloning the repo will show up, which that usually starts with `git` or `https`. Let us denote this URL as `<url>` and continue the previous example:

```
git remote add origin <url>
git remote -v
git push origin main
```

¹⁴ <https://docs.github.com/en/get-started/quickstart/create-a-repo>

- Line 1: add a remote repository, where `origin` is a tag for the added remote.

- Line 2: shows the URL of all remotes, including the `origin` remote we just added.
- Line 3: push commits to the `main` branch of the remote repository `origin`. This command sometimes could fail due to another commit pushed to the remote earlier, where the commit may from another machine or another person. To resolve the issue, you can use `git pull origin main` to fetch the latest snapshot on the remote. `git pull` may also fail, because the remote commit may be incompatible with the local commit, e.g. the same file has been changed. In this worst case, you need to merge two commits manually (link).

1.2.4 Develop features safely - branches

So far, we worked with a single branch `main`. A **branch** in git is a lightweight pointer to a specific commit. Working on a single branch is dangerous due to the following reasons: - *No usable code*. Developers usually develop features based on the current `main` branch, so the `main` branch is expected to always usable. However, working on a single branch can easily break this rule. - *Hard to resolve conflicts*. when multiple developers modify the same file at the same time, works can not be merged easily. Multiple branches can make the feature development process independent of each other, which can avoid conflicts. - *Hard to discard a feature*. For some experimental features, you may want to discard it after testing. A commit on the `main` branch can not be easily reverted.

Understanding the branches is extremely useful when, multiple developers are working on different features.

```
git checkout -b me/feature
echo "Hello, World - Version 2" > README.md
git add -A
git commit -m 'this is my feature'
git push origin me/feature
```

- Line 1: create and switch to the new branch `me/feature`. Here, we use the branch name `me/feature` to indicate that this branch is for the feature developed by `me`, which is a matter of convention.
- Line 2-5: makes some changes to the file `README.md` and commits the changes to the repository. Finally, the changes are pushed to the remote repository `origin`. The remote branch `me/feature` is created automatically.

While developing a feature, you or another developer may want to develop another feature based on the current `main` branch. You can create another branch `other/feature` and develop the feature there.

```
git checkout main
git checkout -b other/feature
echo "Bye Bye, World - Version 2" > feature.md
git add -A
git commit -m 'this is another feature'
git push origin other/feature
```

In the above example, we created a new branch `other/feature` based on the `main` branch, and made some changes to the file `feature.md`.

Finally, when the feature is ready, you can merge the feature branch to the main branch.

```
git checkout main
git merge me/feature
git push origin main
```

1.2.5 Working with others - issues and pull requests

When working with others, you may want to propose changes to a repository and discuss them with others. This is where **issues** and **pull requests** come in. Issues and pull requests are features of git hosting services like GitHub and GitLab. - **Issue** is relatively simple, it is a way to report a bug or request a feature. - **Pull request** (resource: how to create a pull request¹⁵) is a way to propose changes to a repository and discuss them with others. It is also a way to merge code from source branch to target branch. The source branch can be a branch in the same repository or a branch in a **forked repository** - a copy of the repository in your account. Forking a repository is needed when you want to propose changes to a repository that you do not have write access to.

¹⁵ <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request>

FAQ: Should I make a pull requests or push directly to main branch?

To update the main branch, one should use pull requests as much as possible, even if you have write access to the repository. It is a good practice to discuss the changes with others before merging them to the main branch. A pull request also makes the changes more traceable, which is useful when you want to revert the changes.

1.2.6 Summary: a cheat sheet

It is not possible to cover all the feature of git. We will list a few useful commands and resources for git learning.

<https://github.com/johndoe/Book.jl>

```

# global config
git config # Get and set repository or global options

# initialize a repo
git init # Create an empty Git repo or reinitialize an existing one
git clone # Clone repository into new directory

# info
git status # Show the working tree status
git log # Show commit logs
git diff # Show changes between commits, commit and working tree, etc

# work on a branch
git add # Add file contents to the index
git rm # Remove files from the working tree and from the index
git commit # Record changes to the repository
git reset # Reset current HEAD to the specified state

# branch manipulation
git checkout # Switch branches or restore working tree files
git branch # List, create, or delete branches
git merge # Join two or more development histories together

# remote synchronization
git remote # Manage set of tracked repositories
git pull # Fetch from and integrate with another repo or a local branch
git fetch # Download objects and refs from another repository
git push # Update remote refs along with associated objects

```

1.2.7 Resources

- The Official GitHub Training Manual¹⁶
- MIT online course missing semester¹⁷.

¹⁶ <https://githubtraining.github.io/training-manual/book.pdf>

¹⁷ <https://missing.csail.mit.edu/2020/>

1.3 Correctness - Unit Tests

In terms of scientific computing, accuracy of your result is most certainly more important than anything else. To ensure the correctness of the code, we employ two methods: **Unit Testing** and **CI/CD**.

1.3.1 Unit Test

Unit tests are typically automated tests¹⁸ written and run by software devel-

¹⁸ https://en.wikipedia.org/wiki/Automated_test

¹⁹ https://en.wikipedia.org/wiki/Software_developer

operators¹⁹ to ensure that a section of an application (known as the “unit”) meets its design²⁰ and behaves as intended. Unit tests are composed of a series of individual test cases, each of which verifies the correctness by using **assertions**. If all assertions are true, the test case passes; otherwise, it fails. The unit tests are run automatically whenever the code is changed, ensuring that the code is always in a working state. In Julia, there exists a helpful module called Test²¹ to help you do unit testing.

²⁰ https://en.wikipedia.org/wiki/Software_design

²¹ <https://docs.julialang.org/en/v1/stdlib/Test/>

1.3.2 Automate your workflow - CI/CD

CI/CD, which stands for continuous integration and continuous delivery/deployment, aims to streamline and accelerate the software development lifecycle. CI/CD are often integrated with git hosting services, e.g. Github Actions²². Typical CI/CD pipelines include the following steps:

²² <https://docs.github.com/en/actions>

- Automatically **build**, **test** and **merge** the code changes whenever a developer commits code to the repository.
- Automatically **deploy** the code or documentation to a cloud service.

The CI/CD pipeline is a powerful tool to ensure the correctness of the code and the reproducibility of the results. It is also a good practice to use CI/CD to automate the workflow, especially when you are working with a team.

2 Julia

Julia is a high-level, high-performance, dynamic programming language. From the designing stage, Julia is intended to address the needs of high-performance numerical analysis and computational science, without the typical need of separate compilation to be fast, while also being effective for general-purpose programming, web use or as a specification language. Julia is also a free and open-source language, with a large community¹ and a rich ecosystem².

¹ <https://julialang.org/community/>

² <https://juliahub.com/>

We will delve deeper into Julia later in the chapter. For now, we will just install Julia and setup the environment.

2.1 Setup Julia

2.1.1 Step 1: Installing Julia

For Linux/Mac users, please open a terminal and type the following command to install Julia³ with Juliaup⁴. Juliaup is a tool to manage Julia versions and installations. It allows you to install multiple versions of Julia and switch between them easily.

³ <https://julialang.org/>

⁴ <https://github.com/JuliaLang/juliaup>

```
curl -fsSL https://install.julialang.org | sh # Linux and macOS
```

For Windows users, please open and execute the following command in a cmd,

```
winget install julia -s msstore # Windows
```

You can also install Juliaup directly from Windows Store⁵.

⁵ <https://www.microsoft.com/store/apps/9NJNWW8PVKMN>

2.1.2 For users suffering from the slow download speed

Network connectivity can be an issue for some users, especially for those who are in China. You may need to specify another server for installing Juliaup and Julia packages. To do so, execute the following command in your terminal before running the script above.

Linux and macOS

```
export JULIAUP_SERVER=https://mirror.nju.edu.cn/julia-releases/ # Linux & macOS
export JULIA_PKG_SERVER=https://mirrors.nju.edu.cn/julia
```

Windows

```
$env:JULIAUP_SERVER="https://mirror.nju.edu.cn/julia-releases/" # Windows
$env:JULIA_PKG_SERVER="https://mirrors.nju.edu.cn/julia"
```

An alternative approach is downloading the corresponding Julia binary from the Nanjing university mirror website⁶. After installing the binary, please set the Julia binary path properly if you want to start a Julia REPL from a terminal, check this manual page⁷ to learn more.

⁶ <https://mirror.nju.edu.cn/julia-releases/>

⁷ <https://juliaLang.org/downloads/platform/>

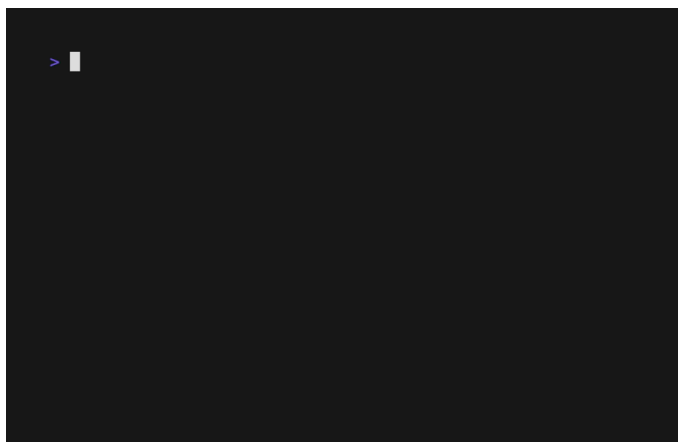
2.1.3 Installing Julia

To verify that Julia is installed, please open a **new** terminal and run the following command in your terminal. `bash julia` - It should start a Julia REPL (Read-Eval-Print-Loop) session like this - If you wish to install a specific version of Julia, please refer to the documentation⁸.

⁸ <https://github.com/JuliaLang/juliaup>

2.1.4 Step 2: Package Management

- Julia has a mature eco-system for scientific computing.
- `Pkg` is the built-in package manager for Julia.



- To enter the package manager, press `]` in the REPL.
- The environment is indicated by the `(@v1.9)`.
- To add a package, type `add <package name>`.
- To exit the package manager press `backspace` key
- Read More⁹

⁹ <https://pkgdocs.julialang.org/v1/managing-packages/>

2.1.5 Step 3. Configure the startup file

First create a new file `~/.julia/config/startup.jl` by executing the following commands

```
mkdir -r ~/.julia/config touch ~/.julia/config/startup.jl
```

You could open the file with your favourite editor and add the following content

```
try
    using Revise
catch e
    @warn "fail to load Revise."
end
```

The contents in the startup file is executed immediately after you open a new Julia session.

Then you need to install `Revise`¹⁰, which is an Julia package that can greatly improve the using experience of Julia. To install `Revise`, open Julia REPL and type

¹⁰ <https://github.com/timholy/Revise.jl>

```
julia> using Pkg; Pkg.add("Revise")
```

If you don't know about `startup.jl` and where to find it, here¹¹ is a good place for further information.

¹¹ <https://docs.julialang.org/en/v1/manual/command-line-interface/#Startup-file>

2.1.6 More Packages

- You may find more Julia packages here¹².

¹² <https://juliahub.com/>

As a final step, please verify your Julia configuration by opening a Julia REPL and type

```
julia> versioninfo()
Julia Version 1.9.2
Commit e4ee485e909 (2023-07-05 09:39 UTC)
Platform Info:
  OS: macOS (arm64-apple-darwin22.4.0)
  CPU: 10 × Apple M2 Pro
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-14.0.6 (ORCJIT, apple-m1)
  Threads: 1 on 6 virtual cores
Environment:
```

```
JULIA_NUM_THREADS = 1
JULIA_PROJECT = @.
JULIA_PKG_SERVER = http://cn-southeast.pkg.juliacn.com/
```

2.1.7 Step 4. Download an editor: VSCode

Install VSCode by downloading the correct binary for your platform from here¹³.¹³ <https://code.visualstudio.com/download>
 Open VSCode and open the `Extensions` tab on the left side-bar of the window, search `Julia` and install the most popular extension. read more...¹⁴

¹⁴ <https://github.com/julia-vscode/julia-vscode>

You are ready to go, cheers!

2.1.8 The four modes of Julia REPL

A Julia REPL has four modes,

1. **Julian mode** is the default mode that can interpret your Julia code.
2. **Shell mode** is the mode that you can run shell commands. Press `;` in the Julian mode and type

```
shell> date
Sun Nov 6 10:50:21 PM CST 2022
```

To return to the Julian mode, type the Backspace key.

3. **Package mode** is the mode that you can manage packages. Press `]` in the Julian mode and type

```
(@v1.8) pkg> st
Status `~/julia/environments/v1.8/Project.toml`
 [295af30f] Revise v3.4.0
```

To return to the Julian mode, type the Backspace key.

4. **Help mode** is the mode that you can access the docstrings of functions. Press `?` in the Julian mode and type

```
help> sum
... docstring for sum ...
```

To return to the Julian mode, type the Backspace key.

read more...¹⁵

¹⁵ <https://docs.julialang.org/en/v1/stdlib/REPL/>

2.2 Why Julia is fast?

2.2.1 What is Julia programming language?

Julia is a modern, open-source, high performance programming language for technical computing. It was born in 2012 in MIT, now is maintained by JuliaHub Inc. located in Boston, US.

Julia is open-source. Julia source code is maintained on GitHub repo JuliaLang/julia¹⁶, and its open-source LICENSE is MIT. Julia packages can be found on JuliaHub¹⁷, most of them are open-source.

¹⁶ <https://github.com/JuliaLang/julia>

¹⁷ <https://juliahub.com/ui/Packages>

Julia is designed for high performance (arXiv:1209.5145¹⁸). It is a dynamic programming language, but it is as fast as C/C++. The following figure shows the computing time of multiple programming languages normalized to C/C++.

¹⁸ <https://arxiv.org/abs/1209.5145>

Julia is a trend in scientific computing. Many famous scientists and engineers have switched to Julia from other programming languages.

- **Steven G. Johnson**, creator of FFTW¹⁹, switched from C++ to Julia years ago.
- **Anders Sandvik**, creator of Stochastic Series Expansion (SSE) quantum Monte Carlo method, switched from Fortran to Julia recently.
 - Course link: Computational Physics²⁰
- **Miles Stoudenmire**, creator of ITensor²¹, switched from C++ to Julia years ago.
- **Jutho Haegeman**, **Chris Rackauckas** and more.

¹⁹ <http://www.fftw.org/>

²⁰ <https://physics.bu.edu/~py502/>

²¹ <https://itensor.org/>

FAQ: Should I switch to Julia?

Before switching to Julia, please make sure:

- the problem you are trying to solve runs more than 10min.
- you are not satisfied by any existing tools.

2.2.2 My first program: Factorial

Before we start, please make sure you have the needed packages installed. Type] in the Julia REPL to enter the package manager, and then type

```
pkg> add BenchmarkTools, MethodAnalysis
```

Go back to the REPL by pressing Backspace.

```
julia> function jlfactorial(n)
    x = 1
    for i in 1:n
        x = x * i
    end
    return x
end
jlfactorial (generic function with 1 method)
```

To make sure the performance is measured correctly, we use the `@btime` macro in the `BenchmarkTools` package to measure the performance of the function.

```
julia> @btime jlfactorial(x) setup=(x=5)
2.208 ns (0 allocations: 0 bytes)
120
```

CPU clock cycle is $\sim 0.3\text{ns}$, so it takes only a few clock cycles to compute the factorial of 5. Julia is really fast!

2.2.3 Compare with the speed of C program

To measure the performance of the C program, we can utilize the benchmark utilities in Julia. Benchmarking C program with Julia is accurate because Julia has perfect interoperability with C, which allows zero-cost calling of C functions.

In the following example, we first write a C program to calculate the factorial of a number. The file is named `demo.c`, and the content is as follows:

```
$ cat demo.c
#include <stddef.h>
int c_factorial(size_t n) {
    int s = 1;
    for (size_t i=1; i<=n; i++) {
        s *= i;
    }
    return s;
}
```

To execute a C program in Julia, one needs to compile it to a shared library.

```
$ gcc demo.c -fPIC -O3 -shared -o demo.so
```

To call the function in Julia, one can use the `@ccall` macro in the `Libdl` package (learn more²²). Please open a Julia REPL and execute the following code:

²² <https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/>


```
julia> using Libdl

julia> c_factorial(x) = Libdl.@ccall "./demo.so".c_factorial(x::Csize_t)::Int
```

The benchmark result is as follows:

```
julia> using BenchmarkTools

julia> @benchmark c_factorial(5)
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
Range (min ... max):  7.333 ns ... 47.375 ns | GC (min ... max): 0.00% ... 0.00%
Time  (median):       7.458 ns                | GC (median):    0.00%
Time  (mean ± σ):     7.764 ns ± 1.620 ns    | GC (mean ± σ): 0.00% ± 0.00%


7.33 ns      Histogram: log(frequency) by time      12.6 ns <

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

Although the C program requires the type of variables to be manually declared, its performance is very good. The computing time is only 7.33 ns.

2.2.4 Compare with the speed of Python program

We use the `timeit` module in `ipython` to measure the performance of the Python program.

```
In [5]: def factorial(n):
...:     x = 1
...:     for i in range(1, n+1):
...:         x = x * i
...:     return x
...:

In [6]: factorial(5)
Out[6]: 120

In [7]: timeit factorial(5)
144 ns ± 0.379 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)

In [8]: factorial(100)
Out[8]:
↪ 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272375224518521936582601657498051845928684221875
```

One can also use the `PyCall` package to call the Python function in Julia.

The computing time of the Python program is 144 ns, which is 20 times slower than the C program and 70 times slower than the Julia program. On the other hand, the python program is more flexible since its integer type is not limited by the machine word size.

```
julia> typemax{Int}
9223372036854775807

julia> jlfactorial(100)
0
```

The reason why python is slow and flexible are the same. In python the type of a variable is not declared when it is defined, and it can be changed at any time. This is why the integer type becomes an arbitrary precision integer type when the number is too large. If a variable does not have a fixed type, the program can not preallocate memory for it due to the lack of size information. Then a dynamic typed language has to use a tuple `(type, *data)` to represent an object, where `type` is the type of the object and `*data` is the pointer to the data. Pointing to a random memory location is slow, because it violates the principle of data locality. Lacking of data locality causes the frequent cache miss - failure to find the data in the L1, L2, or L3 cache. Loading data from the main memory is slow, because of the long latency of reading the main memory.

2.2.5 Combining Python and C/C++?

From the maintainer's perspective, it is hard to maintain a program written in both Python and C/C++: - It makes the build configuration files complicated.
- Learning two programming languages is hard for new contributors.

Using python as glue is not as powerful as it looks, the following problem can not be solved by this approach: - Monte Carlo simulation. - Branching and bound algorithms.

2.2.6 Julia's solution: Just in time (JIT) compilation

Given that you have defined a Julia function, the Julia compiler will generate a binary for the function when it is called for the first time. The binary is called a **method instance**, and it is generated based on the **input types** of the function. The method instance is then stored in the method table, and it will be called when the function is called with the same input types. The method instance is generated by the LLVM compiler, and it is optimized for the input types. The method instance is a binary, and it is as fast as a C/C++ program.

Step 1: Infer the types

```
julia> @code_warntype jlfactorial(10)
MethodInstance for jlfactorial(::Int64)
  from jlfactorial(n) @ Main REPL[4]:1
Arguments
  #self#::Core.Const(jlfactorial)
  n::Int64
Locals
  @_3::Union{Nothing, Tuple{Int64, Int64}}
  x::Int64
  i::Int64
Body::Int64
1 —      (x = 1)
|      %2 = (1:n)::Core.PartialStruct{UnitRange{Int64}, Any[Core.Const(1), Int64]}
|      (@_3 = Base.iterate(%2))
|      %4 = (@_3 === nothing)::Bool
|      %5 = Base.not_int(%4)::Bool
—      goto #4 if not %5
2 ... %7 = @_3::Tuple{Int64, Int64}
|      (i = Core.getfield(%7, 1))
|      %9 = Core.getfield(%7, 2)::Int64
|      (x = x * i)
|      (@_3 = Base.iterate(%2, %9))
|      %12 = (@_3 === nothing)::Bool
|      %13 = Base.not_int(%12)::Bool
—      goto #4 if not %13
3 —      goto #2
4 ...      return x
```

When type inference fails

```
julia> badcode(x) = x > 3 ? 1.0 : 3

julia> @code_warntype badcode(4)
MethodInstance for badcode(::Int64)
  from badcode(x) @ Main REPL[9]:1
Arguments
  #self#::Core.Const(badcode)
  x::Int64
Body::Union{Float64, Int64}
1 — %1 = (x > 3)::Bool
—      goto #3 if not %1
2 —      return 1.0
3 —      return 3
```

`Union{Float64, Int64}` means the return type is either `Float64` or `Int64`.

Type unstable code is slow

```
julia> x = rand(1:10, 1000);

julia> @benchmark badcode.($x)
BenchmarkTools.Trial: 10000 samples with 8 evaluations.
Range (min ... max):  2.927 μs ... 195.198 μs | GC (min ... max):  0.00% ... 96.52%
Time  (median):       3.698 μs                | GC (median):           0.00%
Time  (mean ± σ):     4.257 μs ±  7.894 μs    | GC (mean ± σ):  12.43% ±  6.54%
```



Memory estimate: 26.72 KiB, allocs estimate: 696.

```
julia> stable(x) = x > 3 ? 1.0 : 3.0
stable (generic function with 1 method)
```

```
julia> @benchmark stable.($x)
BenchmarkTools.Trial: 10000 samples with 334 evaluations.
Range (min ... max):  213.820 ns ... 25.350 μs | GC (min ... max):  0.00% ... 98.02%
Time  (median):       662.551 ns                | GC (median):           0.00%
Time  (mean ± σ):     947.978 ns ±  1.187 μs    | GC (mean ± σ):  29.30% ± 21.05%
```



Memory estimate: 7.94 KiB, allocs estimate: 1.

- “.” is the broadcasting operator.
- “\$” is the interpolation operator, it is used to interpolate a variable into an expression.

Step 2: Generates the LLVM intermediate representation

LLVM is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is the backend of multiple languages, including Julia, Rust, Swift and Kotlin.

```
julia> @code_llvm jlfactorial(10)
```

or any instruction set architecture. LLVM is the backend of multiple languages,
 ↳ including Julia, Rust, Swift and Kotlin.


```

; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
  ↳#===#d2429055-58e9-4d84-894f-2e639723e078:1 within `jlfactorial`
define i64 @julia_jlfactorial_3677(i64 signext %0) #0 {
top:
; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
  ↳#===#d2429055-58e9-4d84-894f-2e639723e078:3 within `jlfactorial`
;  | @ range.jl:5 within `Colon`
;  | | @ range.jl:403 within `UnitRange`
;  | | | @ range.jl:414 within `unitrange_last`
    %1 = call i64 @llvm.smax.i64(i64 %0, i64 0)
; LLL
;  | @ range.jl:897 within `iterate`
;  | | @ range.jl:672 within `isempty`
;  | | | @ operators.jl:378 within `>`
;  | | | | @ int.jl:83 within `<`
    %2 = icmp slt i64 %0, 1
; LLLL
    br i1 %2, label %L32, label %L17.preheader

L17.preheader:
                                ; preds = %top
; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
  ↳#===#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`
    %min.iters.check = icmp ult i64 %1, 2
    br i1 %min.iters.check, label %scalar.ph, label %vector.ph

vector.ph:
                                ; preds = %L17.preheader
    %n.vec = and i64 %1, 9223372036854775806
    %ind.end = or i64 %1, 1
    br label %vector.body

vector.body:
                                ; preds = %vector.body, %
  ↳vector.ph
    %index = phi i64 [ 0, %vector.ph ], [ %induction12, %vector.body ]
    %vec.phi = phi i64 [ 1, %vector.ph ], [ %3, %vector.body ]
    %vec.phi11 = phi i64 [ 1, %vector.ph ], [ %4, %vector.body ]
    %offset.idx = or i64 %index, 1
    %induction12 = add i64 %index, 2
; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
  ↳#===#d2429055-58e9-4d84-894f-2e639723e078:4 within `jlfactorial`
;  | @ int.jl:88 within `*`
    %3 = mul i64 %vec.phi, %offset.idx
    %4 = mul i64 %vec.phi11, %induction12
    %5 = icmp eq i64 %induction12, %n.vec
    br i1 %5, label %middle.block, label %vector.body

middle.block:
                                ; preds = %vector.body
; L
; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
  ↳#===#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`
    %bin.rdx = mul i64 %4, %3

```

```

%cmp.n = icmp eq i64 %1, %n.vec
br i1 %cmp.n, label %L32, label %scalar.ph

scalar.ph:
    ; preds = %middle.block, %L17.
    →preheader
    %bc.resume.val = phi i64 [ %ind.end, %middle.block ], [ 1, %L17.preheader ]
    %bc.merge.rdx = phi i64 [ %bin.rdx, %middle.block ], [ 1, %L17.preheader ]
    br label %L17

L17:
    ; preds = %L17, %scalar.ph
    %value_phi4 = phi i64 [ %7, %L17 ], [ %bc.resume.val, %scalar.ph ]
    %value_phi6 = phi i64 [ %6, %L17 ], [ %bc.merge.rdx, %scalar.ph ]
; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    →#===#d2429055-58e9-4d84-894f-2e639723e078:4 within `jlfactorial`
; r @ int.jl:88 within `*`
    %6 = mul i64 %value_phi6, %value_phi4
; L
; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    →#===#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`
; r @ range.jl:901 within `iterate`
; r | @ promotion.jl:521 within `==`
    %.not = icmp eq i64 %value_phi4, %1
; L |
    %7 = add nuw i64 %value_phi4, 1
; L
    br i1 %.not, label %L32, label %L17

L32:
    ; preds = %L17, %middle.block,
    → %top
    %value_phi10 = phi i64 [ 1, %top ], [ %bin.rdx, %middle.block ], [ %6, %L17 ]
; @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    →#===#d2429055-58e9-4d84-894f-2e639723e078:6 within `jlfactorial`
    ret i64 %value_phi10
}

```

Step 3: Compiles to binary code

```

julia> @code_native jlfactorial(10)
.section    __TEXT,__text,regular,pure_instructions
.build_version macos, 14, 0
.globl _julia_jlfactorial_3726          ; -- Begin function
    →julia_jlfactorial_3726
.p2align    2
_julia_jlfactorial_3726:
    ; @julia_jlfactorial_3726
; r @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    →#===#d2429055-58e9-4d84-894f-2e639723e078:1 within `jlfactorial`
; %bb.0:
    ; %top
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    →#===#d2429055-58e9-4d84-894f-2e639723e078:3 within `jlfactorial`
; r | @ range.jl:5 within `Colon`

```

```

; r|| @ range.jl:403 within `UnitRange`
; r||| @ range.jl:414 within `unitrange_last`
    cmp x0, #0
    csel    x9, x0, xzr, gt
; LLL|
    cmp x0, #1
    b.lt    LBB0_3
; %bb.1:                                ; %L17.preheader
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↪===#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`
    cmp x9, #2
    b.hs    LBB0_4
; %bb.2:
    mov w8, #1
    mov w0, #1
    b      LBB0_7
LBB0_3:
    mov w0, #1
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↪===#d2429055-58e9-4d84-894f-2e639723e078:6 within `jlfactorial`
    ret
LBB0_4:                                ; %vector.ph
    mov x12, #0
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↪===#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`
    and x10, x9, #0x7fffffffffffffff
    orr x8, x9, #0x1
    mov w11, #1
    mov w13, #1
LBB0_5:                                ; %vector.body
                                ; =>This Inner Loop Header: Depth=1
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↪===#d2429055-58e9-4d84-894f-2e639723e078:4 within `jlfactorial`
; r| @ int.jl:88 within `*`
    madd    x11, x11, x12, x11
; L|
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↪===#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`
    add x14, x12, #2
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↪===#d2429055-58e9-4d84-894f-2e639723e078:4 within `jlfactorial`
; r| @ int.jl:88 within `*`
    mul x13, x13, x14
    mov x12, x14
    cmp x10, x14
    b.ne    LBB0_5
; %bb.6:                                ; %middle.block
; L|
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↪===#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`

```

```

mul x0, x13, x11
cmp x9, x10
b.eq LBB0_9
LBB0_7:                                ; %L17.preheader15
add x9, x9, #1
LBB0_8:                                ; %L17
                                ; =>This Inner Loop Header: Depth=1
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↳#==#d2429055-58e9-4d84-894f-2e639723e078:4 within `jlfactorial`
; | @ int.jl:88 within `*`
mul x0, x0, x8
; |
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↳#==#d2429055-58e9-4d84-894f-2e639723e078:5 within `jlfactorial`
; | @ range.jl:901 within `iterate`
add x8, x8, #1
; |
cmp x9, x8
b.ne LBB0_8
LBB0_9:                                ; %L32
; | @ /Users/liujinguo/jcode/ModernScientificComputing2024/Lecture2/3.julia.jl
    ↳#==#d2429055-58e9-4d84-894f-2e639723e078:6 within `jlfactorial`
ret
; |
                                ; -- End function

.subsections_via_symbols

```

```

julia> methods(jlfactorial)
# 1 method for generic function "jlfactorial" from Main:
[1] jlfactorial(n)
    @ REPL[4]:1

```

```

julia> using MethodAnalysis

julia> methodinstances(jlfactorial)
1-element Vector{Core.MethodInstance}:
MethodInstance for jlfactorial(::Int64)

```

Single method, multiple instances

```

julia> jlfactorial(UInt32(5))
120

julia> methodinstances(jlfactorial)
2-element Vector{Core.MethodInstance}:
MethodInstance for jlfactorial(::Int64)
MethodInstance for jlfactorial(::UInt32)

```

2.2.7 Summary: Key ingredients of performance

- **JIT compilation**: compile the code with **LLVM** compiler framework when a method is called for the first time;
- **Multiple dispatch**: invoke the correct method instance according to the type of multiple arguments;

2.3 Type and Multiple-dispatch

2.3.1 Julia Types

Julia has rich type system, which is not limited to the **primitive types** that supported by the hardware. The type system is the key to the **multiple dispatch** feature of Julia.

As an example, let us consider the type for complex numbers.

```
Complex{Float64}
```

where `Float64` is the **type parameter** of `Complex`. Type parameters are a part of a type, without which the type is not fully specified. A fully specified type is called a **concrete type**, which has a fixed memory layout and can be instantiated in memory. For example, the `Complex{Float64}` consists of two fields of type `Float64`, which are the real and imaginary parts of the complex number.

```
julia> fieldnames(Complex{Float64})
(:re, :im)

julia> fieldtypes(Complex{Float64})
(Float64, Float64)
```

Extending the example, we can define the type for a matrix of complex numbers.

```
Array{Complex{Float64}, 2}
```

`Array` type has two type parameters, the first one is the **element type** and the second one is the **dimension** of the array.

One can get the type of value with `typeof` function.

```
julia> typeof(1+2im)

julia> typeof(randn(Complex{Float64}, 2, 2))
```

Then, what the type of a type?

```
julia> typeof(Complex{Float64})
DataType
```

2.3.2 Example: define you first type

We first define of an abstract type `AbstractAnimal` with the keyword `abstract type`:

```
julia> abstract type AbstractAnimal{L} end
```

where the type parameter `L` stands for the number of legs. Defining the number of legs as a type parameter or a field of a concrete type is a design choice. Providing more information in the type system can help the compiler to optimize the code, but it can also make the compiler generate more code.

Abstract types can have subtypes. In the following we define a concrete subtype type `Dog` with 4 legs, which is a subtype of `AbstractAnimal{4}`.

```
julia> struct Dog <: AbstractAnimal{4}
    color::String
end
```

where `<:` is the symbol for subtyping. `A <: B` means `A` is a subtype of `B`. Concrete types can have fields, which are the data members of the type. However, they can not have subtypes.

Similarly, we define a `Cat` with 4 legs, a `Cock` with 2 legs and a `Human` with 2 legs.

```
julia> struct Cat <: AbstractAnimal{4}
    color::String
end

julia> struct Cock <: AbstractAnimal{2}
    gender::Bool
end

julia> struct Human{FT <: Real} <: AbstractAnimal{2}
    height::FT
    function Human(height::T) where T <: Real
        if height <= 0 || height > 300
            error("The tall of a Human being must be in range 0~300, got $(
↪height)")
        end
        return new{T}(height)
    end
end
```

Here, the `Human` type has its own constructor. The `new` function is the default constructor.

We can define a **fall back method** `fight` on the abstract type `AbstractAnimal`

```
julia> fight(a::AbstractAnimal, b::AbstractAnimal) = "draw"
```

where `::` is a type assertion. This function will be invoked if two subtypes of `AbstractAnimal` are fed into the function `fight` and no more **explicit** methods are defined.


We can define many more explicit methods with the same name.

```
julia> fight(dog::Dog, cat::Cat) = "win"
fight (generic function with 2 methods)

julia> fight(hum::Human, a::AbstractAnimal) = "win"
fight (generic function with 3 methods)

julia> fight(hum::Human, a::Union{Dog, Cat}) = "loss"
fight (generic function with 4 methods)

julia> fight(hum::AbstractAnimal, a::Human) = "loss"
fight (generic function with 5 methods)
```

where `Union{Dog, Cat}` is a **union type**. It is a type that can be either `Dog` or `Cat` . Union types are not concrete since they do not have a fixed memory layout, meanwhile, they can not be subtyped! Here, we defined 5 methods for the function `fight`. However, defining too many methods for the same function can be dangerous. You need to be careful about the ambiguity error!

```
julia> fight(Human(170), Human(180))
ERROR: MethodError: fight(::Human{Int64}, ::Human{Int64}) is ambiguous.

Candidates:
  fight(hum::AbstractAnimal, a::Human)
    @ Main REPL[37]:1
  fight(hum::Human, a::AbstractAnimal)
    @ Main REPL[35]:1

Possible fix, define
  fight(::Human, ::Human)

Stacktrace:
 [1] top-level scope
    @ REPL[38]:1
```

It makes sense because we claim `Human` wins any other animals, but we also claim any animal losses to `Human`. When it comes to two `Humans`, the two functions are equally valid. To resolve the ambiguity, we can define a new method for the function `fight` as follows.

```
julia> fight(hum::Human{T}, hum2::Human{T}) where T<:Real = hum.height > hum2.
    ↪ height ? "win" : "loss"
```

Now, we can test the function `fight` with different combinations of animals.

```
julia> fight(Cock(true), Cat("red"))
"draw"

julia> fight(Dog("blue"), Cat("white"))
"win"

julia> fight(Human(180), Cat("white"))
"loss"

julia> fight(Human(170), Human(180))
"loss"
```

Quiz: How many method instances are generated for `fight` so far?

```
julia> methodinstances(fight)
```

2.3.3 Julia number system

The type tree rooted on `Number` looks like:

```
Number├
├─ Base.MultiplicativeInverses.MultiplicativeInverse{T}
│   ├── Base.MultiplicativeInverses.SignedMultiplicativeInverse{T<:Signed}
│   └─ Base.MultiplicativeInverses.UnsignedMultiplicativeInverse{T<:Unsigned}├
├─ Complex{T<:Real}├
├─ Real
│   ├── AbstractFloat
│   │   ├── BigFloat
│   │   ├── Float16
│   │   ├── Float32
│   │   └─ Float64
│   └─ AbstractIrrational
└─ ...
```

The Julia type system is a tree, and `Any` is the root of type tree, i.e. it is a super type of any other type. The `Number` type is the root type of julia number system, which is also a subtype of `Any`.


```
julia> Number <: Any
```

There are utilities to analyze the type system:

```
julia> subtypes(Number)
3-element Vector{Any}:
 Base.MultiplicativeInverses.MultiplicativeInverse
 Complex
 Real

julia> supertype(Float64)
AbstractFloat

julia> AbstractFloat <: Real
true
```

The leaf nodes of the type tree are called **concrete types**. They are the types that can be instantiated in memory. Among the concrete types, there are **primitive types** and **composite types**. Primitive types are built into the language, such as `Int64`, `Float64`, `Bool`, and `Char`, while are built on top of primitive types, such as `Dict`, `Complex` and the user-defined types.

The list of primitive types

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end

primitive type Int8 <: Signed 8 end
primitive type UInt8 <: Unsigned 8 end
primitive type Int16 <: Signed 16 end
primitive type UInt16 <: Unsigned 16 end
primitive type Int32 <: Signed 32 end
primitive type UInt32 <: Unsigned 32 end
primitive type Int64 <: Signed 64 end
primitive type UInt64 <: Unsigned 64 end
primitive type Int128 <: Signed 128 end
primitive type UInt128 <: Unsigned 128 end
```

2.3.4 Extending the number system

Extending the number system in Julia is much easier than in object-oriented languages like Python. In the following example, we show how to implement

addition operation of a user defined class in Python.

```
class X:
    def __init__(self, num):
        self.num = num

    def __add__(self, other_obj):
        return X(self.num+other_obj.num)

    def __radd__(self, other_obj):
        return X(other_obj.num + self.num)

    def __str__(self):
        return "X = " + str(self.num)

class Y:
    def __init__(self, num):
        self.num = num

    def __radd__(self, other_obj):
        return Y(self.num+other_obj.num)

    def __str__(self):
        return "Y = " + str(self.num)

print(X(3) + Y(5))

print(Y(3) + X(5))
```

Here, we implemented the addition operation of two classes X and Y . The `__add__` method is called when the `+` operator is used with the object on the left-hand side, while the `__radd__` method is called when the object is on the right-hand side. The output is as follows:

```
X = 8
X = 8
```

It turns out the `__radd__` method of Y is not called at all. This is because the `__radd__` method is only called when the object on the left-hand side does not have the `__add__` method by some artificial rules.

Implement addition in Julian style is much easier. We can define the addition operation of two types X and Y as follows.

```
julia> struct X{T} <: Number
    num::T
end

julia> struct Y{T} <: Number
```

```

    num::T
end

julia> Base.:(+)(a::X, b::Y) = X(a.num + b.num);

julia> Base.:(+)(a::Y, b::X) = X(a.num + b.num);

julia> Base.:(+)(a::X, b::X) = X(a.num + b.num);

julia> Base.:(+)(a::Y, b::Y) = Y(a.num + b.num);

```

Multiple dispatch seems to be more expressive than object-oriented programming.

Now, supposed you want to extend this method to a new type `z`. In python, he needs to define a new class `z` as follows.

```

class Z:
    def __init__(self, num):
        self.num = num

    def __add__(self, other_obj):
        return Z(self.num+other_obj.num)

    def __radd__(self, other_obj):
        return Z(other_obj.num + self.num)

    def __str__(self):
        return "Z = " + str(self.num)

print(X(3) + Z(5))

print(Z(3) + X(5))

```

The output is as follows:

```

X = 8
Z = 8

```

No matter how hard you try, you can not make the `__add__` method of `z` to be called when the object is on the left-hand side. In Julia, this is not a problem at all. We can define the addition operation of `z` as follows.

```

julia> struct Z{T} <: Number
    num::T
end

julia> Base.:(+)(a::X, b::Z) = Z(a.num + b.num);

```

```
julia> Base.:(+)(a::Z, b::X) = Z(a.num + b.num);

julia> Base.:(+)(a::Y, b::Z) = Z(a.num + b.num);

julia> Base.:(+)(a::Z, b::Y) = Z(a.num + b.num);

julia> Base.:(+)(a::Z, b::Z) = Z(a.num + b.num);

julia> X(3) + Y(5)
X{Int64}(8)

julia> Y(3) + X(5)
X{Int64}(8)

julia> X(3) + Z(5)
Z{Int64}(8)

julia> Z(3) + Y(5)
Z{Int64}(8)
```

There is a deeper reason why multiple dispatch is more expressive than object-oriented programming. The Julia function space is exponentially large! If a function f has k parameters, and the module has t types, there can be t^k methods for the function f .

```
f(x::T1, y::T2, z::T3...)
```

However, in an object-oriented language like Python, the function space is only linear to the number of classes.

```
class T1:
    def f(self, y, z, ...):
        self.num = num
```

2.3.5 Example: Fibonacci number

The Fibonacci number is defined as follows.

```
julia> fib(x::Int) = x <= 2 ? 1 : fib(x-1) + fib(x-2)
fib (generic function with 1 methods)

julia> addup(x::Int, y::Int) = x + y
addup (generic function with 1 methods)

julia> @btime fib(40)
278.066 ms (0 allocations: 0 bytes)
```

```
102334155
```

Oops, it is really slow. There is definitely a better way to calculate the Fibonacci number, but let us stick to the current implementation for now.

If you know the Julia type system, you can implement the Fibonacci number in a zero cost way. The trick is to use the type system to calculate the Fibonacci number at compile time. There is a type `Val` defined in the `Base` module, which is just a type with a type parameter. The type parameter can be a number:

```
julia> Val{3.0}
Val{3.0}()
```

We can define the addition operation of `Val` as the addition of the type parameters.

```
julia> addup(::Val{x}, ::Val{y}) where {x, y} = Val(x + y)
addup (generic function with 2 methods)

julia> addup(Val{5}, Val{7})
Val{12}()
```

Finally, we can define the Fibonacci number in a zero cost way.

```
julia> fib(::Val{x}) where x = x <= 2 ? Val(1) : addup(fib(Val(x-1)), fib(Val(x
    ↪ -2)))
fib (generic function with 2 methods)

julia> @btime fib(Val{40})
0.792 ns (0 allocations: 0 bytes)
Val{102334155}()
```

Wow, it is really fast! However, this trick is not recommended. It is not a good practice to abuse the type system. You simply transfer the run-time to compile time, which violates the Performance Tips²³. On the other hand, we find the compiling time of the function `fib` is much shorter than the run-time. This is because the function `fib` is a **recursive function**. The compiler can not optimize the recursive function very well.

²³ <https://docs.julialang.org/en/v1/manual/performance-tips/>

2.3.6 Summary

- *Multiple dispatch* is a feature of some programming languages in which a function or method can be dynamically dispatched based on the **run-time** type.

- Julia's multiple dispatch provides exponential abstraction power comparing with an object-oriented language.
- By carefully designed type system, we can program in an exponentially large function space.

2.4 Tuple, Array and broadcasting

Tuple has fixed memory layout, but array does not.

```
tp = (1, 2.0, 'c')
```

```
(1, 2.0, 'c')
```

```
typeof(tp)
```

```
Tuple{Int64, Float64, Char}
```

```
isbitstype(typeof(tp))
```

true

```
arr = [1, 2.0, 'c']
```

1

2.0

```
'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
```

```
typeof(arr)
```

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

```
isbitstype(typeof(arr))
```

false

Broadcasting

```
x = 0:0.1:π
```

```
y = sin.(x)
```

```
using Plots
```

```
plot(x, y; label="sin")
```

```
mesh = (1:100)'
```

Broadcasting over non-concrete element types may be type unstable.

```
eltype(arr)
```

```
Any
```

```
arr .+ 1
```

```
2
```

```
3.0
```

```
'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)
```

```
eltype(tp)
```

```
Any
```

2.4.1 Julia package development

```
using TropicalNumbers
```

```
nothing
```

The file structure of a package

```
project_folder = dirname(dirname(pathof(TropicalNumbers)))
```

```
/home/runner/.julia/packages/TropicalNumbers/kRhOI
```

Unit Test

```
using Test
```

```
nothing
```

```
@test Tropical(3.0) + Tropical(2.0) == Tropical(3.0)
```

```
Test Passed
```

```
@test_throws BoundsError [1,2][3]
```

```
Test Passed
```

```
    Thrown: BoundsError
```

```
@test_broken 3 == 2
```

```
Test Broken
```

```
    Expression: 3 == 2
```

```
@testset "Tropical Number addition" begin
    @test Tropical(3.0) + Tropical(2.0) == Tropical(3.0)
    @test_throws BoundsError [1][2]
    @test_broken 3 == 2
end
```

```
Test.DefaultTestSet("Tropical Number addition", Any[Test Broken
    Expression: 3 == 2], 2, false, false, true, 1.707316862782449e9,
    ↪1.707316862807562e9, false, "none")
```

2.4.2 Case study: Create a package like HappyMolecules

With PkgTemplates.

<https://github.com/CodingThrust/HappyMolecules.jl>

```
julia> isbitstype(Complex{Float64})
```

```
julia> sizeof(Complex{Float32})
```

```
julia> sizeof(Complex{Float64})
```

But `Complex{BigFloat}` is not


```
julia> sizeof(Complex{BigFloat})

julia> isbitstype(Complex{BigFloat})
```

The size of `Complex{BigFloat}` is not true! It returns the pointer size!

2.4.3 How to measure the performance of your CPU?

The performance of a CPU is measured by the number of **floating point operations per second** (FLOPS) it can perform. The floating point operations include addition, subtraction, multiplication and division. The FLOPS can be related to multiple factors, such as the clock frequency, the number of cores, the number of instructions per cycle, and the number of floating point units. A simple way to measure the FLOPS is to benchmarking the speed of matrix multiplication.

```
julia> using BenchmarkTools

julia> A, B = rand(1000, 1000), rand(1000, 1000);

julia> @btime $A * $B;
12.122 ms (2 allocations: 7.63 MiB)
```

The number of FLOPS in a $n \times n \times n$ matrix multiplication is $2n^3$. The FLOPS can be calculated as: $2 \times 1000^3 / (12.122 \times 10^{-3}) = 165$ GFLOPS.

2.4.4 Case study: Vector element type and speed

Any type vector is flexible. You can add any element into it.

```
vany = Any[] # same as vany = []

typeof(vany)

push!(vany, "a")

push!(vany, 1)
```

Fixed typed vector is more restrictive.

```
vfloat64 = Float64[]

vfloat64 |> typeof

push!(vfloat64, "a")
```

Do not abuse the type system. e.g. a “zero” cost implementation

```
Val(3.0) # just a type

f(::Val{1}) = Val(1)

f(::Val{2}) = Val(1)
```

It violates the Performance Tips²⁴, since it transfers the run-time to compile time.

²⁴ <https://docs.julialang.org/en/v1/manual/performance-tips/>

```
let biganyv = collect{Any, 1:2:20000}
  @benchmark for i=1:length($biganyv)
    $biganyv[i] += 1
  end
end
```

```
let bigfloatv = collect{Float64, 1:2:20000}
  @benchmark for i=1:length($bigfloatv)
    $bigfloatv[i] += 1
  end
end
```

```
fib(x::Int) = x <= 2 ? 1 : fib(x-1) + fib(x-2)

@benchmark fib(20)
```

```
addup(::Val{x}, ::Val{y}) where {x, y} = Val(x + y)
```

```
f(::Val{x}) where x = addup(f(Val(x-1)), f(Val(x-2)))
```

```
@benchmark f(Val(20)) end
```

2.5 Publishing a Package

Now that you have an amazing package, it’s time to make it available to the public. Before that, there is one final task to be done which is to choose a license.

- GNU’s Not Unix! (GNU) (1983 by Richard Stallman)

Its goal is to give computer users freedom and control in their use of their computers and computing devices²⁵ by collaboratively developing and pub-

²⁵ https://en.wikipedia.org/wiki/Computer_hardware

lishing software that gives everyone the rights to freely run the software, copy and distribute it, study it, and modify it. GNU software grants these rights in its license²⁶.

²⁶ https://en.wikipedia.org/wiki/GNU_General_Public_License

- The problem of GPL Lisense: The GPL and licenses modeled on it impose the restriction that source code must be distributed or made available for all works that are derivatives of the GNU copyrighted code.

Case study: Free Software fundation v.s. Cisco Systems²⁷

²⁷ <https://www.notion.so/Wiki-53dd9dafd57b40f6b253d6605667a472>

Modern Licenses are: MIT²⁸ and Apache²⁹.

²⁸ https://en.wikipedia.org/wiki/MIT_License

²⁹ https://en.wikipedia.org/wiki/Apache_License

Appendix

This is the appendix.

3 *References*

