Beginning Julia Programming

For Engineers and Scientists

Sandeep Nagar

Apress

Beginning Julia Programming: For Engineers and Scientists

Sandeep Nagar
New York, USA

ISBN-13 (pbk): 978-1-4842-3170-8 ISBN-13 (electronic): 978-1-4842-3171-5
https://doi.org/10.1007/978-1-4842-3171-5

Library of Congress Control Number: 2017959880

Contents

CHAPTER 1

Introduction

1.1 Welcome to the Julian World

When you consider the vast sea of programming languages, learning yet one more can feel like an overwhelming task. Learning new programming schemes and constructs requires time, patience, and dedicated efforts, so there should be really strong reasons to invest in such a time-consuming activity. Julia is a programming language that provides such reasons. Since there are so many established programming languages, including Python, C, C++, Java, R, and MATLAB, you need to be really motivated to invest time in learning this new language, Julia. By the end of this chapter, I hope that you will see there are more than enough reasons to dive into Julian world.

Julia is touted to be the one programming language that meets all needs because it removes the requirement of knowing multiple languages. Most programming languages were designed to meet the needs at the time of their creation. For example, C was designed to be an efficient procedural programming language. C++ was developed to add object-oriented programming features to the already efficient and popular C language. Java also added new features to the area of objects. MATLAB was invented to ease the burden of coding efforts required to define a mathematical problem. Python grew with a similar philosophy, but ventured into areas where MATLAB was inefficient. Since the two languages were similar in structure, a lot of MATLAB coders shifted to Python without much effort. It was open source and modular as well, which added to the ease of using others' code. However, one of the main problems with this kind of development in computer science was that each programming language was only good in specific areas. As a result, users needed multiple programming languages for different tasks and then they needed to tweak them as required to make a needed software. Some programming languages like C and C++ were created for speed, while others were designed for efficient computation in their domain. The emergence of data science tasks required a language that not only was fast but that also had a feature to remove the need for multiple languages to complete a computational job.

Julia fits these new requirements almost perfectly. The web site http://julialang.org/ states the following:

> Julia provides the functionality, ease-of-use and intuitive syntax of R, Python, MATLAB, SAS or Stata combined with the speed, capacity and performance of C, C++ or Java [1].

Being open source in nature, Julia attracted a good number of developers to write modules that are now used for most work. Julia is one of the the fastest modern open source languages for data science, machine learning, and scientific computing.

With Julia's impressive set of facilities, you should now have enough reasons to explore the langauge. It is one of the newest programming languages that can be used for almost any type of computational tasks at present and, hopefully, in the future as well. In addition, many high-tech companies seek Julian coders.

If you have some experience with the Python programming language, you know that it became popular for a variety of reasons. It has an extremely simple learning curve. With open source architecture, millions of developers have poured in thousands of packages to perform various tasks. These packages are easy to use. You only have to import them using a simple command. Python can also work on a variety of platforms. It is object-oriented, which makes it one of the best-suited, high-level languages for simulation and computation in general. It can also run parts of code written in other programming languages. With these characteristics, it has become the most popular language among coders at the time of writing. But, it has one big problem: it is slow. Consequently, developers have found themselves in a fix when they need to write time-efficient code. Many times, they choose to write time-inefficient parts in faster languages like C or C++. This process makes the overall experience very enriching but cumbersome. The primary motivation to create Julia mainly arose from this issue.

Julia's creators themselves remark [2] on these issues:

> We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like MATLAB ®. We want something as usable for general programming as Python,

as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as MATLAB, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled. (Did we mention it should be as fast as C?)

The introduction of LLVM (Low Level Virtual Machine) enabled [3] this ideology. It became possible to design a language from the the onset that satisfies most requirements and, hence, eliminates the two-language approach. The LLVM-based JIT (just-in-time) compiler allows Julia to approach and often match the performance of `C/C++`. (Explaining this concept in detail is beyond the scope of this book and irrelevant for a beginner.)

1.2 JIT Compiler

All programming instructions end up as machine code that is run on hardware (hence, machine code is hardware-specific). Thus, faster code simply means efficient machine code. Machine code can be done by hand, but it is a tedious job. Assembly language (lower level language) is a symbolic representation of machine code and can be fed by hand to a hardware device, but it suffers from two major roadblocks:

1. It is not easy to read and write.
2. It is hardware-specific.

These two problems are overcome by higher-level language. Currently, a variety of higher-level languages exists. FORTRAN, C, C++, and so on, are compiler-based languages where writing and reading code is easier than assembly language; they convert to machine code quite efficiently. However, users are forced to provide a lot of information about how the code should execute and what data types are used. With ample information given to the compiler, the compiler builds machine code AOT (ahead-of-time). On the other hand, interpreted languages like Python generate machine code on the fly, that is, during program execution. Instead of a compiler, these languages use an interpreter that interprets each line of code to be ultimately converted into machine code. This allows a flexible and interactive environment that is loved by all. Programmers can even delegate defining data types and defining memory allocation tasks to the interpreter. But these facilities come at a cost since interpreted languages need time to create proper machine code.

JIT compilation brings together the best of both the AOT and interpreter worlds. The primary difference is that functions for specific tasks are compiled as requested. In some instances, the programmer supplies all the information to the compiler for efficient conversion to machine code. When some pieces of information are missing, the compiler tries to infer missing information based on its usage.

However, in some cases, JIT isn't the most optimum approach and fails miserably. This happens when type inference fails or the compiler has insufficient information to optimize effectively.

With these ideas in mind, the creators of Julia set upon an interesting journey that provided the world with one of the most promising programming languages. Learning about the history of its development can be very inspiring. The next section will describe how people from various backgrounds came together to collaborate and make Julia.

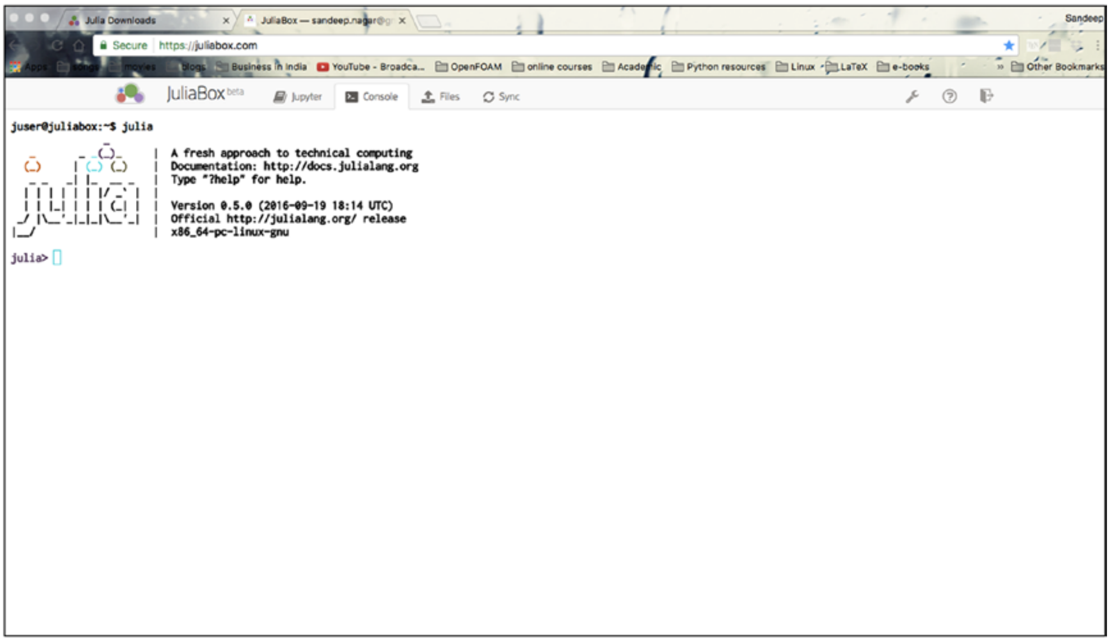
1.3 Brief History

A popular mantra in the Julia community is "walk like Python; run like C." Speed and ease of use has been the primary criteria for developing Julia. A team of four developers (Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman) created this practical language at MIT (Massachusetts Institute of Technology).

An article in *Wired* magazine illustrates the development of Julia [4]. The primary motivation for developing this language originated from the problem Karpinski faced when he designed a network simulator. For various parts of his project, he had to glue together different programming languages that were best-suited for the job. He went to get advice from Shah, who introduced him to Bezanson. Bezanson had concluded that the trade-off between various programming languages was avoidable. This conclusion addressed Karpinski's problem. The three developers brought in the mathematician Edelman, and they embarked on the project. Soon a high-level, high-performance, dynamic programming language for technical computing with syntax that is familiar to users of other technical computing environments began to take shape. It could rival the calculation speeds of C. It also provided the much-needed sophisticated compiler, allowed distributed parallel execution, and enabled higher numerical accuracy and an extensive mathematical function library. With this armor in command, the group released a series of research articles [5, 6, 7, 8], giving a glimpse into the wonderful Julian world.

1.4 Installation

Let’s dive into the Julian world and start learning about the power of the program. As mentioned before, the primary resource for Julia developers is its own web site [9]. At the time of writing, Current Release (v0.5.0) is downloadable from the Download tab of the web site. First, choose to download the version as per your operating system [10]. Installation for MacOSX, Windows, and Linux-based OS is discussed on the web site. An interesting way to explore Julia is using JuliaBox, where you don’t need to install any software. Instead, you can access a server running a notebook instance with a Julia kernel. (See Figure 1-1) Users can choose any of these methods to practice code.



```
juser@juliabox:~$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.5.0 (2016-09-19 18:14 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-pc-linux-gnu

julia> 
```

Figure 1-1. *JuliaBox console*

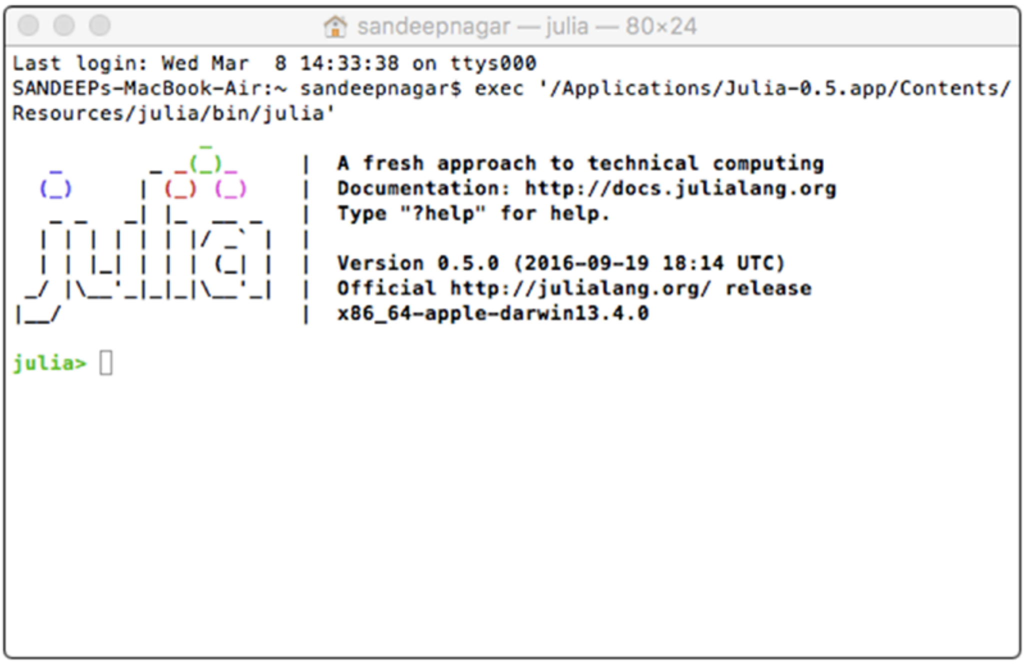
1.4.1 JuliaBox

If you have uninterrupted Internet access, you might like to work on Julia without downloading and installing software on your local machine. The web site for JuliaBox is given in the list of references [11]. Sign in using one of the accounts and open the Julia console using the tab options.

Each cell can be used to write code or textual information in a markup language. Beginners can simply choose to write the code in a cell and execute it by pressing Shift+Enter. The results will be displayed below the implemented cell and a new cell will get ready to take the next set of Julia commands.

1.4.2 MacOS

Julia runs on MacOS 10.7 and later releases. Installation of Julia on Mac can be easily performed by downloading the `Julia-<version>.dmg` file from its web site. This file contains `Julia.app`. Installation can be performed by copying the `Julia-<version>.app` to the hard drive (anywhere) or run from the disk image. Double-clicking the shortcut icon starts the Julia console (Figure 1-2) in a similar manner as the Julia console in JuliaBox (Figure 1-1). Multiple `Julia.app` binaries can coexist without interfering with each other. Thus, multiple versions of Julia can be installed and used without interfering with each other.



```
Last login: Wed Mar  8 14:33:38 on ttys000
SANDEEPs-MacBook-Air:~ sandeepnagar$ exec '/Applications/Julia-0.5.app/Contents/
Resources/julia/bin/julia'
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.5.0 (2016-09-19 18:14 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-apple-darwin13.4.0

julia>
```

Figure 1-2. *Julia console at MacOSX*

For uninstalling Julia, delete the `Julia.app` and the packages directory in `~/.julia`. If you would also like to remove your preferences files, remove `~/.juliarc.jl`.

1.4.3 Windows OS

Julia is available for Windows 7 and later. Both 32-bit and 64-bit OS versions are compatible with Julia installations. The first step is to download the `julia.exe` file (installer for your platform). It is worth noting that the 32-bit version of Julia works on both `x86` and `x86_64`. The `64`-bit version of Julia will only run on 64-bit Windows (`x86_64`) OS. Hence, it's important to check the version information of your installed windows OS. By default, it will install to your `AppData` folder. You may keep the default or choose your own directory (for example, `C:\Julia`). Next, run the downloaded `julia.exe` file. This will start the installation. After the installation is complete, a shortcut to the Julia program will appear on your desktop. Double-click this Julia shortcut in the unpacked folder to start the Julia environment, similar to the one seen in the JuliaBox (Figure 1-1). In the event that problems arise, some dependencies might be missing. The Windows README files contain information on dependencies. You have to install these dependencies to complete installation.

In case problems force you to uninstall Julia, delete the extracted directory and the packages directory in `\%HOME\%/.julia`. If you would also like to remove your preferences files, remove `\%HOME\%/.juliarc.jl` and `\%HOME\%/.julia_history`.

1.4.4 Linux OS

Installing from PPA for Ubuntu and Its Derivatives

For Ubuntu-based Linux distribution, add a repository and install Julia from the command line terminal by typing the following:

```
$ sudo add-apt-repository ppa:staticfloat/juliareleases
$ sudo add-apt-repository ppa:staticfloat/julia-deps
$ sudo apt-get update
$ sudo apt-get install julia
```

The main advantage of using this method is that Julia will be updated with the other software on an installed machine.

To remove Julia, type the following:

```
$ sudo apt-get purge julia
```

To remove the Julia repository so that, while updating, it does not seek to update Julia (make sure you remove the package before removing the repository), use the following code snippet:

```
$ sudo add-apt-repository --remove ppa:staticfloat/juliareleases
```

Installation on Fedora/RHEL/CentOS/SL/OEL

A copr (Cool Other Package Repo) [12] repository can be used in this case. This repository provides a Julia RPM package for Fedora and EPEL (RHEL/CentOS/SLES/OEL). If you are using RHEL, CentOS, Scientific Linux, or Oracle Enterprise Linux (version 5 or higher), first enable EPEL for your distribution version by running a command prompt:

```
$ sudo dnf copr enable nalimilan/julia
```

Another way is to copy the relevant `.repo` file available to `/etc/yum.repos.d/`. Finally, you can simply issue the following command-to-command terminal:

```
$ sudo yum install julia
```

New versions are built every night. If you have already installed Julia and you want to upgrade to the latest version, issue the following command to the terminal:

```
$ sudo yum upgrade julia
```

Building from Source Code

Building from source code is usually preferred by experienced programmers since it requires knowledge about handling files in Linux and changing permissions. Julia's source code is hosted at GitHub [13]. Hence, `Git` should be installed [14] in your system. Once this is done, you must satisfy the dependencies [15] first. The following instructions are also given at `README` file [16].

1.5 Package Installation

Apart from built-in Julia functions, users can add packages with specific functionalities and can even make their own packages (and subsequently release them to the Julia community). A list of packages is provided at the main web site [17]. It is clearly a very rich ecosystem of computing facilities. Packages for all sorts of computational tasks are already present and the Julia community appends this list on a regular basis.

For the purpose of adding and deleting packages to a Julia installation in a clean manner, Julia has a built-in package manager [18], but this requires an active Internet connection. Because the package manager uses Git internally to manage the package Git repositories, users may run into protocol issues (if behind a firewall, for example) when running `Pkg.add()` (to install packages). The following command can be run from the command line to tell Git to use HTTPS instead of the Git protocol when cloning repositories:

```
$ git config --global url."https://".insteadOf git://
```

1.5.1 Initialization of Package Manager

Initialization of a package manager can be done using `Pkg.init()` in the following manner. First, open the Julia app by clicking its icon. A Julia terminal opens up that has `julia>` as its command prompt. We shall issue a Julia command here. Try the following:

```
julia> Pkg.init()
```

The command will produce information about initialization of the package directory.

1.5.2 Updating Package Repository

Suppose we want to work with the package named `AlgebraicDiffEq`. Its source code can be found at its Git address [19]. It's advisable to update the metadata of the package repository to update the version of packages to the latest ones. Thus, we can issue the following command:

```
julia> Pkg.update()
```

Depending on your state of machine, it will update versions for installed packages. This may take quite some time depending on the speed of your machine and Internet connection. It is good practice to issue this command periodically for keeping packages in the most up-to-date condition.

1.5.3 Installing a New Package

The Julia package manager is declarative rather than imperative. This means that users simply issue the command about what they want. The package manager figures out what versions to install (or remove) to satisfy those requirements optimally—and minimally. Hence, a user just adds the name of the package to the list of requirements. The package manager then resolves the issues pertaining to its installation. This means that if some package had been installed because it was needed by a previous version of something you wanted but a newer version doesn't have that requirement anymore, updating will actually remove that package.

The installation of a package follows this pattern:

- Package requirements are in the file `~/.julia/v0.4/REQUIRE`.
- The name of the package to be installed is added to this file.
- `Pkg.resolve()` is then called to resolve the dependencies issues for final installation.

These tasks can be achieved by issuing a single command:

```
julia> Pkg.add("AlgebraicDiffEq")
```

The great benefit of a built-in installer is that it will take care of the dependency tree for the new installation. The list of dependencies can be found in the `REQUIRE` file of the package.

1.5.4 Removing a Package

Just as a package can be installated with great ease, a package can be removed by simply issuing the following command:

```
julia> Pkg.rm("AlgebraicDiffEq")
```

The procedure to remove a package is similar to its installation:

- Package requirements are in the file `~/.julia/v0.4/REQUIRE`.
- The name of the package to be removed is removed from this file.
- `Pkg.resolve()` is then called to resolve the dependencies issues for final installation.

Both `Pkg.add()` and `Pkg.rm()` are convenient ways for adding and removing requirements for a single package. However, in the case of multiple packages to be handled, we recommend using `Pkg.edit()`. Issuing this command lets users edit the contents of `~/.julia/v0.4/REQUIRE` manually, change the contents, and then update their packages accordingly. This process should only be done by experienced users so we have not discussed it here.

1.5.5 Status of Installed Packages

The command `Pkg.status()` prints out a summary of the state of packages you have installed. As an example, when this command is run on my machine, the following output is generated:

```
julia> Pkg.status()
1 required packages:
- IJulia                            1.6.0
9 additional packages:
- BinDeps                           0.7.0
- Compat                            0.30.0
- Conda                             0.7.0
- Homebrew                          0.5.8
- JSON                              0.13.0
- MbedTLS                           0.4.5
- SHA                               0.5.1
- URIParser                         0.2.0
- ZMQ                               0.4.3
```

It can be clearly seen that all the installed packages cache is being updated for further usage. Let's look at the process [18] of updating a package:

- The first step of updating packages is to *pull* new changes to the file found at address `~/.julia/v0.4/METADATA` and see if any new registered package versions have been published.
- Next, `Pkg.update()` attempts to update packages that are checked out on a branch and not dirty (that is, no changes have been made to files tracked by Git) by pulling changes from the package's upstream repository.

- Upstream changes will only be applied if no merging or rebasing is necessary (in other words, if the branch can be fast-forwarded).
- If the branch cannot be fast-forwarded, it is assumed that the users are working on it and will update the repository themselves.
- As a final step, the update process recomputes an optimal set of package versions to have installed to satisfy users' top-level requirements and the requirements of "fixed" packages.

1.5.6 Off-line Installation of Packages [18]

In case the Internet is not available, packages may be installed by copying the package root directory `Pkg.dir()` from a machine with the same operating system and environment. Issuing the command `Pkg.add()` performs the following within the package root directory:

- Adds the name of the package to `REQUIRE`
- Downloads the package to `.cache` and then copies the package to the package root directory
- Recursively performs step 2 against all the packages listed in the package's `REQUIRE` file.
- Runs `Pkg.build()`

1.6 Using Code in This Book

The code in this book can be simply written at the Julia prompt and executed by pressing the Enter key. If you are using JuliaBox or notebook format, you can execute a Julia cell by pressing the Shift and Enter keys simultaneiously. The comments are written starting with a # sign. They are optional and only assist conceptual clarity. Code is presented with a different font for visual clarity.

Julia files are stored with a `.jl` format. If the compiler path is well defined, then a Julia file can simply be executed by running the following command:

```
$julia <filename.jl>
```

The best way to use this book is to treat it as a workbook. Read each concept and run the sample code given with it. Try to run the code in the book and reason out the errors and warnings, if any. Finally, try to write your own code to understand the concept in greater detail.

1.7 Summary

In this chapter, we introduced the world of the Julia programming language and provided instructions for its proper installation on Windows, Linux, and MacOSX operating systems. These instructions might change over time as the availability of computing resources and configurations changes. Hence, we advise users to keep a tab on Julia's language web site [9] to get the most updated version of Julia. In the subsequent chapters, we will explore the structure of Julia in order to use it effectively.

1.8 Bibliography

[1] https://juliacomputing.com/

[2] http://julialang.org/blog/2012/02/why-we-created-julia

[3] http://llvm.org/

[4] www.wired.com/2014/02/julia/

[5] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012

[6] V. B. Shah, A. Edelman, S. Karpinski, J. Bezanson, and J. Kepner, "Novel algebras for advanced analytics in Julia," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pp. 1–4, Sept 2013

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," November 2014

[8] J. Bezanson, J. Chen, S. Karpinski, V. Shah, and A. Edelman, “Array operators using multiple dispatch: a design methodology for array implementations in dynamic languages,” in *ARRAY'14 Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, (New York, NY, USA), pp. 56–61, ACM, 2014

[9] http://julialang.org/

[10] http://julialang.org/downloads/platform.html

[11] https://juliabox.com/

[12] https://copr.fedorainfracloud.org/coprs/nalimilan/julia/

[13] https://github.com/JuliaLang/julia

[14] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

[15] https://github.com/JuliaLang/julia/blob/master/README.md#Required-Build-Tools-External-Libraries

[16] https://github.com/JuliaLang/julia/blob/master/README.md.

[17] http://pkg.julialang.org/

[18] http://docs.julialang.org/en/release-0.4/manual/packages/

[19] http://github.com/JuliaDiffEq/AlgebraicDiffEq.jl

CHAPTER 2

Object-Oriented Programming

2.1 Introduction

Julia is designed to be an object-oriented programming (OOP) language. This choice is inspired by the tremendous success of OOP languages in the computing community. The ease of defining a computational problem in OOP made it a famous computing paradigm; hence, Julia also adopted OOP. To learn Julia, getting to know OOP is a prerequisite. We have dedicated this chapter for this purpose.

2.2 Procedural Programming vs. OOP

When defining computational tasks, you can define a set procedure to solve a problem using blocks of data and connecting them as dictated by procedures. The paradigm that emphasizes setting procedures irrespective of the type of data and their different usage pattern is called *procedural programming*. On the other hand, OOP lays emphasis on objects and their relationships with one another using operators (acting on objects, they change their values and other attributes to solve a computational task). C++, Java, and Python are OOP languages as is Julia.

2.3 Idea of OOP

The idea of objects originated from observations of the day-to-day life around us. An object is understood as something that has a set of attributes and a related set of behaviors. This is what human babies learn first about their environments. For example, babies learn that a ball (*object*) has a color, shape, and size (*attributes*), and it rolls, skids, and bounces (*behavior*).

In the early 1970s, Alan Kay at Xerox PARC (Palo Alto Research Center) worked on the concept of OOP[1]. While working on a programming language called *Smalltalk* [2], he employed the ideas of OOP. The key ideas were based on applying computer programming to physical simulations. The real world can be imagined as various objects (having attributes and behaviors) interacting with each other. So, it is natural to adopt the same ideas while constructing a simulated world. All GUI-based systems inherit their philosophy from Kay's efforts toward these ideas and now all major programming languages follow them religiously.

2.4 Object

In the Julian world, everything is an object. But what is an object? An object is an abstract concept to signify an entity on which computation can be performed. Just like a physical object, a computer's object has a set of attributes and a related set of behaviors. A number, string, pictures, videos, files, and so forth, can be visualized as objects. Within numbers, you can subcategorize objects into other objects as integers, floating point numbers, and boolean numbers, or their collection in an ordered or unordered fashion. Within strings, you can have characters, words, sentences, and so on. Within files, it is possible to have text files, media files, data files, script files, and so forth.

2.5 Types of Object

An object has an associated type. Type dictates the memory storage requirements and what can be done computationally with an object. For example, `int` and `float` are distinct `types` of objects in the sense that `int` stores integers and `float` stores floating point numbers. These types will be discussed in greater detail in subsequent chapters. While `float` needs to store information regarding the number of digits preceding and succeeding the decimal point, an `int` object does not need to worry about the same information. Similarly, a complex number is stored in another type of object, aptly named `complex` since it needs to store two aspects of a complex numbers (i.e., their real and imaginary parts). In this way, they must be stored quite differently in a computer's memory and then be used quite differently in subsequent computations. The creation of an object necessarily requires the declaration of its type. Unless explicitly specified, Julia assigns the type dynamically. The type of object can be obtained by using the built-in function called `typeof()`, which takes the object whose types needs to be scanned.

Let's test this concept using an example. Even though the value of objects is numeric 1, they are stored as an integer, floating point number, and a character:

```
julia> a = 1
1

julia> b = 1.0
1.0

julia> c = '1'
'1': ASCII/Unicode U+0031
(category Nd: Number, decimal digit)

julia> typeof(a) # 64-bit Integer object
Int64

julia> typeof(b) # 64-bit Floating point object
Float64

julia> typeof(c) # Character object
Char
```

The integer object is quite different from a floating point object and string. Some functions will explicitly demand a particular type of object. In these cases, they need to be converted into one another. Sometimes the conversion is troublesome and occassionally it cannot be accomplished. Details of conversion problems are outlined with the help of number objects in Chapter 3.

2.6 Object Reference

Julian objects are represented as a reference to an object in a computer's memory. "A" points to a memory location. This memory location can be accessed using the built-in function `pointer_from_objref(x)` where `x` is the reference to the object. Similarly, the size of memory in bytes can be obtained by using the function `sizeof(x)`. This can be understood by a code that we have previously used:

```
julia> a = 1 # a references to a Int64 type integer
#  object
1

julia> b = 1.0 # b references to a Float64 type floating
#point object
1.0

julia> c = '1' # C  references to a Char type object
'1': ASCII/Unicode U+0031
(category Nd: Number, decimal digit)

julia> typeof(a)
Int64

julia> typeof(b)
Float64

julia> typeof(c)
Char

julia> pointer_from_objref(a) # memory address of a
Ptr{Void} @0x000000011b8240a0

julia> pointer_from_objref(b) # memory address of b
Ptr{Void} @0x0000000120804e30

julia> pointer_from_objref(c) # memory address of c
Ptr{Void} @0x0000000120a672e0

julia> sizeof(a) # a occupies 8 bytes (Int64)
8

julia> sizeof(b) # b occupies 8 bytes (Int64)
8

julia> sizeof(c) # c occupies 4 bytes (Char)
4
```

The command `whos()` gives the detailed state of the machine's memory consumption. For example, at the time of writing, the state of my machine shows the following:

```
julia> whos()
Base  34434 KB     Module
Core  12485 KB     Module
Main  41112 KB     Module
a         8 bytes  Int64
ans       4 bytes  Char
b         8 bytes  Float64
c         4 bytes  Char
```

The state of the machine clearly shows that apart from the modules `Base`, `Core`, and `Main`, four references were created in present namespace (or, in simpler words, working environment):

- `a` (8 bytes for `Int64` data type of object)
- `b` (8 bytes for `Float64` data type of object)
- `c` (4 bytes for `Char` data type of object)
- `ans` ((4 bytes for `Char` data type of object))
    - `ans` is automatically created and references the last used object at the Julia prompt.

2.6.1 Multiple References for the Same Object

Multiple references can refer to the same memory location of an object. This can be accomplished by assignment operator =. The symbol of the assignment operator should not be confused with the symbol for mathematical equality generally used while defining mathematical equations. Let's look at an example to further explain this concept:

```
julia> a = 1.5
1.5

julia> pointer_from_objref(a)
Ptr{Void} @0x0000000120804060
```

```
julia> b = a
1.5

julia> pointer_from_objref(b)
Ptr{Void} @0x0000000120804060

julia> c = b
1.5

julia> pointer_from_objref(c)
Ptr{Void} @0x0000000120804060

julia> a
1.5

julia> b
1.5

julia> c
1.5

julia> a = 1.2 # a now points to a different object
1.2

julia> pointer_from_objref(a) # different memory location
Ptr{Void} @0x0000000123509090

julia> b
1.5

julia> c
1.5

julia> a
1.2
```

A reference to object (named `a`) is created that refers to a floating point value 1.5. `pointer_from_objref(a)` shows that the address of the memory location is `0x0000000120804060` (a hexadecimal number representing a memory location). Now the object is assigned to another reference (named `b`). It has been verified that the memory addresses for a and b are the same. Please note that at the time of creation, only a was used for referencing. In a similar fashion, a new reference is created (named c) from a newly created reference named b. All three (a, b, and c) refer to the same memory location. When a is reassigned to a new object, it changes its reference, whereas b and c keep theirs.

2.7 Variables

A variable is a name associated (or bound) to a value—that is, a reference to data stored in a memory location. Data are treated like an object; a variable refers, or points, to the object. The value of the object can be changed while the code runs—hence, the name *variable*. (In other words, the object is able to store different values at different points of time.) For example, in the following code, a variable named a can store an integer, then another value of the integer, then a floating point number, then a string, and finally a rational number. Hence, the value of a can be variable.

```
julia> a = 1 # 'a' refers to integer object valued as 1
1

julia> a = 2 # Now 'a' refers to a new integer object
#valued as 2
2

julia> a = 3.4 # Now 'a' refers to a new floating point
#object valued as 3.4
3.4

julia> a = "hi" # Now 'a' refers to a string valued
# as "hi"
"hi"

julia> a =  1//2 # Now 'a' refers to a rational
# number 1/2
1//2
```

2.7.1 Naming a Variable

Variable names are case-sensitive and do not have any semantic meaning within Julia. For this reason, keywords cannot be used as variable names. Julia provides an extremely flexible system of naming a variable. Even Unicode characters are allowed for variable names. In the Julia REPL and several other Julia editing environments, Unicode characters are invoked by issuing LaTeX commands for [3] them and then pressing Tab. For example, to give the variable the name a, you would need to type `\alpha` and then press the Tab key to see α as the variable name.

A few rules exist regarding variable names:

- Variable names must begin with a letter (A–Z or a–z), underscore, or a subset of Unicode code points greater than 00A0.
- Subsequent characters may also include special characters (for example, &, @, %, ^, #, and so forth) and digits (0–9) or other Unicode characters.
- Keywords are not allowed for naming variables.

2.7.2 Naming Style Convention [4]

To maintain uniformity, Julia proposes a styling convention (which is suggested, but is not compulsory). The main aspects of style conventions are the following:

- Names of variables are written in lowercase alphabet letters.
- Word separation can be indicated by underscores ("_"), but use of underscores is discouraged unless the name would be hard to read otherwise.
- Names of types and modules begin with an uppercase letter and word separation is shown with uppercase letters instead of underscores.
- Names of functions and macros are lowercase, without underscores.
- Functions that write to their arguments have names that end in "!". These are sometimes called "mutating" or "in-place" functions because they are intended to produce changes in their arguments after the function is called, not just return a value.

2.8 Summary

In this chapter, we have discussed the basic paradigm of OOP and how Julia truly justifies its role as an OOP candidate. Dealing with objects makes the tasks modular as the flexible nature of computation gives freedom to the developer to explore dimensions of the computational tasks in a variety of ways. Different physical systems can be easily simulated since you just need to define an appropriate computational object and declare its properties as associated functions, called *methods*. Objects can be referenced by variable names and calling by reference makes it easy to change the values.

2.9 Bibliography

[1] http://propella.sakura.ne.jp/earlyHistoryST/EarlyHistoryST.html

[2] http://web.cecs.pdx.edu/~harry/musings/SmalltalkOverview.html

[3] http://detexify.kirelabs.org/symbols.html

[4] https://docs.julialang.org/en/stable/manual/variables/

CHAPTER 3

Basic Math with Julia

3.1 Introduction

In this chapter, we will explore how Julia can be used to perform simple mathematical calculations that are the basis of most computational tasks. A basic knowledge of high-school-level mathematics is required for understanding the contents of this chapter. The chapter will include illustrations that represent mathematical numbers of various kinds and their algebraic operations as well as other operations used to define mathematical computations. In fact, Julia proves to be a good option while teaching basic mathematics due to its simple learning curve.

It is important to note that apart from performing basic mathematical computation, Julia is a good candidate for high-performance computing. The general conception that high-performance computing means working with super computers is slowly and steadily being replaced because cheap and powerful computation power is readily available. A cluster of Raspberry Pi computers is a poor man's (financially poor, academically rich!) super computer. GPU (graphics processor unit)-based mini super computers are within the reach of the common man now, but the role of a faster programming language cannot be ignored here.

Laptops and desktops with few GHz multicore processors and between 2 and 8GB RAM have become the worldwide norm. If you can use these machines to perform high-performance computations, then the need for expensive computing systems becomes obsolete. This removes the fundamental roadblock for researchers and students from economically challenged social structures. If the solutions can be presented within an open source framework, value is added to their ease of accessibility.

The Julia programming language satisfies most of these conditions. It is open source, it supports multiparallel processing, it has a flat learning curve, and it boasts a speed comparable to C/C++. Hence, Julia is fast becoming popular, especially for the task of data analytics dealing with huge amounts of data that need to be crunched quickly. Since the base of such a job is mathematics, let's start learning how Julia treats basic mathematics within its basic framework (without using additional packages).

3.2 REPL

Julia comes with a full-featured, interactive, command-line REPL (Read-Eval-Print Loop) built into the executable. The interactive shell of the Julia programming language is commonly known as REPL because

- it *reads* what a user types,
- the compiler *evaluates* what it reads,
- it *prints* out the return value after evaluation, and
- it *loops* back and does it all over again.

As soon as we click the Julia shortcut, we obtain the REPL environment, as shown in Figure 1-2. The prompt is obtained as `julia>` and the cursor blinks at this prompt. It is waiting for the input to be read and evaluated, and then it prints the output of the evaluation and waits for the next input. A lot of similar environments exist in the computing world. Linux's shell, Python's interactive environment, MATLAB's interactive environment, and so on, follow the same philosophy.

In addition to allowing quick and easy evaluation of Julia statements, it has

- a searchable history,
- tab completion,
- many helpful key bindings, and
- dedicated help and shell modes.

3.2.1 Hello World!

Let's print the string `"Hello World,"` the very first program in the world of computing:

```
julia> println("Hello  World")
Hello World
```

When you feed the Julia command as the words `println("Hello World")` at the command prompt `julia>`, Julia *reads* this statement in the sense that it searches for the built-in function `println` and feeds it a string (defined by enclosing characters in a pair of quotation marks). This is *evaluated* by the Julia compiler as per definition of the `println` function. The function simply displays on the command prompt whatever string is fed to it. Hence, the result of evaluating `println` is *printed* on the console as the words `Hello World`. As soon as this is done, the environment goes back to the Julia prompt `julia>`, waiting for the next input.

3.2.2 I/O at REPL

Let's experiment with giving inputs and observing outputs at REPL:

```
julia> 2
2

julia> -2
-2

julia> 2.
2.0

julia> println(2-2.0)
0.0

julia> println("2-2.0")
2-2.0
```

- When we entered the number 2 at the prompt, it was evaluated as the *value* 2, which looks like a positive integer number in mathematical terms. It's important to remember that the *value* in the computer may or may not be an exact *mathematical* quantity in some cases.
- Similarly, when we entered the number -2 at the prompt, it was evaluated as the *value* -2, which looks like a negative integer number in mathematical terms.
- When we entered the number 2. at the prompt, it was evaluated as the *value* 2.0, which looks like a positive real number in mathematical terms. Thus, writing numbers before and after the decimal points in Julia is optional.

- When the value 2 is given to the built-in function `println`, it evaluates it to be as the value 2. It looks like nothing has happened, but that is not the case.
- When the mathematical expression 2 - 2.0 is fed to the built-in function `println`, it *evaluates* the expression and *prints* the output as the value 0.0.
- When the mathematical expression 2 - 2.0 is fed to the built-in function `println` as a string, it prints the string just like it printed the string `Hello World` before.

It can be noted that the Julia function `println` takes care of the fact that it might get a different *type* of data and must act accordingly. When it got a mathematical expression, it evaluated the same as per the rule of mathematics. When it got the same as a string, it just displayed it at the terminal by printing on the computer screen below the prompt. Let's look at what happens if `println` gets multiple values (separated by commas):

```
julia> println("2-2.0","@#%^&! ",2+2)
2-2.0@#%^&! 4
```

The strings `"2-2.0","@#%^&!"` is printed as such and the mathematical expression 2 + 2 is evaluated. The result, 4, is printed after the string because it appears in this order as input to the function. Please note that whitespace is also one of the characters in a string and it is also printed as a whitespace. Whitespace does not mean that it prints a *white-colored* space, but rather it prints nothing in the sense that it prints a space colored the same as the background color of the terminal.

3.2.3 Tab Completion

Just like Linux's shell as well as MATLAB's and Python's interactive environments, Julia's REPL supports tab completion. You can enter the first few characters of a function or type and then press the Tab key to get a list of all matches:

```
julia> pri
primitive type    print_shortest    println
print             print_with_color
```

Please note that after writing `pri`, you need to press the Tab key to get options and to get the output, as shown in the following section. It either completes it if it finds a unique option, or else it just prints all possibilities for you to choose. It helps reduce syntax errors and proves to be a great help while coding.

3.2.4 Seeking Help from Julia

The best way to learn Julia is to ask for help from the language itself! We saw a variety of ways in which the function `println()` can be used. Suppose we wish to learn more about it. You can write `? println` at the Julia prompt. It will output a brief description of its usage. As soon as it encounters `?`, REPL goes into *help mode* (prompt changes from `julia>` to `?help>`) where anything written is searched within help files:

```
help?> println
search: println print_with_color print
print_shortest sprint @printf isprint

println(io::IO, xs...)

Print (using print) xs followed by a newline.
If io is not supplied, prints
to STDOUT.
```

If REPL cannot find a match for a query, it suggests similar words, assuming the user has made a mistake while typing:

```
help?> clear
search: clear! ClusterManager

Couldn't find clear
Perhaps you meant clear!, close,
clamp, cld, ceil, Cchar, cat, cor or Char
No documentation found.

Binding clear does not exist.
```

Another use of the Tab key is to write LaTeX math symbols such as π, α, μ, and so on. In addtion to using ASCII [1] characters (128 in number), Julia supports printing Unicode [2] characters ($< 128,000$ characters).

3.2.5 Shell Mode

Linux shell commands are quite useful to execute processes. Julia REPL has a shell mode for this purpose. A semicolon (`;`) activates the shell mode. It can be exited by pressing the Backspace key at the beginning of the line:

```
julia>;
shell> ls
Applications
Desktop
Documents
Downloads
```

3.2.6 Search Mode

All the executed lines get saved to a history file that can be searched. For this purpose, a search mode needs to be enabled. This mode is activated by pressing the Control key with the R key. The command prompt changes to `(reverse-i-search)`'':`. Now, as the query is typed, the search query will appear in the quotes. Just like Ctrl+R activates reverse search, Ctrl+S activates a forward search.

3.2.7 Key Bindings

Julia makes use of key bindings as a main feature for working with REPL. A list of some useful key bindings is given in Table 3-1.

Table 3-1. *Some Important Key Bindings*

`^D`	Exit (when buffer is empty)
`^C`	Exit (interrupt or cancel)
`^L`	Clear console screen
Return/Enter	New line, executing if it is complete
`?,;`	Enter help or shell mode (when at start of a line)
`^R,^S`	Incremental history search

For a full list of key bindings, users are advised to check the reference number [3].

3.2.8 Version Information

`versioninfo()` prints the information about the version of Julia that is installed on a particular machine, as in the following example:

```
julia> versioninfo()
Julia Version 0.6.0
Commit 903644385b (2017-06-19 13:05  UTC)
Platform Info:
OS: macOS (x86_64-apple-darwin13.4.0)
CPU: Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz
WORD_SIZE: 64
BLAS: libopenblas (USE64BITINT DYNAMIC_ARCH NO_AFFINITY Haswell)
LAPACK: libopenblas64_
LIBM: libopenlibm
LLVM: libLLVM-3.9.1 (ORCJIT, broadwell)
```

3.3 Some Experiments with Numbers

Julia is an excellent tool for numerical computing owing to its elaborate system of handling numbers flawlessly for complex mathematical calculations. Understanding how numbers are defined within the Julian world is critical for a user before attempting to create complex mathematical structures with Julia.

3.3.1 Number Systems

Mathematics entertains many different number systems in common use. For example:

- The **integers**: $I = \{\ldots - 3, -2, -1, -, 1, 2, 3\ldots\}$
- The **rational** numbers: $\left\{\frac{p}{q} : p, q \in I, q \neq 0\right\}$
- The **real** numbers: $R = \{x : \infty < x < \infty\}$
- The **complex** numbers: $\{a + bi : a, b \in R\ i^2 = -1\}$

Special subsets are defined on top of these definitions, such as the natural numbers {0, 1, 2...}, the even numbers, the odd numbers, the positive numbers, non-negative numbers, and so on. Mathematically, these number systems are naturally nested within each other since integers are rational numbers that are real numbers, which can be viewed as part of the complex numbers.

Julia defines each number system as a data type. In other words, Julia creates a type of numbers around which rules of their algebra are defined. Users must understand how Julia defines the type for a particular number for its appropriate usage. An excellent resource, in addition to this book, is at reference number [4].

3.3.2 Julia as Calculator

We have already seen that Julia REPL behaves like a calculator. You can feed a particular calculation involving numbers at its terminal and it will output the result of the calculation.

Integers and real numbers are treated differently for a digital computer. Hence, 5 and 5.0 are two distinct entities. Consequently, when two integers (2 and 3 are added), the result is an integer:

```
julia> 2+3
5
```

It can be noted that when 2 (an integer) and 3.0 (a real number) are added, the result is an integer (5).

```
julia> 2.+3
5
```

However, when two real numbers (2.0 and 3.0) are added, the result is a real number (5.0).

```
julia> 2.+3.
5.0
```

The complex number $A+iB$ is represented as `A+B im` in Julia. Other languages use the alphabet `i` or `j` to signify the imaginary part of a complex number, but Julia uses the set of alphabets `im` deliberately for this purpose because `i` and `j` are used conventionally while defining counters in loops.

When an integer (in other words, real number) is added to a complex number, the result is a complex number but its components (real part and imaginary part) get changed accordingly:

```
julia> 2 + (2+3im)
4 + 3im

julia> 2. + (2+3im) #real number is afloating point number
4 + 3im

julia> 2. + (2.0+3im) #complex number's real part
# is floating point number
4.0 + 3.0im

julia> 2 + (2.0+3im)
4.0 + 3.0im
```

Rational numbers have numerators and denominators. Julia uses the command `Rational(A,B)` to mean the mathematical rational number $\frac{A}{B}$. Let's check how rational numbers behave w.r.t addition with other kinds of numbers:

```
julia> Rational(2,3)
2//3

julia> Rational(2,3)+2
8//3
```

Hence, we get the following result:

$$\frac{2}{3}+2=\frac{8}{3}$$

Let's see what happens if we use a real number instead of an integer:

```
julia>  Rational(2,3)+2.0
2.6666666666666665
```

It is worth noting that the output is no longer represented as a rational number, but instead as a real number:

$$\frac{2}{3}+2.0=2.6666666666666665$$

Let's check what happens when we add a rational number and a complex number:

```
julia> Rational(2,3)+ (2+3im)
8//3 + 3//1*im
```

$$\left(\frac{2}{3}\right)+(2+3i)=\frac{8}{3}+\frac{3}{1}i$$

As expected mathematically, the result is printed as a complex number with both real and imaginary parts defined as rational numbers.

The following example demonstrates what happens when we add a rational number, a complex number, and a real number:

```
julia> Rational(2,3)+ (2+3im) + 3.0
5.666666666666666 + 3.0im
```

$$\left(\frac{2}{3}\right)+(2+3i)+3.0=5.666666666666666+3.0i$$

The output is a complex number where real and imaginary parts are represented by real numbers.

Let's scan for irrational numbers such as *π* and *e,* which are predefined in Julia as `pi` and `e`:

```
julia> pi
π = 3.1415926535897...

julia> e
e = 2.7182818284590...
```

The following examples shows how they behave when added with integers, rationals, complex numbers, and other irrational numbers:

```
julia> e+pi
5.859874482048838

julia> e+2
4.718281828459045

julia> e+2.0
4.718281828459045
```

```
julia> e+Rational(2,3)
3.3849484951257116

julia> e+(2+3im)
4.718281828459045 + 3.0im
```

Efforts to define a rational number with two irrational numbers, π and e as numerator and denominator, fail:

```
Rational(pi,e)
ERROR: MethodError: no method
matching  Rational(::Irrational{:pi}, ::Irrational{:e})
```

Both error messages indicate syntax errors while matching the input error types and finding incompatibility in doing so.

Boolean numbers in Julia are depicted by `true` and `false`:

```
julia> true
true

julia> false
false
```

It is meaningless to add boolean numbers to natural numbers of any kind because you would normally get error messages of incompatibility. However, this is not the case with Julia:

```
julia> true + 1
2

julia> true + 0
1

julia> true + 0.1
1.1

julia> true + Rational(2,3)
5//3
```

```
julia> true + pi
4.141592653589793

julia> true + (2 + 3im)
3 + 3im
```

The boolean numbers `true` and `false` have numerical values `1` and `0`. This fact should be taken into consideration to avoid confusion and errors.

An obvious conclusion derived from these simple experimental calculations is that Julia not only functions as a calculator, but it also identifies the data type for calculations dynamically and performs accordingly. In other words, you do not need to *declare* the data type in advance as you do while writing code in C and C++. Julia identifies the data type from its value. It is also important to note that Julia matches data types of numbers for performing a particular operation. Some matches are incompatible. This indicates a sort of hierarchical structure of defining data type. Now let's investigate these concepts in greater detail.

3.4 Data Type for Integers and Real Numbers

Numbers are the basic building blocks of numerical mathematics. Representation of numbers as computable quantities for a computer requires them to be stored as data in a computer's memory. Since the memory is limited in nature, fixed spaces of memory are assigned for various number types like integers, real numbers, complex numbers, and so forth. Their representation has been briefly discussed in Section 3.3.2. The next section will discuss this topic in detail.

3.5 Type Assignment

It has become clear that integers and other numbers are stored and treated differently for arithmetic calculations. Whereas integers are stored as just one unit in all of the allocated memory space, real numbers are stored with information about numbers before and after decimal points, rationals as information about numerators and denominators, and complex numbers as information about their real as well as imaginary components. Julia differs from C and C ++ in this regard because it is a dynamically typed language. That is, the data type does not need to be declared explicitly. It is guaged by Julia depending on the value. Julia also maintains a hierarchy among data types for calculations to assign a data type to output if two or more data types are mixed within a calculation.

If we have a machine with a 64-bit architecture, then it can assign 64 bits for each entity. But would it be wise to use 64 bits to store the small values (say 0)? Automatic assignment faces this inefficient way of computation. Thus, it remains a developer's choice to either declare the data type strictly or let Julia take care of the same. When used judiciously, this speeds up computation and lessens the requirements of memory space.

3.5.1 Hierarchy Tree of Number Types

When you encounter a number of data types for mathematical numbers, you need a hierarchy tree for conversion of a data type from one to another. The hierarchy tree of Julia's type system for numbers is shown in Figure 3-1. Some of the data types have been introduced earlier in this chapter and others may seem very new.

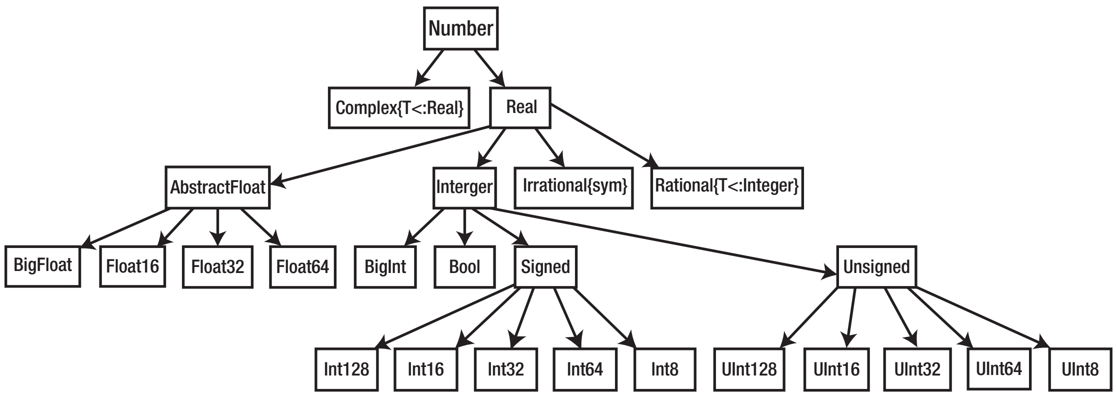


Figure 3-1. *Hierarchy tree for types of numbers [5]*

At the very top of the tree is the type `Number`. It has two subtypes named `Complex` and `Real`. Whereas the former does not have any subtype, the latter has further subtypes. They are depicted in the following illustration:

- `AbstractFloat` Floating point numbers, `Integer`, `Irrational`, `Rational`.
  - `AbstractFloat` has four subtypes:
    - `BigFloat:` Arbitrary precision decimal numbers
    - `Float16:` 16-bit precision decimal numbers
    - `Float32:` 32-bit precision decimal numbers
    - `Float64:` 64-bit precision decimal numbers

- `Integer` type has three subtypes:
    - `BigInt:` Arbitrary precision integers
    - `Bool:` Boolean Numbers
    - `Signed:` Signed Integers
        - `Int8:` 8-bit precision signed integer numbers
        - `Int16:` 16-bit precision signed integer numbers
        - `Int32:` 32-bit precision signed integer numbers
        - `Int64:` 64-bit precision signed integer numbers
        - `Int128`: 128-bit precision signed integer numbers
    - `Unsigned`: Unsigned Integers
        - `UInt8:` 8-bit precision unsigned integer numbers
        - `UInt16`: 16-bit precision unsigned integer numbers
        - `UInt32`: 32-bit precision unsigned integer numbers
        - `UInt64`: 64-bit precision unsigned integer numbers
        - `UInt128`: 128-bit precision unsigned integer numbers
    - `Irrational`: Irrational numbers
    - `Rational`: Rational numbers

Number Types

There are four basic number types in Julia:

- `Int`
- `Float`
- `Rational`
- `Complex`

The type of number dictates how it will be stored and how precisely the stored value is to the mathematical value it represents. To distinguish between these number types, Julia's parser uses the following easy-to-understand syntax rules [4]:

- Integers do not have decimal points.
- Floating point numbers have a decimal point (or are in scientific notation).
- Rationals are constructed from integers using the double division operator `//`.
- Complex numbers are formed by including a term with the imaginary unit `im`.

The abstract data types in Julia play a vital role in defining hierarchy, even if they do not play a direct role in defining a calculation. Abstract types allow code to be written generically for different concrete types such as `Int64`, `Float64`, `Complex`, `Rational`, and so on.

Precision

The variety of data types allows us to choose the precision of numbers for a particular mathematical calculation. To scan the precision of a data type, the built-in function `precision()` comes in handy. It outputs the effective number of bits in the mantissa (explained in Section 3.5.2):

```
julia> precision(BigFloat)
256

julia> precision(Float16)
11

julia> precision(Float32)
24

julia> precision(Float64)
53
```

Converting between different data types results in saving the computer's memory and speeding up calculations at the cost of precision. These decisions must be made by the developer beforehand by using the information given in this chapter. You do not always need higher precisions. For example, if you are working with dimensions of a bridge and the numbers are represented in units of meters, then you can usually work with a precision of $\frac{1}{10}^{th} m$. But if you are working with a calculation involving precision around Å= $10^{-9}m$, then you obviously need to be more accurate and precise.

3.5.2 Floating Point Arithmetic

Real numbers are represented as floating point numbers in a computer. The mapping of a real number to a computer's storage system is a formulaic representation (called *floating point representation*) [6]. Here real numbers are expressed in three parts: significand, base, and exponent.

For example, the value of π is 3.1415926535897… . Let's suppose that we have only four significant digits for a particular calculation, so the value can be rewritten as 3.1415. Now this number is represented as 31415×10^{-4} where 31415 is called *significand*, 10 is called *base*, and -4 is called *exponent*.

While assigning a number to the part called significand, the information about the number of significant digits is used. The significant figures of a number are digits that carry meaning contributing to its measurement resolution. In the previous case, we assumed only four significant digits depending on the requirements of calculations/ measurements. The term *floating point* refers to the fact that a number's *radix point* (decimal point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated as the exponent component, and thus the floating point representation can be thought of as a kind of scientific notation.

How to Store Floating Point Numbers

Computers can store numbers as floating point objects. A floating point object stores a number as follows:

$$\pm d_1 d_2 \ldots d_s \times \beta^e$$

where $d_i = 0, 1, 2 \ldots \beta - 1$ but $d_1 \neq 0$ and $m \leq e \leq M$ where $m \epsilon I^-$ and $M \epsilon I^+$.

Following are the three parts of a floating point number:

- Sign (±)
- Significand (Mantissa) ($d_1d_2...d_s$)
- Exponent (β)

Each part of a floating point number is stored at different memory locations and occupy a specified number of bits. How many bits are defined to which part? These questions have been answered by IEEE standards known as IEEE754 [6]. First, let's understand the concept of precision of a number representation.

1. **Single precision:**
   - Occupies 4 bytes = 32 bits. (See Figure 3-2.)
2. **Double precision:**
   - Occupies 8 bytes = 64 bits. (See Figure 3-3)
3. **Extended double precision:**
   - Occupies 80 bits. (See Figure 3-4.)
4. **Quadruple precision:**
   - Occupies 16 bytes = 128 bits. (See Figure 3-5.)

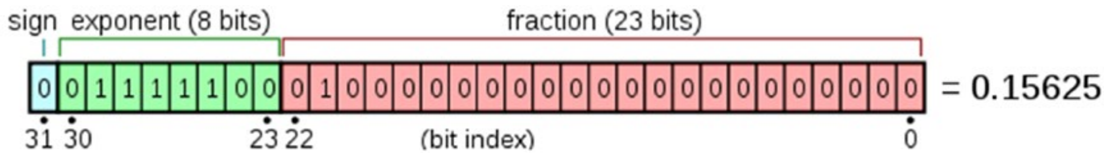


Figure 3-2. *IEEE 754 standard's single precision floating point number format [7]*

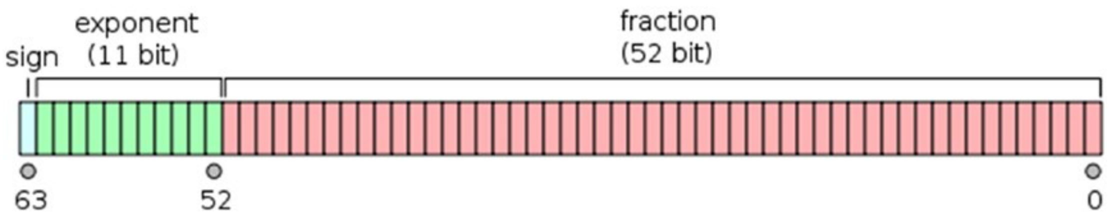


Figure 3-3. *IEEE 754 standard's double precision floating point number format [8]*

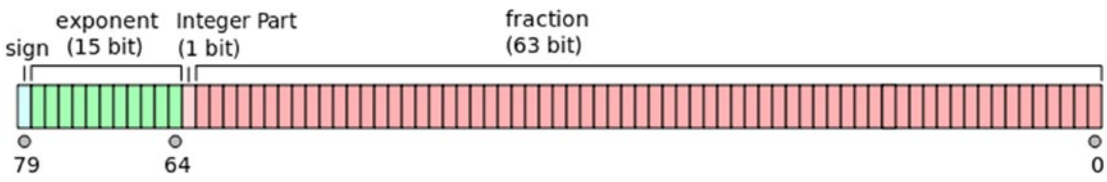


Figure 3-4. *IEEE 754 standard's extended precision floating point number format [9]*

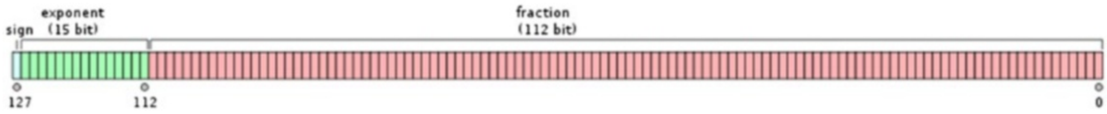


Figure 3-5. *IEEE 754 standard's quadruple precision floating point number format [10]*

Each version has one bit reserved for depicting the sign of a number. Others bits are divided for the significand and exponent. Since all numbers are stored as binary numbers in a computer, the base is always 2. Depending on the number of bits available for storage, the maximum numeral value can be defined for a data type.

For example, if *n* bits are available for the significand, then the maximum value can be 2^n. For the overall data type, if *m* bits are available for storage and one of them must be used for assigning the *sign bit*, then $2^n - 1$ is the maximum numeral value that can be stored by that data type. The limits are toward the two extremes (positive and negative numbers) for each data type. Hence, crossing the limits define overflow and underflow errors.

Julia follows the data type declaration as defined by the IEEE745 system. This system is discussed in Table 3-2.

Table 3-2. *Number Data Types of Julia and Their Properties*

Type	Signed?	No. of Bits	Smallest Value	Largest Value
`Int8`	Yes	8	-2^7	2^7
`UInt8`	No	8	0	$2^8 - 1$
`Int16`	Yes	16	-2^{15}	2^{15}
`UInt16`	No	16	0	$2^{16} - 1$
`Int32`	Yes	32	-2^{31}	2^{31}
`UInt32`	No	32	0	$2^{32} - 1$

(continued)

Table 3-2 (*continued*)

Type	Signed?	No. of Bits	Smallest Value	Largest Value
`Int64`	Yes	64	-2^{63}	2^{63}
`UInt64`	No	64	0	$2^{64} - 1$
`Int128`	Yes	128	-2^{127}	2^{127}
`UInt128`	No	128	0	$2^{128} - 1$
`Float16`	Yes	16	$-2^{10} \times 2^{5}$	$-2^{10} \times 2^{5}$
`Float32`	Yes	32	$-2^{23} \times 2^{8}$	$2^{23} \times 2^{8}$
`Float64`	Yes	64	$-2^{52} \times 2^{11}$	$-2^{20} \times 2^{11}$

It is important to perform back-of-the-envelop calculations for a particular problem to get an idea about maximum and minimum numbers expected during the running of a program. Accordingly, you can assign data types. If you do not perform the same, then Julia will assign them according to its own rules that might incur precision errors, underflow errors, and overflow errors.

Julia provides facility to input the number values as binary, octal, hexadecimal, or decimal numbers. The function `typeof()` can be used to probe the type of data. Let's start with integers. My computer has a 64-bit version of OS and a 64-bit version of the Julia compiler. Hence, the default word size of my system is 64 bits. With these definitions, let's scan from small to bigger integers:

```
julia> typeof(1234567890)
Int64

julia> typeof(-1234567890)
Int64

julia> typeof(12345678900000000000)
Int128

julia> typeof(-12345678900000000000
0000000000000000)
Int128
```

```
julia> typeof(12345678900000000000000
000000000000000000000000000)
BigInt
```

Julia does not assigns `UInt64` data type to all positive numbers by default. They must be declared by using `UInt64()` function. Unsigned integers are input and output using the `0x` prefix and hexadecimal (base 16) digits `0-9a-f`. (The capitalized digits `A-F` also work for input.) The size of the unsigned value is determined by the number of hex digits used:

```
julia> typeof(0x1)
UInt8

julia> typeof(0x111)
UInt16

julia> typeof(0x11111111)
UInt32

julia> typeof(0x11111111abcdef)
UInt64
```

Binary and octal representations are also supported, as follows:

```
julia> typeof(0b1)
UInt8

julia> typeof(0b110111111)
UInt16

julia> typeof(0b110111111000000
111000101001)
UInt32

julia> typeof(0b110111111000000
1110001010010101101010101010101)
UInt64

julia> typeof(0o11111)
UInt16
```

The built-in functions `typemin()` and `typemax()` can be used to find the minimum and maximum numbers that can be stored within a data type:

```
julia> typemin(Int8)
-128

julia> typemin(UInt8)
0x00

julia> typemin(UInt64)
0x0000000000000000

julia> typemin(Int64)
-9223372036854775808

julia> typemax(Int64)
9223372036854775807

julia> typemax(Int128)
170141183460469231731687
303715884105727

julia> typemin(Int128)
-170141183460469231731687
303715884105728
```

3.5.3 Overflow and Division Error

When a number bigger than the biggest possible number is stored within a data type, we encounter an overflow error message. Let's say that we assign the maximum storeable number in a variable named `a` and then we increase if by 1 and store the new value in a new variable named `b`:

```
julia> a = typemax(Int64)
9223372036854775807

julia> b = a + 1
-9223372036854775808

julia> typemin(Int64)
-9223372036854775808
```

```
julia> c = a + 2
-9223372036854775807

julia> typeof(a)
Int64

julia> typeof(b)
Int64

julia>  typeof(c)
Int64
```

Working within the default data type for integers (i.e., `Int64`), we see a wraparound behavior where adding 1 to the maximum number makes it the lowest. In mathematics, modular arithmetic is a system of arithmetic for integers, where numbers “wrap around” upon reaching a certain value. Arithmetic with Julian integers also follows the same concept. If a bigger number than defined for a particular data type is fed, an error message is displayed:

```
julia> Float64(2e900)
ERROR: syntax: overflow in numeric constant "2e900"
```

3.5.4 Floating Point Numbers vs. Real Numbers

You should keep in mind that floating point numbers are abstracts of real numbers. Sometimes this abstraction fails to represents the real numbers precisely.

The users must decide that if failure of this abstraction is insignificant, they can still confidently use floating point representation for the calculations while keeping in mind the errors. A few cases will make this clearer:

1. If $a,b,n \in \Re \exists c = \frac{a+b}{n}$ such that $c \in \Re$ where $\Re$ represent set of real numbers
   - This essentially says that between any two real numbers, there exists another real number.
   - However, this is not true for floating point numbers because floating point numbers are defined for a finite precision. (See Section 3.5.1.)

2. For the previous reason, floating point numbers are approximations of real numbers.

   - $\sqrt{7} \times \sqrt{7} - 7 = 0$ but Julia shows a finite small number for this calculation is due to the finite precision nature of the floating point number used to define $\sqrt{7}$:

     ```
     julia> (sqrt(7)*sqrt(7))
     7.000000000000001

     julia> (sqrt(7)*sqrt(7))-    7
     8.881784197001252e-16
     ```

   - Defined as a floating point, $\left(\frac{1}{2} + \frac{1}{6}\right) - \frac{5}{6} = 0$ is miscalculated. When it is converted to a rational type, it is calculated correctly:

     ```
     julia> 1//2 + 1//3 == 5//6
     true

     julia> (1//2+1//3)-(5//6)
     0//1

     julia> 1/2 + 1/3 == 5/6
     false

     julia> (1/2+1/3)-(5/6)
     -1.1102230246251565e-16
     ```

3. The property of associativity may not hold properly when defined with floating point numbers, but it will hold properly if defined with rational data type:

   ```
   julia> a = 1//10
   1//10

   julia> b = 2//10
   1//5

   julia> c = 3//10
   3//10
   ```

```
julia> a1 = 1/10
0.1

julia> b1 = 2/10
0.2

julia> c1 = 3/10
0.3

julia> (a+b)+c == a+(b+c)
true

julia> (a1+b1)+c1 == a1+(b1+c1)
false
```

3.5.5 Machine Precision

The concept of *machine precision* must be explained here. Machine precision is the smallest number of a particular data type. Julia provides a function to find that out, namely `eps()`. Try searching for `eps()` in `help` mode. The documentation is quite illustrative. If the data type is described as `T`, then `eps(T)` gives the distance between 1.0 and the next larger representable floating point value of data-Type `T`.

```
julia> eps(Float16)
Float16(0.000977)

julia> eps(Float32)
1.1920929f-7

julia> eps(Float64)
2.220446049250313e-16

julia> eps(BigFloat)
1.72723371101888892507727
03725600799142232000728
87256277004740694033718
360632485e-77
```

3.6 Arbitrary Precision Arithmetic

Apart from floating point precision, Julia also provides facility for arbitrary point precision by wrapping around the libraries GNU Multiple Precision Arithmetic Library (GMP) [11] and the GNU MPFR [12] Library. This is not discussed in details here since a basic user would not need this information too often.

For integers `BigInt` data types and for floating point numbers, `BigFloat` data types are made available using the following approach:

```
julia> a = BigInt(typemax(Int64))
9223372036854775807

julia> typeof(a)
BigInt

julia> a = BigFloat(1.0)
1.000000000000000000000
000000000000000000000
000000000000000000000
0000000000000000

julia> a = BigFloat(1.5)
1.500000000000000000000
000000000000000000000
000000000000000000000
0000000000000000
```

The problem exists when the user tries to change decimal literals to floating point numbers. For example:

```
julia> BigFloat(2.1)
2.100000000000000088817
84197001252323389053344
72656250000000000000000
00000000000
```

To overcome this problem, usage of big is recommended. big converts a number to a maximum precision representation (typically BigInt or BigFloat):

```
julia> a = big"2.1"
2.099999999999999999999999
999999999999999999999999
999999999999999999999999
9999999986
```

```
julia> typeof(a)
BigFloat
```

The default precision, nominally 256 bits, and the rounding mode of BigFloat can be changed using with_bigfloat_precision() and with_rounding() functions.

3.7 Numerical Conversion

When you wish to convert one data type to another, you must be aware that conversion might result in errors. For example, a floating point number being converted to an integer must be rounded off and will lead to round-off errors. There are various rules to round off, too. Also, when bigger numbers are converted to smaller-sized computer memory formats, inexactness is introduced, which is shown by *inexact* errors.

Julia supports three forms of numerical conversion, which differ in their handling of inexact conversions:

- T(x) or convert(T,x) converts x to a value of type T
    - Suppose we wish to convert the integer 3 into a Float64:

      ```
      julia> a = Int64(3)
      3

      julia> a1 = Float64(a)
      3.0

      julia> a2 = Int64(a1)
      3
      ```

```
julia> a3 = Int8(a2)
3

julia> a4 = convert(Float64,a3)
3.0

julia> a5 = convert(Int8,a4)
3
```

- If T is a floating point type, the result is the nearest representable value, which could be positive or negative infinity.
- If T is an integer type, an `InexactError` is raised if `x` is not representable by T.

```
julia> a = 2.1
2.1

julia> typeof(a)
Float64

julia> a1 = BigInt(a)
julia> a1=BigInt(a)
ERROR: InexactError()
Stacktrace:
[1] convert(::Type{BigInt},
::Float64) at ./gmp.jl:162
[2] BigInt(::Float64)
at ./sysimg.jl:24

julia> a1 = Int64(a)
ERROR: InexactError()
Stacktrace:
[1] convert(::Type{Int64},
::Float64) at ./float.jl:679
[2] Int64(::Float64)
at ./sysimg.jl:24
```

    - `x % T` converts an integer `x` to a value of integer type `T` congruent to `x` modulo `2^n`, where `n` is the number of bits in `T`. In other words, the binary representation is truncated to fit.

      ```
      julia> a = 128 %  Int8
      -128

      julia> a1 = 127 %  Int8
      127
      ```

    - Since 127 is the maximum number that can be stored in the type `Int8` ($2^7 - 1 = 127$), when the number 128 needs to be converted to the data type `Int8`, its bits are truncated to fit in.

- The rounding function `Rounding off` takes a type `T` as an optional argument. For example, `round(Int,x)` is shorthand for `Int(round(x))`:

  ```
  julia> round(Int8,127.2)
  127

  julia> round(Int8,125.9)
  126

  julia> round(127.2)
  127.0

  julia> round(125.9)
  126.0

  julia> Int8(round(127.2))
  127

  julia> Int8(round(125.9))
  126
  ```

- Other rounding functions are `floor()`, `ceil()`, and `trunc()`. The following are example code:

Name	Behavior	Return Type
`round(x)`	round x to the nearest integer	`typeof(x)`
`round(T,x)`	round x to the nearest integer	`T`
`floor(x)`	round x towards `-Inf`	`typeof(x)`
`floor(T,x)`	round x towards `-Inf`	`T`
`ceil(x)`	round x towards `Inf`	`typeof(x)`
`ceil(T,x)`	round x towards `Inf`	`T`
`trunc(x)`	round x towards 0	`typeof(x)`
`trunc(T,x)`	round x towards 0	`T`

```
julia> round(2.6)
3.0

julia> round(Int8,2.6)
3

julia> floor(2.6)
2.0

julia> floor(Int8,2.6)
2

julia> ceil(2.6)
3.0

julia> ceil(Int8,2.6)
3

julia> trunc(2.6)
2.0

julia> trunc(Int8,2.6)
2
```

3.8 Arithmetic Operators

Apart from defining mathematical numbers, users must also define arithmetic operators such as `+`,`-`,`*`, and `/` to perform arithmetic calculations as per given the algebra of the data type.

Some examples will make this clear:

```
julia> a = 1.0
1.0
```

Operator Symbol	Name	Behavior
+a	unary plus	Identity operation
-a	unary minus	Maps value to additive inverse of a
a+b	binary plus	Performs a plus b
a-b	binary minus	Performs a minus b
a*b	times	Performs a multiplied by b
a/b	divide	Gives quotient given when a divided by b is performed
a\b	inverse divide	Gives quotient given when b divided by a is performed
a%b	remainder	Gives remainder obtained by a divided by b
a^b	power	Perform a^b

```
julia> b = 1.5
1.5

julia> +a # Identity operation does
#not change  value
1.0

julia> -a # Gives additive inverse of 1.0 as -1.0
-1.0

julia> a+b # 1.0+1.5 = 2.5
2.5
```

```
julia> a-b # 1.0-1.5 = 0.5
-0.5

julia> a*b # 1.0 times 1.5 is 1.5
1.5

julia> a/b # Quotient of 1.0/1.5 is 0.66 ...
0.6666666666666666

julia> a\b # Quotient of 1.5/1.0 is 1.5
1.5

julia> a%b # Remainder of 1.0/1.5 is 1.0
1.0

julia> a^b # Remainder of 1.0 raised
#to the power 1.5 is 1.0
1.0
```

3.9 Boolean Numbers

Boolean numbers, along with boolean algebra, has framed the backbone of modern computing. George Bool developed boolean algebra to work with boolean numbers (`true` and `false` along with boolean operators such as `AND`, `OR`, `NOT`, and `XOR`. They can be used for comparison of quantities as well as making logical statements and finding their truth value.

3.9.1 Comparison of Mathematical Quantities

Two logical numbers, namely `true` and `false`, exist in Julia that can be used to perform boolean arithmetic operations. In their simplest form, they can be used to check for inequalities and equalities between quantities.

```
julia> a = 1.0
1.0

julia> b = 1
1
```

```
julia> a == b # Value of 1.0 and 1 is 1
true

julia> a!=-b # Mathematically 1.0 is not equal to -1
true

julia> a  <  b  #  Because  1.0=1  mathematically
false

julia> a <= b # Because atleast equality holds true
true

julia> a  >  b  #  Because  1.0=1  mathematically
false

julia> a>= b # Because atleast equality holds true
true
```

The following list of operators has been probed:

Operator Symbol	Meaning
`==`	equality
`!=`	inequality
`<`	less than
`<=`	less than or equal to
`>`	more than
`>=`	more than or equal to

Comparison of integers is straightforward for a computer as it just compares the bit values. Floating point numbers are a bit different in this respect. They are compared as per IEEE754 standard: [6]

- Positive zero is equal but not greater than negative zero.
- `Inf` is equal to itself and greater than everything else except `NaN`.
- `-Inf` is equal to itself and less then everything else except `NaN`.
- `NaN` is not equal to, not less than, and not greater than anything, including itself.

These statements can be checked easily, as follows:

```
julia> +0.0 == 0.0 #+ve zero is same as zero
true

julia> +0.0 == -0.0 # +ve zero is same as -ve zero
true

julia> +0.0 < -0.0
false

julia> +0.0 > -0.0
false

julia> Inf == -Inf #+ve infinity is not equal to -ve infinity
false

julia> Inf > Inf #+ve infinity isn't more than itself
false

julia> Inf > -Inf #+ve infinity is more than -ve infinity
true

julia> NaN  ==  Inf #Inf and  NaN  can't be  compared  valuewise
false

julia> Inf >  NaN
false

julia> -Inf > NaN
false

julia>  -NaN
NaN

julia> NaN  >  Inf
false

julia> NaN > -Inf
false

julia> NaN == NaN # Two NaN values aren't same
false
```

```
julia> NaN != NaN
true

julia> NaN <= NaN
false

julia> NaN >= NaN
false

julia> NaN > NaN
false

julia> NaN > NaN
false
```

To avoid discrepancies with operator behavior with different data types, Julia provides some built-in functions.

Function	Behavior
`isequal(x,y)`	x and y are identical
`isfinite(x)`	x is not `Inf` or `-Inf`
`isinf(x)`	x is equal to `Inf` or `-Inf`
`isnan(x)`	x is equal to NaN

```
julia> isequal(1.0, 1)
true

julia> isfinite(Inf)
false

julia> isfinite(-Inf)
false

julia> isfinite(1.0)
true

julia> isinf(Inf)
true
```

```
julia> isinf(-Inf)
true

julia> isinf(1.0)
false

julia> isnan(NaN)
true

julia> isnan(Inf)
false

julia> isnan(-Inf)
false

julia> isnan(1.0)
false
```

3.9.2 Chaining Comparisons

Comparisons can be arbitrarily chained in Julia:

```
julia> 1 < 2 > 3
false
```

`1<2>3` can be understood by first assigning `1<2` to `a`, which is valued `true`, and then `a>3` is calculated to be `false`:

```
julia> a=1 < 2
true

julia> a > 3
false
```

The order of evaluations in a chained comparison is undefined unless brackets are used. The statement is read from the right-hand side and successive comparison results are stitched together.

Expressions inclosed inside brackets are calculated first. The bracketed expressions are also read from the right-hand size and successive comparisons are stitched together. Following is an example:

```
julia> (1<2)>(3==3)
false

julia> 1<2
true

julia> 3==3
true

julia> true>true
false

julia> (1<2)>(3>3)
true

julia> 3>3
false

julia> true>false
true
```

3.9.3 Boolean Operators

The AND (&), NOT (! or ~), XOR ($), and OR (|) operators can be used to make complex logical statements. It is worth noting that boolean operators are mostly *bitwise operators*. That is, while performing comparison operations, they operate bitwise.

Expression	Behavior
a	bitwise NOT (NOT *a*)
a & *b*	bitwise AND (*a* AND *b*)
a | *b*	bitwise AND (*a* AND *b*)
a $ *b*	bitwise XOR (*a* XOR *b*)

(continued)

Expression	Behavior
a >>> b	logical shift right
a >> b	arithmetic shift right
a << b	logical/arithmetic shift left

A logical shift is a bitwise operation that shifts all the bits of its operand. On the other hand, an arithmetic shift is also a shift operator, sometimes termed a signed shift (even though it is not restricted to signed operands). Both these operators can be defined for left and right directions.

For binary numbers, it is a bitwise operation. (In other words, it shifts all of the bits of its operand by the given number of bit position(s).) The vacant bit positions are filled in. Instead of being filled with all 0s, as in a logical shift, when shifting to the right, the leftmost bit (usually the sign bit in signed integer representation) is replicated to fill in all the vacant positions (this is a kind of sign extension).

Let's first scan the basic boolean operators and then understand the bitwise shift operators. The variables named `a` and `b` are first defined to hold boolean values `true` and `false`. Using boolean operators, simple as well as complex logical statements can be made to obtain results:

```
julia> a = Bool(true)
true

julia> b = Bool(false)
false

julia> !a # NOT operator
false

julia> ~a # bitwise not
false

julia> a & b # AND operator
false

julia> a | b # OR operator
true
```

```
julia> a $ b # XOR operator
true

julia> a & (a | b) & a # A complex logical statement
true
```

3.10 Updating Operators

Every binary arithmetic and bitwise operator has an updating version, too. These operators assign the result of the operation back into its left operand.

They are quite simple to define. The updating version of the binary operator is formed by placing a = immediately after the operator.

Expression	Behavior
a+=1	a = a+1
a-=1	a = a-1
a*=2	a = a*2
a/=2	a = a/2
a\=2	a = 2/a
a%=2	a = a%2
a^=2	a = a^2
a!=a	a = !a
a&=2	a = a&a
a|=a	a = a | a

```
julia> a = 1.5
1.5

julia> a+=1 # a is now valued as 1.5+1 = 2.5
2.5

julia> a-=1 # a is now valued as 2.5-1=1.5
1.5
```

```
julia> a*=2 # a is now valued as 1.5*2=3.0
3.0

julia> a/=2 # a is now valued as 3.0/2 = 1.5
1.5

julia> a\=2 # a is now valued as 2/1.5 = 1.33 ...
1.3333333333333333

julia> 2/1.5 # verified
1.3333333333333333

julia> a=10 # redefining a to be values as 10
10

julia> a%=2 # a is now valued as 10%2 = 0
0

julia> 10%2  #  verified
0

julia> a=10 # redefining a to be values as 10
10

julia> a^=2 # a is now valued as 10^2 = 100
100

julia> a = Bool(true) # a is defined as boolean true
true

julia> a!=a # a is updated as (NOT a) and hence get
#valued as false
false

julia> a # a is verified to be values as false
false

julia> a&=a # a is updated with values of (a&a)
#i.e false&false
false
```

```
julia> a|=a # a is updated with values of
#(a|a) i.e false|false
false

julia> a$=a # a is updated with values of
$(a$a) i.e false$false
false
```

It is worth noting that the updating operator rebinds the variable on the left-hand side. As a result, the type of the variable may change.

```
julia> a = UInt8(12)
0x0c

julia> typeof(a)
UInt8

julia> a ^= 200.5
2.375963871483476e216

julia> typeof(a)
Float64
```

3.11 Operator Precedence

For the purpose of mathematical evaluations using mathematical numbers, it is important for a programming number to define operator precedence. For example, if we want to calculate

$$2+3*5/2^3.5$$

we must understand that the result depends on the order in which the mathematical functions are operated. Let's try the calculation on the Julia console:

```
julia> 2+3*5/2^3.5
3.3258252147247767
```

Now let's understand the order in which operators were applied to get this value. As per Julia's documentation [13], the following is the operator precedence defined:

Symbol	Meaning
^	Exponentiation
// and \\	Fractions
* / % & \	Algebraic operations
+ - | $	Addition
> < >= <= == === != !== <:	Comparison

```
julia> 2+3*5/2^3.5
3.3258252147247767

julia> 2^3.5 # First exponenttaion is applied
11.31370849898476

julia> ans # ans stored last calculated value
11.31370849898476

julia> 5/ans # Division operator
0.4419417382415922

julia> ans*3 # Multiplication operator
1.3258252147247767

julia> ans+2 # Addition
3.3258252147247767
```

The operator precedence logic follows famous BEDMAS (Bracket—Exponentiation—Division—Multiplication—Addition—Subtraction) rule of mathematics. It's recommended to use brackets for numbers that need to be calculated separately. For example:

$$(2+3)*(5/2)^{3.5}$$

will first solve the calcuations in each bracket and then apply operator precedence. Consequently, the result will be different:

```
julia>   (2+3)*(5/2)^3.5
123.52647110032733

julia> 5/2 # First the bracket is solved for 5/2 and 2+3
2.5

julia> ans^3.5 # Next exponentiation is carried out
24.705294220065465

julia> ans*(5) # result is added to 2+3 = 5
123.52647110032733
```

3.12 Summary

This chapter has discussed the ways in which Julia performs mathematical tasks. It makes a variety of objects for various types of numbers. Each object has specific methods to deal with mathematical operators. Knowledge of the usage of an operator with particular data lets you decide about their usage in a meaningful way. The hierarchy tree of the number system must also be understood in totality as conversion rules dictate how numbers will be dealt with in computational tasks. Decisions about using particular data types for computation involve an effective trade-off between precision and memory usage. Operator precedence is another important task discussed in this chapter since it decides the ultimate result. Also, the Julia REPL and its various modes come in handy when users perform mathematical computation.

3.13 Bibliography

[1] https://en.wikipedia.org/wiki/ASCII#ASCII_control_characters

[2] https://en.wikipedia.org/wiki/List_of_Unicode_characters

[3] http://docs.julialang.org/en/release-0.4/manual/interacting-with-julia/

[4] http://calculuswithjulia.github.io/toc.html

[5] https://commons.wikimedia.org/wiki/File:Type-hierarchy-for-julia-numbers.png

[6] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008

[7] https://en.wikipedia.org/wiki/IEEE_754-1985#/media/File:IEEE_754_Single_Floating_Point_Format.svg

[8] https://en.wikipedia.org/wiki/IEEE_754-1985#/media/File:IEEE_754_Double_Floating_Point_Format.svg

[9] https://en.wikipedia.org/wiki/Extended_precision#media/File:X86_Extended_Floating_Point_Format.svg

[10] https://commons.wikimedia.org/wiki/File:IEEE_754_Quadruple_Floating_Point_Format.svg

[11] https://gmplib.org/

[12] www.mpfr.org/

[13] http://docs.julialang.org/en/stable/

CHAPTER 4

Complex Numbers

4.1 Introduction

Having a basic understanding of preliminary mathematical constructs, you now need to understand how complex numbers are dealt with. Computations involving complex numbers can be found in almost all branches of science and mathematics. All Julia-based numerical computation developers must understand a variety of ways of defining complex numbers and their mathematics to compute efficiently.

The world of complex numbers encompasses important scientific domains. When used for descriptions of reality, they present more enriched pictures of physical phenomena than real numbers. Every programming language that boasts of performing mathematical calculations robustly must handle complex numbers with ease. Julia truly is one such language. Complex numbers are defined in simple terms and most functions for their handling are present. In addition, their usage in calculations with other data types is quite flexible and flawless. These qualities make it one of the best choices to perform complex analysis tasks.

4.2 Defining Complex Numbers

The first and most important task in defining complex numbers is to define a symbol for the complex number $i = \sqrt{-1}$. A global constant represented by `im` is used to *i*. This represents the principal $\sqrt{-1}$. Mathematicians usually use the alphabets `i` and `j` to represent *i*, but these alphabets are also used widely within computer science faculties for index variables. To avoid mistakes, the symbol `im` was chosen instead of symbols `i` or `j`.

The complex number 1 + 4*i* can thus be written as follows:

```
julia> a = 1+4im
1 + 4im

julia> real(a)
1

julia> imag(a)
4
```

The real part of the complex number is 1 and the imaginary part is 4*i*. Complex numbers in Julia are stored as two numbers, `re` (real part) and `im` (imaginary part). Both of these are some type of real number. They can be obtained using functions `real()` and `imag()`.

Another way to make a complex number object is by using `complex()`, `complex32()`, `complex64()`, and `complex128()` functions:

```
julia> a = 3.5 # defining "a"
3.5

julia> b = -4.9 # defining "b"
-4.9

julia> z = complex(a,b) # making complex number
# with a as real part and b as imaginary part
3.5 - 4.9im

julia> typeof(z)
Complex{Float64}

julia> typeof(real(z)) # Real part is stored as a
# Floati64 object
Float64

julia> typeof(imag(z)) # Imaginary part is stored as a
# Floati64 object
Float64
```

```
julia> z1 = Complex32(a,b) # Complex number with
# 32 bit storage
Float16(3.5) - Float16(4.9)im

julia> typeof(real(z1)) # z1 stores real part as
# Float16 object
Float16

julia> typeof(imag(z1)) # z1 stores imaginary part
# as Float16 object
Float16

julia> z2 = Complex64(a,b) # Complex number with
# 64 bit storage
3.5f0 - 4.9f0im

julia> typeof(real(z2)) # z2 stores real part as
# Float32 object
Float32

julia> typeof(imag(z2)) # z2 stores imaginary part as
# Float32 object
Float32

julia> z3 = Complex128(a,b) # Complex number with
# 128 bit storage
3.5 - 4.9im

julia> typeof(real(z2)) # z3 stores real part as
# Float64 object
Float64

julia> typeof(imag(z2))# z3 stores imaginary part as
# Float64 object
Float64
```

It is worth noting that data types of real and imaginary parts are retained as per definitions of defining functions and rules of conversions are similar to those used for integers and real numbers.

4.3 Properties of Complex Numbers

Complex numbers are graphically defined, as shown in Figure 4-1. On a real-imaginary axis based complex plane, a particular point is defined by a complex number $a + ib$ where a is the magnitude of the projection of the point on the real axis and b is the magnitude of the projection of the point on the imaginary axis.

Figure 4-1 shows a point depicting the complex number $z = x + iy$ and demonstrates how the value of $r = |z|$ (absolute value) and ϕ (argument) are given.

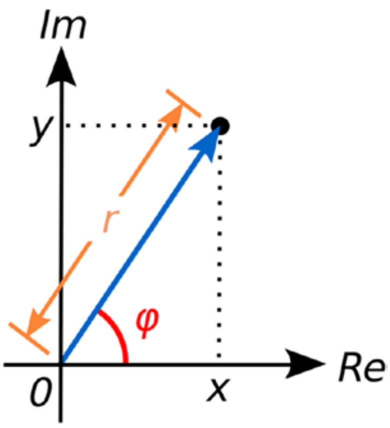


Figure 4-1. *Complex number depicted on a complex plane [1]*

$$r = \sqrt{x^2 + y^2} \tag{4.1}$$

$$\phi = tan^{-1}\left(\frac{y}{x}\right) \tag{4.2}$$

The absolute value of a complex number is simply its distance from the origin. The argument of a complex number is simply the angle it makes with the horizontal axis in an anticlockwise direction.

The principle and argument (in radians) for a complex number, say $z = -4+3i$, can be calculated using Julia:

```
julia> z = -4 + 3im
-4 + 3im

julia> r = abs(z)
5.0
```

```
julia> r_squared = abs2(z)
25

julia> phi = angle(z)
2.498091544796509

julia> z = -4 + 3im
-4 + 3im

julia> z_conjugate = conj(z)
-4 - 3im

julia> abs2(z) == z*z_conjugate
true
```

The conjugate of a complex number is its mirror image along the horizontal axis. In other words, its imaginary part is the negative of the original number. When squared with its conjugate, the result is r^2, which is verified by the last line in the previous code.

`Inf` and `NaN` propagate through complex numbers in the real and imaginary parts of a complex number. Let's work with three complex numbers: `z1 = complex(NaN,Inf)`, `z2 = complex(Inf,NaN)`, and a simple complex number `z3 = complex(1,2)`. Then, let's calculate `z1+z2, z2+z3`, and `z1+z3` to test how complex numbers with `NaN` and `Inf` are treated:

```
julia> z1 = complex(NaN,Inf)
NaN + Inf*im

julia> z2 = complex(Inf,NaN)
Inf + NaN*im

julia> z3 = complex(1,2)
1 + 2im

julia> z1+z3
NaN + Inf*im

julia> z2+z3
Inf + NaN*im

julia> z1+z2
NaN + NaN*im
```

4.4 Complex Arithmetic

Complex arithmetic involves similar operations as previously discussed for real numbers in Chapter 4. These operations include addition, subtraction, multiplication, division, raised to a power, and so on. However, rules for complex numbers are a bit different for these operations.

Adding two complex numbers involves adding their real and imaginary parts. This is also the case with subtraction. Suppose we define two complex numbers in the following manner:

$$z_1 = a_1 + b_1 i$$
$$z_2 = a_2 + b_2 i$$

Now we can define their addition and subtraction as follows:

$$z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$$
$$z_1 - z_2 = (a_1 + a_2) - (b_1 + b_2)i$$

Multiplication and division operations for complex numbers are not so straightforward:

$$z_1 \times z_2 = (a_1 \times a_2) + (a_1 \times b_2)i + (a_2 \times b_2) + (b_1 \times b_2)\left(i^2\right)$$

This simplifies by collecting real terms and imaginary terms because $i^2 = -1$:

$$z_1 \times z_2 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i$$

Multiplying and dividing a complex number with a real number can be done in a simpler manner by simply performing the multiplication or division for the real and imaginary parts respectively.

A complex conjugate of a complex number $z_1 = a_1 + b_1 i$ is defined as $z_1^* = a_1 - b_1 i$. Geometrically, z_1^* is the "reflection" of z_1 about the real axis. Hence, if we calculate the conjugate twice, we get the same number: $\left(z_1^*\right)^* = z_1$.

Division of a complex number can be performed using its conjugate as follows:

$$\frac{a_1 + b_1 i}{a_2 + b_2 i} = \frac{a_1 + b_1 i}{a_2 + b_2 i} \times \frac{a_2 - b_2 i}{a_2 - b_2 i} = \frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2} + \frac{b_1 a_2 - a_1 b_2}{a_2^2 + b_2^2} i$$

As a result, multiplying the denominator's complex conjugate with both numerator and denominator yields a new complex number that is the result of division of two complex numbers.

Julian operators, such as `+`,`-`,`*`,`^`, and `/`, work well with real numbers as well as with complex numbers without any additional effort. For example:

```
julia> a = 3.2
3.2

julia> b = -4.4
-4.4

julia> z1 = complex(a,b)
3.2 - 4.4im

julia> z2 = -z1
-3.2  +  4.4im

julia> z3 = z1 + z2 # Adding two complex numbers
0.0 + 0.0im

julia> z3 = z1 - z2 # Subtracting two complex numbers
6.4 - 8.8im

julia> z3 = z1 * z2 # Multiplying two complex numbers
9.120000000000001 + 28.160000000000004im

julia> z3 = z1 / z2 # Dividing two complex numbers
-1.0 - 0.0im

julia> z3 = z1^z2 # complex number raised to the power
# of another complex number
-0.1407063188343073 - 0.24121298541124633im

julia> z3 = z1^2 # complex number raised to the power
# of an real number
-9.120000000000001 - 28.160000000000004im
```

```
julia> z3 = 2*z1 # A real number multiplied with
# a compelx number
6.4 - 8.8im

julia> z3  =  z1*(z2^3.2  - z2/z1)
1211.337685375584 - 232.35311358949485im
```

When involved numbers are of mixed data types, hierarchy laws are followed:

```
julia> a = Int8(9)
9

julia> b = Float64(287.876567)
287.876567

julia> z = complex(a,b)
9.0 + 287.876567im

julia> typeof(real(z))
Float64

julia> typeof(imag(z))
Float64

julia> c = Int64(3456)
3456

julia> z1 = c*(z)
31104.0 + 994901.415552im

julia> typeof(imag(z1))
Float64

julia> typeof(real(z))
Float64
```

4.5 Summary

The ability to deal with complex numbers and their arithmetic allows Julia to enter into the real-world simulation in a realistic manner. Complex analysis is one of the cornerstones of mathematical studies in physical and engineering science. The physical importance of real and imaginary parts is critical to scientific interpretation, especially with time-varying phenomena (in general, dynamical problems). The ease of defining complex numbers coupled with the ease of extracting real and complex parts as well as operators for performing complex analysis makes Julia one of the most advanced options for simulating real-life problems.

4.6 Bibliography

[1] https://en.wikipedia.org/wiki/File:Complex_number_illustration_modarg.svg

CHAPTER 5

Rational and Irrational Numbers

5.1 Numbers and Ratios

A rational number is a number that can be written as a ratio of two numbers. This ratio is also called a *fraction representation*. A fraction representation includes two parts: the numerator (the number on top) and the denominator (the number on the bottom). The following is a fraction representation of a rational number:

$$\frac{a}{b}$$

where $a, b \in I$ and $b \neq 0$.

For example, 0.50 can also be written as $\frac{1}{2}$. Similarly 0.60 can be written as $\frac{6}{10}$, which can then be reduced to $\frac{3}{5}$. Rational numbers can be formally defined as *equivalence classes* of pairs of integers (p, q) such that $q \neq 0$, for the *equivalence relation* defined by $(p_1, q_1) \sim (p_2, q_2)$ if $p_1q_2 = p_2q_1$. This is simple to follow since

$$\frac{p_1}{q_1} = \frac{p_2}{q_2} \tag{5.1}$$

implies

$$p_1q_1 = p_2q_2 \tag{5.2}$$

The decimal expansion of a rational number always either terminates after a finite number of digits or begins to repeat the same finite sequence of digits over and over. It is important to note that the discussion is not only true for numbers with base 10 (decimal numbers), but it is also true with any other base such as 2 (binary numbers), 6 (hexadecimal numbers), and 8 (octadecimal numbers), and so on.

All the numbers that cannot be expressed as a ratio are called irrational numbers or, in other words, a real number that is not rational is called irrational, such as the number $\pi = 3.1415926535897\ldots$ and the number $e = e = 2.7182818284590\ldots$ One can argue that the ratio $\frac{22}{7}$ can be written for π, but the ratio is an inexact representation of the actual number. The decimal expansion of an irrational number continues without repeating:

```
julia> a = pi
a = 3.1415926535897...

julia> e
e = 2.7182818284590...

julia> inexactness = 22/7 - pi
inexactness = 0.0012644892673496777
```

5.2 Rational Numbers

Rational numbers represent exact ratios of numbers. For example, $\frac{2}{3}$ can be valued as 0.66… . The number used must be restricted to a finite number of digits that will induce errors in calculations due to the inexact representation as a fractional number of a ratio of integers. If, on the other hand, the ratio itself can be used for calculations, the exactness of the calculation can be preserved.

5.2.1 Representation of Rational Numbers

The operator `//` is used to define a rational number. For example, the rational number $\frac{2}{3}$ can be defined as follows:

```
julia> a = 2//3
2//3

julia> typeof(a)
Rational{Int64}
```

The numerator and denominator can be extracted from a rational number using `num()` and `den()` functions:

```
julia> a = 2
2

julia> b = -4
-4

julia> a1 = a//b
-1//2

julia> num(a1)
-1

julia> den(a1)
2
```

This example also outlines an important fact. The `//` evaluates the rational number by solving the rational number—that is, factorizing the numerator and denominator and then canceling common factors:

$$\frac{2}{-4}=\frac{2\times 1}{-1\times 2\times 2}=\frac{-1}{2}$$

5.2.2 Complex Numbers as Numerators and Denominators

Rational numbers can be constructed using complex numbers as follows:

```
julia> a = complex(2,3)
2 + 3im

julia> b = complex(-3,2)
-3 + 2im

julia> 1//a
2//13 - 3//13*im

julia> -3//b
9//13 + 6//13*im
```

```
julia> a//b
0//1 - 1//1*im

julia> (-3//b)^a
0.09624621106941285 + 0.06935501144252361im
```

When `a = complex(2,3)` and `b = complex(-3,2)`, then `1//a` (Equation 5.3) and `a//b` (Equation 5.4) can be calculated mathematically as follows:

$$\frac{1}{2+3i}=\frac{1}{2+3i}\times\frac{2-3i}{2-3i}=\frac{2-3i}{4+9}=\frac{2}{13}-\frac{-3}{13}i \tag{5.3}$$

Similarly, for the expression `a//b`, you can construct a rational number made of two complex numbers as $\frac{2+3\mathrm{i}}{-3+2i}$, which can be solved as follows:

$$\frac{2+3\mathrm{i}}{-3+2i}=\frac{2+3\mathrm{i}}{-3+2i}\times\frac{-3-2\mathrm{i}}{-3-2i}=\frac{-6-4\mathrm{i}-9i+6}{9+4}+\frac{0}{13}+\frac{1}{1}i \tag{5.4}$$

5.2.3 Mathematical Operations on Rational Numbers

A Julian rational number can be operated upon just like a mathematical one:

$$\frac{1}{2}+\frac{1}{2}=\frac{1}{1}=1$$
$$\frac{1}{2}-\frac{1}{2}=\frac{0}{1}=0$$
$$\frac{1}{2}/\frac{1}{2}=\frac{1}{1}=1$$
$$\frac{1}{2}*\frac{1}{2}=\frac{1}{4}$$

```
julia> a = 1//2
1//2

julia> a+a
1//1

julia> a-a
0//1
```

```
julia> a/a
1//1

julia> a*a
1//4
```

When the numerator and/or denominator are negative, the number is converted with an appropriate sign:

```
julia> a = -3
-3

julia> b = 4
4

julia> c = -2
-2

julia> a1 = a//b
-3//4

julia> a2 = a//c
3//2
```

Comparison operators can also be used on rational data types:

```
julia> a = -2
-2

julia> b = 3
3

julia> a1 = a//b
-2//3

julia> a2 = (2*a)//(2*b)
-2//3

julia> a1 == a2
true
```

```
julia> a3 = (2*a)//(3*b)
-4//9

julia> a1 < a3
true

julia> ((a1 + a2) < a3) & (a2 == a3)
false

julia> a1+a2
-4//3

julia> (a1+a2)<a3
true

julia> a2 == a3
false

julia> true & false
false
```

5.2.4 Converting a Rational Number to a Floating Point Number

A rational number can be converted to a decimal point representation by dividing the numerator by the denominator and writing the quotient. For example, $\frac{2}{5} = 0.4$. Sometimes this representation results in an infinitely recurring set of digits. For example, $\frac{1}{3} = 0.333\ldots$. Julia supports these mathematical calculations as follows:

```
julia> float(2//5)
0.4

julia> float(1//3)
0.3333333333333333
```

5.2.5 Rationals with Zero Denominator

Rational numbers are defined for nonzero denominators, but Julia allows zero denominators and even allows usual computation. In other words, constructing infinite rational values is acceptable in Julia. However, the construction of a fraction $\frac{0}{0}$ is not allowed:

```
julia> a = -1
-1

julia> b = 0
0

julia> c = 3
3

julia> a1 = a//b
-1//0

julia> 0//0
ERROR: ArgumentError: invalid
rational: zero(Int64)//zero(Int64)
Stacktrace:
[1] Rational{Int64}(::Int64, ::Int64)
at ./rational.jl:13
[2] //(::Int64, ::Int64)
at ./rational.jl:40
```

5.2.6 Rationals with Other Data Types

Rational data types interact with other data types as per promotion rules for data type. In the following example, `a1` stores a rational number with `Int64` numerator and denominator, and `a2` stores a complex number. When `a1+a2` is calculated, the complex number is `c = obtained`, whose real and imaginary parts are of the type `Rational`. The same is true for the division operation.

```
julia> a1 = 2//3
2//3

julia> a1+1
5//3

julia> a1 + 2.3
2.9666666666666663

julia> a2 = complex(-2,4)
-2 + 4im

julia> a1+a2
-4//3 + 4//1*im

julia> a1/a2
-1//15 - 2//15*im
```

Now let's test how exact and inexact rations can be compared using boolean operations where $\frac{1}{2} = 0.5, \frac{1}{3} \neq 0.333$. This inexactness can be calculated by performing `(1//3)-0.33`. But when compared to `float(1//3)` (that is, the rational number is represented as a floating point number and then subtracted by `1//3`), we obtain a zero. This indicates that before calculations between rational numbers and integers or floating point numbers occur, rational numbers are converted into `Float64` data type.

```
julia> 0.5 == 1//2
true

julia> 0.3 == 1//3
false

julia> 0.333 == 1//3
false

julia> a = float(1//3)
0.3333333333333333

julia> a == 1//3
false
```

```
julia> (1//3) - a
0.0

julia> (1//3)-0.333
0.0003333333333332966

julia> (1//3)-a # a = float(1//3)
0.0
```

5.3 Irrational Numbers

Irrational numbers are simply those numbers that are not rational; they cannot be written as a ratio of two whole numbers. There are many examples such as *π* and *e*. Julia defines a data type aptly named `Irrational`. For example, *π* and *e* are predefined as irrational constants in Julia:

```
julia> pi
pi = 3.1415926535897...

julia>  e
e = 2.7182818284590...

julia> typeof(pi)
Irrational{:pi}

julia> typeof(e)
Irrational{:e}
```

Notice that the numeric representation of an irrational number ends with three dots, highlighting the fact that the digital representation does not end here but, in fact, continues.

5.4 Summary

The ability to define and work with fractions was one of the cornerstones of Greek mathematics. Most children learn how to work with rational numbers and their arithmetic at a young age and the ease of providing exact solutions as fractions is well-known. The ability to define rational numbers within a programming language makes it quite suitable for numerical computing where exactness of a solution is critical. Defining irrational numbers is equally important since irrational numbers, when introduced into a computation, leads to inexactness and, thus, errors. The degree of inexactness depends on the precision of the representation of an irrational number. Julia's ability to define a set precision of irrational numbers allows us to determine the degree of inexactness in a numerical computation beforehand, as this chapter has demontrated.

CHAPTER 6

Mathematical Functions

6.1 Introduction

A mathematical function is a relation between a set of inputs and a set of permissible outputs with the property such that each input is related to exactly one output. Most users are already familiar with many such functions including the trigonometric functions `sin(x)`, `cos(x)`, and `tan(x)`; logarithms to the base of a number—say 10—as $\log_{10}(x)$; exponentiation e^x; and so on. A programming language boasting to perform complex mathematical calculations in an efficient manner must provide easy and intuitive ways to interact with such mathematical functions and must also provide ways to construct user-defined functions. Present chapter will illustrate some in-built mathematical functions within Julia.

6.2 Division Functions

Division is one of the four basic arithmetic operations; the other three are addition, subtraction, and multiplication. The division of two natural numbers is the process of calculating the number of times one number is contained within the other. This is essentially counting the number of groups we can make within the second number. For example, when 10 (divisor) is divided by 3 (dividend), we can make 3 (quotient) groups and 1 (remainder) remains. Division can also be described as the cycling of one number over another until we find the end. The cycling can be linear, polar, and so on. Cycling is depicted by the *modulo* function where *a modulo b* means that *a* is cycled *b* times to find the number of cycles (quotient) and what remains (remainder).

A set of Julia functions is defined for performing the division of one or more numbers in a specified manner. Table 6-1 outlines their syntax and behavior. Now let's scan their usage with the help of examples.

Table 6-1. *Division Functions*

Syntax	Behavior
`div(x,y)`	truncated division; quotient rounded toward zero
`fld(x,y)`	floored division; quotient rounded toward `-Inf`
`cld(x,y)`	ceiling division; quotient rounded toward `+Inf`
`rem(x,y)`	remainder; satisfies `x == div(x,y) *y + rem(x,y)`; sign matches `x`
`mod(x,y)`	modulus; satisfies `x == fld(x,y) *y + rem(x,y)`; sign matches `x`
`mod1(x,y)`	`mod()` with offset 1
`mod2pi(x,y)`	modulus with respect to 2π
`divrem(x,y)`	returns `(div(x,y),rem(x,y))`
`fldmod(x,y)`	returns `(fld(x,y),mod(x,y))`
`gcd(x,y, ...)`	greatest positive common divisor of `x, y,...`
`lcm(x,y, ...)`	least common multiple of `x, y,...`

6.2.1 div(x,y), fld(x,y), and cld(x,y)

Suppose we wish to perform $\frac{3}{5}$. The built-in functions `div(x,y), fld(x,y)`, and `cld(x,y)` produce the following results:

```
julia> div(3,5) # simple division
0

julia> fld(3,5) # floor division
0

julia> cld(3,5) # ceil division
1

julia> 3/5
0.6
```

We know that $\frac{3}{5}$ produces 0.6 as quotient and 0 as remainder. Hence, for three functions, the following behavior is observed:

- For `div()`, the result is truncated toward 0 so 0.6 becomes 0.
- For fld`()`, the result is truncated toward $-Inf$ so 0.6 becomes 0.
- For `cld()`, the result is truncated toward $+inf$ so 0.6 becomes 1.

Different kinds of data types can be used with these functions. Using `Float64` gives the following results:

```
julia> 2.24/3.45
0.6492753623188406

julia> div(2.24,3.45)
0.0

julia> cld(2.24,3.45)
1.0

julia> fld(2.24,3.45)
0.0
```

When you use mixed data types, the rules of conversion and promotions are applied. You can use `methods(f)` to scan all the combinations of data types that will be entertained by the function `f`. Thus, it is useful to scan `methods(div)`, `methods(fld)`, and `methods(cld)`. You can check that complex numbers cannot be used as inputs to these functions:

```
julia> a = Float64(2.24)
2.24

julia> b = Int64(3)
3

julia> r1 = div(a,b)
0.0

julia> typeof(r1)
Float64
```

```
julia> r2 = fld(a,b)
0.0

julia> r3 = cld(a,b)
1.0
```

6.2.2 rem(), mod(), and mod1()

In addition to getting quotients by using functions `div()`, `fld()`, and `fld()`, you can obtain the remainder by using the `rem()` function. Suppose we wish to check for $\frac{2}{3}$. We know that the quotient is 1 and the remainder is 1:

```
julia> a = 3
3

julia> b = -2
-2

julia> div(a,b) # Quotient
-1

julia> rem(a,b) # Remainder, sign matches a
1

julia> mod(a,b) # Modulo, sign matches b
-1

julia> mod1(a,b) # Moulo with offset 1
1
```

In computing, the `modulo` operation finds the remainder after the division of one number by another number (sometimes called "modulus"). The function `mod(2,3)` performs this task and returns 1 as a result.

6.2.3 mod2pi()

Just like the modulus functions cycle w.r.t real numbers, `mod2pi()`finds the remainder by cycling over 2π. As a result, the value is always within the limits [0, 2π]:

```
julia> pi
pi= 3.1415926535897...

julia> mod2pi(1*pi)
3.141592653589793

julia> typeof(mod2pi(1*pi))
Float64

julia> typeof(1*pi)
Float64

julia> mod2pi(-1*pi)
3.1415926535897936

julia> mod2pi(2*pi)
6.283185307179586

julia> mod2pi(2*pi+1)
0.9999999999999998

julia> mod2pi(2*pi-1)
5.283185307179586

julia> mod2pi(1*pi-1)
2.141592653589793
```

It is worth noting that the `mod2pi()` function does not have `Irrational` data type as one of its methods so `mod2pi(pi)` results in an error message. On the other hand, when the irrational number `pi` is operated upon, it becomes another data type, which can then be used in the `mod2pi()` function.

6.2.4 divrem() and fldmod()

Instead of using `div()` and `rem()` functions separately, we can compute them within a single statement using the `divrem()` function:

```
julia> quotient,remainder = divrem(10,3)
(3,1)

julia> quotient,remainder = divrem(10,3.3)
(3.0,0.10000000000000053)

julia> quotient,remainder = fldmod(10,3)
(3,1)

julia> quotient,remainder = fldmod(10,3.3)
(3.0,0.10000000000000053)
```

6.2.5 gcd()

The greatest common divisor is the biggest number that divides all the elements of a set of numbers. `gcd(x,y)` gives the greatest common (positive) divisor (or zero if `x` and `y` are both zero):

```
julia> gcd(3,6)
3

julia> gcd(319,666)
1

julia> gcd(-319,666)
1

julia> gcd(-319,-666)
1
```

6.2.6 lcm()

The least common multiple is the smallest positive number that occurs in a list of multiples for a set of numbers. Julia function `lcm` gives the least common (non-negative) multiple:

```
julia> lcm(40,55)
440

julia> lcm(33,11)
33

julia> lcm(33,-11)
33

julia> lcm(-33,-11)
33
```

6.3 Sign and Absolute Value Functions

The sign of a number seems a small, insignificant property but proves to be a valuable tool. A mere change of sign changes the quadrant in which a number is defined. Oppositely signed numbers are mirror images of the original numbers in different quadrants. Julia provides a set of functions to derive the information about the sign of a number (Table 6-2).

Table 6-2. *Julia Functions for Sign of a Number*

Syntax	Behavior
`abs(x)`	a positive value with the magnitude of x
`abs2(x)`	the squared magnitude of x
`sign(x)`	indication of the sign of x, returning −1, 0, or +1
`signbit(x)`	indication whether the sign bit is on (true) or off (false)
`copysign(x,y)`	a value with the magnitude of x and the sign of y
`flipsign(x,y)`	a value with the magnitude of x and the sign of x*y

6.3.1 abs() and abs2()

The absolute value of a number is the positive value of a number. Now there are two ways to obtain a positive value with a computer. The first is to just flip the sign bit used while defining a signed integer. The second is to square the number and then find the square root of that number. The problem with the second method is that you may incur approximation errors while performing these operations in some cases:

```
julia> a = -900707.7097680866 # Floating point number
-900707.7097680866

julia> sq = a*a
8.112743784356716e11

julia> sqroot = sqrt(sq)
900707.7097680866

julia> -sqroot == a
true

julia> a = -pi # Irrational number
-3.141592653589793

julia> sq = a*a
9.869604401089358

julia> sqroot = sqrt(sq)
3.141592653589793

julia> -sqroot == a
true

julia> a = -(2//3) # Rational number
-2//3

julia> sq = a*a
4//9
```

```
julia> sqroot = sqrt(sq)
0.6666666666666666

julia> -sqroot == float(a)
true
```

Julia provides the `abs(x)` function to know the absolute value of a number. `methods(abs)` gives the options about various data types that can be used with the `abs()` function:

```
julia> abs(-190)
190

julia> abs(190)
190

julia> abs(-190.08967)
190.08967

julia> abs(190.08967)
190.08967
```

6.3.2 Absolute Value of a Complex Number

As discussed in Chapter 4, the absolute value of a complex number $a + ib$ is

$$\sqrt{a^2 + b^2}$$

length of vector (defined by a complex number) from the origin.

```
julia> a = complex(2,-3)
2 - 3im

julia> abs(a)
3.6055512754639896

julia> abs2(a)
13
```

The squared absolute value gives

$$a^2 + b^2$$

using the function abs2().

Problem with abs() Function

When abs is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when abs is applied to the minimum representable value of a signed integer.

```
julia> a = typemin(Int8)
-128

julia> abs(a)
-128

julia> a = typemin(Int32)
-2147483648

julia> abs(a)
-2147483648
```

6.3.3 sign(), signbit(), copysign(), and flipsign()

As described in by Section 3.5.2, when signed numbers are stored, a sign bit is reserved for assigning the sign to a number. This bit can be scanned with the signbit() function, which returns true if the value of the sign of x is negative. Otherwise, it returns false:

```
julia> a = -123 # Negative integer
-123

julia> signbit(a)
true

julia> b = 123 # Positive integer
123
```

```
julia> signbit(b)
false

julia> c = 0 # Zero
0

julia> signbit(c)
false

julia> a1 = -123.123 # Negative float
-123.123

julia> signbit(a1)
true

julia> b1 = 123.123 # Positive float
123.123

julia> signbit(b1)
false

julia> c1 = 0.0 # Zero float
0.0

julia> signbit(c1)
false

julia> a2 = -2//3 # Negative rational
-2//3

julia> signbit(a2)
true

julia> a3 = 2//3 # Positive rational
2//3

julia> signbit(a3)
false

julia> a4 = -2//-3 # Positive rational
2//3
```

```
julia> signbit(a4)
false

julia> a5 = 0//1 # Zero rational
0//1

julia> signbit(a5)
false
```

To know the sign of a number, the `sign()` function is used. It returns 0 if the input number is zero, 1 if number is positive and -1 if input number is negative.

```
julia> a = -123
-123

julia> sign(a)
-1

julia> b = 123
123

julia> sign(b)
1

julia> c = 0
0

julia> sign(c)
0
```

Manipulating Signs

Sometimes, you need to assign a chosen sign to a number. One might need to scan a particular number and assign the sign of the chosen number to a new number. This is done by the function `copysign(x,y)`. It returns `x` such that it has the same sign as y. `methods(copysign)` gives the list of data types that can be fed to this function:

```
julia> copysign(123,-231)
-123
```

```
julia> copysign(-123,231)
123

julia> copysign(-123,-231)
-123

julia> copysign(123,231)
123

julia> copysign(123,0)
123

julia> copysign(-123,0)
123
```

`flipsign(x,y)` function returns `x` with its sign flipped if `y` is negative.

```
julia> flipsign(123,-231) # Sign flipped
-123

julia> flipsign(-123,231) # Sign unchanged
-123

julia> flipsign(-123,-231) # Sign flipped
123

julia> flipsign(123,231) # Sign unchanged
123

julia> flipsign(123,0) # Sign unchanged
123

julia> flipsign(-123,0) # Sign unchanged
-123
```

6.4 Power, Logs, and Roots

Raising a number to a power essentially signifies the number of times the power is multiplied with itself. (For example, a^n means a is multiplied n times.) It is interesting to note that n can be any real number. Equations define the rules of calculations:

$$a^n \times a^m = a^{n+m} \tag{6.1}$$

$$\frac{a^n}{a^m} = a^{n-m} \tag{6.2}$$

$$a^0 = 1 \tag{6.3}$$

$$a^{-n} = \frac{1}{a^n} \tag{6.4}$$

$$a^n \times a^{-n} = a^0 = 1 \tag{6.5}$$

$$(a \times b)^n = a^n \times b^n \tag{6.6}$$

When n is a rational number with the form $\frac{1}{n}$, then $a^{\frac{1}{n}}$ is called the n^{th} root of a.

$$a^{\frac{1}{n}} = b \Rightarrow a = b^n \tag{6.7}$$

$$a^{\frac{n}{m}} = \sqrt[m]{a^n} \tag{6.8}$$

Most interesting is the complex number i:

$$i^2 = -1 \tag{6.9}$$

$$i^3 = -1 \times i = -i \tag{6.10}$$

$$i^4 = -i \times i = -(-1) = 1 \tag{6.11}$$

$$i^5 = 1 \times i = i \tag{6.12}$$

These are called the *complex roots of unity.*

6.4.1 Numbers Raised to Some Power

When a number is powered to another number, a range of mathematical functions are used in this regard. Users must check Equation 6.13 for defining the power of a number, 6.14 for defining the root of a number, and 6.15 for defining the logarithm of a number with a chosen base:

$$a = b^c \tag{6.13}$$

$$b = \sqrt[c]{a} \tag{6.14}$$

$$log_b a = c \tag{6.15}$$

Let's understand these equations with a simple example:

$$1000 = 10^3$$
$$10 = \sqrt[3]{1000}$$
$$log_{10} 1000 = 3$$

The facility to define powers, roots, and logarithms to a chosen based is provided by a range of Julia functions, as given in Table 6-3.

Table 6-3. *Julia Functions for Power, Roots, and Logarithm Calculations*

Syntax	Behavior
`sqrt(x)`	calculates $\sqrt{x}$
`cbrt{x}`	calculates $\sqrt[3]{x}$
`hypot(x,y)`	calculates $\sqrt{x^2 + y^2}$ (i.e., hypotenuse of a right-angled triangle with sides as *x* and *y*)
`exp(x)`	calculates e^x
`expm1(x)`	calculates e^x – 1 accurate for *x* near 0
`ldexp(x,n)`	calculates x* 2^n computed efficiently for integer values of n
`log(x)`	calculates $log_e(x)$
`log(b,x)`	calculates $log_b(x)$

(*continued*)

Table 6-3. (*continued*)

Syntax	Behavior
`log2(x)`	calculates $log_2(x)$
`log10(x)`	calculates $log_{10}(x)$
`log1p(x)`	calculates $log(1 + x)$ accurate for x near zero
`exponent(x)`	binary exponent of x
`significand(x)`	binary significand (also known as *mantissa*) of a floating point number x

The following sections will scan them one by one for understanding their proper usage. Sample Julia code is given after each function's description so that you can test the usage of the function.

6.4.2 sqrt(), cbrt(), and hypot()

The square root of a number x is a number raised to the power $\frac{1}{2}$, which is symbolically shown by $\sqrt{x}$. Similarly, a cube root of a number is a number raised to the power $\frac{1}{3}$, which is symbolically shown by $\sqrt[3]{x}$.

```
julia> sqrt(2)
1.4142135623730951

julia> cbrt(2)
1.2599210498948732
```

`methods(sqrt)` and `methods(cbrt)` outline various data types that can be used with these functions. Particularly exciting is the number $i = \sqrt{-1}$. The command `sqrt(-1)` will produce a domain error message saying that a complex argument is needed. Hence, the following code should be issued instead:

```
julia> a = complex(0,1)# complex number "i"
0 + 1im

julia> sq = a^2
-1 + 0im
```

```
julia> sqrt(sq)
0.0 + 1.0im
```

The function `cbrt()` does not entertain complex data types, but it does entertains negative numbers:

```
julia> cbrt(pi) # Irrational number
1.4645918875615231

julia> cbrt(-3.54) # Negative number
-1.5240565695688593

julia> cbrt(3.54) # Positive number
1.5240565695688593

julia> cbrt(2//3) # Rational number
0.8735804647362988

julia> cbrt(float(2//3))
0.8735804647362988
```

The third function, `hypot(x,y)`, is used to calculate the hypotenuse of a triangle made by a right triangle. This is also the length of a 2D vector defined by a complex number $a + ib$:

$$\sqrt{a^2 + b^2}$$

```
julia> hypot(2,3) # x=2,y=3
3.6055512754639896
```

```
julia> sqrt(2^2 + 3^2) # same calcualtion as done by hypot()
3.605551275463989

julia> hypot(2//3,3//4) # Rational numbers
1.0034662148993578

julia> hypot(complex(2,3),complex(3,4)) # Complex nos.
6.164414002968977
```

6.4.3 Problem with hypot() Calculations

John D. Cook outlines a problem with `hypot()` in an interesting article [2]. If `x` is so large that `x*x` overflows, the code will produce an infinite result. To avoid this problem, it is suggested Cook suggests that the algorithm takes another route to calculate the hypotenuse.

Without risking overflow, $\sqrt{x^2 + y^2}$ can be calculated as follows:

1. $max = maximum(|x|, |y|)$
2. $min = minimum(|x|, |y|)$
3. $r = \frac{\max}{\min}$
4. $ans = \max \times \sqrt{1 + r^2}$

Since step 4 includes the square root argument, which inputs *max* and *min* values, you can avoid overflow errors. The data type `Float64` includes the maximum numeric value as 10^{308} (in other words, `1e308`).

```
julia> 1e308
1.0e308

julia> 1e308*10 # Multiplication with 10
Inf
```

Now consider that x and y are 0.5 times 10^{308} when calculating the hypotenuse:

```
julia> x = 1e308
1.0e308

julia> y = 1e308
1.0e308

julia> h =  hypot(x,y)
1.4142135623730951e308

julia> x^2 # Overflow
Inf
```

```
julia> y^2 # Overflow
Inf

julia> h1 = sqrt(x^2 + y^2) # wrong result
Inf
```

6.4.4 exp(), expm1(), ldexp(), and exponent()

Exponentiation is raising a number to the power of *e. e* is Euler's number (irrational) defined to be valued as 2.7182818284590…

```
julia> e
e   =   2.7182818284590...
```

exp()

The function `exp(x)` outputs e^x. It is important to note that the use of exponential function without proper care will incur overflow (e^x) and/or underflow (e^{-x}) problems with the result as `+Inf` and `-Inf`:

```
julia> exp(1)
2.718281828459045

julia> exp(2)
7.38905609893065

julia> exp(-1)
0.36787944117144233

julia> exp(-2)
0.1353352832366127

julia> exp(1//2)
1.6487212707001282
```

It is worth noting that while working with `exp()` and similar functions, an approximation of number `e` is used. In other words, *e* is a never-ending irrational number, but e^1 is calculated with finite precision as a floating point number and, hence, it is finite in nature. Consequently, `exp(1) == e` results as `false`.

```
julia> e
e = 2.7182818284590...

julia> exp(1)
2.718281828459045

julia> exp(1) == e
false
```

expm1()

The problem with the `exp(x)` function occurs when we wish to calculate `exp(1+x)` [3] and the value of `x` is comparable to machine precision (that is, extremely small). In this case, `1+x` is approximated as `x`. To overcome this issue, `exp1(x)` is used to calculate `exp(1+x)` cases where x is very small:

```
julia> exp(1)
2.718281828459045

julia> exp(1+1e-100)
2.718281828459045

julia> expm1(1+1e-100)
1.718281828459045
```

ldexp()

`ldexp(x,n)` uses a base 2 exponentiation and computes $x \times 2^n$:

```
julia> ldexp(3.5,2)
14.0

julia> 3.5*(2^2)
14.0
```

exponent(x)

`exponent(x)` returns the *x* for 2^y which is closest to *x* rounded toward zero:

```
julia> exponent(100.0)
6

julia> exponent(1000.0)
9

julia> 2^6
64

julia> 2^9
512

julia> exponent(128.0)
7

julia> 2^7
128
```

6.4.5 log(), log2(), log10(), and log1p()

The logarithm of a number is defined as follows:

$$a = b^c \Rightarrow log_b a = c \tag{6.16}$$

This example uses Equation 6.16:

$$10^2 = 100 \Rightarrow log_{10}(100) = 2$$

This examples also uses Equation 6.16, but with a different base, 2:

$$2^7 = 128 \Rightarrow log_2(128) = 7$$

The logarithm is an essential function in mathematics, particularly for those quantities that rise or fall very fast. In such cases, it’s useful to analyze them on a logarithmic scale rather than on a linear scale.

log()

Natural logarithms (for example, base e logarithm) are calculated using the function `log()`:

$$log_e(e)=1$$

This can be verified:

```
julia> e # Irrational number
e = 2.7182818284590...

julia> float(e) # Floating point number for e
2.718281828459045

julia> e^1 # e^1=e
2.718281828459045

julia> e^1 == float(e)
true

julia> log(e) $ log(e) to the base e is 1
1
```

Some physical properties show an exponential increase. The numerical values of such data points become very big numbers very quickly. When the `log` function is operated on such functions, you obtain smaller numbers. This is particularly important to avoid overflow and underflow errors.

log2()

`log2()` simply calculates log_2 for a number. The base is 2 instead of *e* here. Hence $log_2(2) = 1$ and $log_2 4 = 2$ because $2^1 = 2$ and $2^2 = 4$. Julia code can easily verify the same:

```
julia> log2(2)
1.0

julia> log2(4)
2.0
```

```
julia> 2^1
2

julia> 2^2
4
```

It is important to note that when you use a negative real exponent, you encounter a DomainError. To avoid this problem, use floating points number representation for the exponent.

```
julia> 2^-1.0
0.5

julia> log2(0.5)
-1.0
```

Rational numbers can also be used in the same manner:

```
julia> 2^(2//3)
1.5874010519681994

julia> log2(1.5874010519681994)
0.6666666666666665

julia> float(2//3)
0.6666666666666666
```

log10()

`log10()` calculates the $log_{10}(x)$ (for example, logarithm with base 10). Since $10^1 = 10$ and $10^2 = 100$,

$$log_{10}(10) = 1$$

and

$$log_{10}(100) = 2$$

This can be verified easily with Julia code:

```
julia> log10(10)
1.0

julia> log10(100)
2.0

julia> log10(105.4) # Floating point number
2.022840610876528

julia> log10(2//3) # Rational number
-0.17609125905568127

julia> log10(pi) # Irrational number pi
0.49714987269413385

julia> log10(e) # Irrational number e
0.4342944819032518

julia> log10(complex(2,3)) # Complex numbers
0.5569716761534184 + 0.42682189085546657im
```

log1p()

As we discussed earlier in Section 6.4.4 for the case of calculating e^{1+x} when x is a small number, you would encounter problems while calculating $log_e(1 + x)$ as well [3]. To overcome this issue, the function `log1p()` is proposed:

```
julia> log(1e10)
23.025850929940457

julia> log(1e10+1e-10) # Not different output
23.025850929940457

julia> log1p(1e10+1e-10) # Different output
23.025850930040455
```

6.5 Trigonometric and Hyperbolic Functions

6.5.1 Trigonometric Functions

Trigonometric functions relate angles of a right-angled triangle to the length of its sides. In Table 6-4, *P*, *H*, and *B* represent *perpendicular*, hypotenuse, and *base*; and various trigonometric functions are defined. The input argument to these functions is the angle in units of radians. Angles in radians (*r*) can be converted to angle in degrees (*d*) using the formula.

Table 6-4.. *Trigonometric Functions*

Function	Abrv.	Julia Function	Formula	Identity
sine	sin	`sin`	$\frac{P}{H}$	$sin(\theta) = cos\left(\frac{\pi}{2} - \theta\right) = \frac{1}{csc(\theta)}$
cosecant	csc	`csc`	$\frac{H}{P}$	$csc(\theta) = sec\left(\frac{\pi}{2} - \theta\right) = \frac{1}{sin(\theta)}$
cosine	cos	`cos`	$\frac{B}{H}$	$cos(\theta) = sin\left(\frac{\pi}{2} - \theta\right) = \frac{1}{sec(\theta)}$
secant	sec	`sec`	$\frac{H}{B}$	$sec(\theta) = csc\left(\frac{\pi}{2} - \theta\right) = \frac{1}{cos(\theta)}$
tangent	tan	`tan`	$\frac{P}{B}$	$tan(\theta) = cot\left(\frac{\pi}{2} - \theta\right) = \frac{1}{cot(\theta)} = \frac{sin(\theta)}{cos(\theta)}$
cotangent	cot	`cot`	$\frac{B}{P}$	$cot(\theta) = tan\left(\frac{\pi}{2} - \theta\right) = \frac{1}{tan(\theta)} = \frac{cos(\theta)}{sin(\theta)}$

$$d = \frac{r \times 180}{\pi} \tag{6.17}$$

From Equation 6.17, it can be easily deduced that 1 radian ≈ 57.3°.

```
julia> 180/pi
57.29577951308232
```

Since *sin*(90° = 1), while working in radians, you first convert 90° into radians and then feed it to the `sin` function:

```
julia> r =(90*pi)/180 # Convert 90 degree into radians
1.5707963267948966

julia> sin(r) # sin(90)=1
1.0
```

Some example Julia code will outline their usage. Let's start by feeding known values of `sin` and `cos` functions.

Function	0°	30°	45°	60°	90°
sin	0	$\frac{1}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{\sqrt{3}}{2}$	1
cos	1	$\frac{\sqrt{3}}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{2}$	0
tan	0	$\frac{1}{\sqrt{3}}$	1	$\sqrt{3}$	Not defined

```
julia> r45d =(45*pi)/180 # 45 degrees to radians
0.7853981633974483

julia> r30d =(30*pi)/180 # 30 degrees to radians
0.5235987755982988

julia> r90d =(90*pi)/180 # 90 degrees to radians
1.5707963267948966

julia> sin(r45d) # sin(45) = 1/sqrt(2)
0.7071067811865475
```

```
julia> 1/sqrt(2) # confirmation
0.7071067811865475

julia> tan(r45d) # tan(45) = 1
0.9999999999999999

julia> sin(r30d) # sin(30) = 1/2
0.49999999999999994
```

Discrepancies in Calculations

It is important to note that the functional values are not absolute in nature, but are, in fact, approximate values. For this reason, discrepancies are bound to occur, as in the following example:

$$sin\left(45^\circ = cos\left(45^\circ\right)\right) = \frac{1}{\sqrt{2}}$$

But Julia code outputs different approximate values:

```
julia> sin(r45d) == cos(r45d)
false

julia>   sin(r45d)
0.7071067811865475

julia>  cos(r45d)
0.7071067811865476

julia> sin(r45d) - cos(r45d)
-1.1102230246251565e-16

julia> a = 1/sqrt(2)
0.7071067811865475

julia> sin(r45d)-a
0.0

julia> cos(r45d)-a
1.1102230246251565e-16
```

Another discrepancy occurs when calculating *tan*(90°), which is not defined mathematically since the following is true:

$$tan(90°) = \frac{sin(90°)}{cos(90°)} = \frac{1}{0}$$

But Julia code does give a value (very small). This happens because *cos*(90°) is not approximated to be truly zero, but a very small number. Since the result is not zero, *tan*(90°) has a finite value. This demonstrates that you should not convert radians to degrees and then start working with Julia's trigonometric functions. Instead, Julia provides a separate set of functions for usage with degree values as input. (They are outlined in following section.)

```
julia> tan(r90d)
1.633123935319537e16

julia> sin(r90d)/cos(r90d)
1.633123935319537e16

julia> sin(r90d)
1.0

julia> cos(r90d)
6.123233995736766e-17
```

Additional Features

Some additional functions can be made from these preliminary functions:

- Inverse functions include `asin`, `acos`, `atab`, `acsc`, `asec` and `acot`.
  - Inverse functions are defined such that

$$asin(x) = y \Rightarrow sin(y) = x$$

```
julia> a = sin(r45d)
0.7071067811865475
```

```
julia> asin(a)
0.7853981633974482

julia> r45d # calculated previously
0.7853981633974483
```

- Equivalent functions that take input angle in degrees are `sind`, `asind`, `cosd`, `acosd`, `tand`, `atand`, `cscd`, `acscd`, `secd`, `asecd`, `cotd`, and `acotd`.

```
julia> tand(90)
Inf

julia> sind(0)
0.0

julia> sind(90)
1.0

julia> sind(45)
0.7071067811865476

julia> 1/sqrt(2)
0.7071067811865475
```

- `sinpi(x)` and `cospi(x)` are provided for more accurate computations of $\sin(\pi \times x)$ and $\cos(\pi \times x)$ respectively, especially for bigger values of x.

```
julia> sinpi(1)
0.0

julia> sinpi(0.5)
1.0

julia> sinpi(0.25)
0.7071067811865476

julia> sinpi(0.44)
0.9822872507286887

julia> sin(10e20)
-0.6671201770718048
```

6.5.2 Hyperbolic Functions

Just as the points *cos*(x) and *sin*(x) form a circle with a unit radius, the points *cosh*(x) and *sinh*(x) form the right half of the equilateral hyperbola. They take a real number as an argument called *hyperbolic angle*. Julia provides a list of hyperbolic functions for evaluations including `sinh(x)`, `cosh(x)`, `tanh(x)`, `csch(x)`, `sech(x)`, and `coth(x)`. Their inverse counterparts are `asinh(x)`, `acosh(x)`, `atanh(x)`, `acsch(x)`, `asech(x)`, and `acoth(x)`.

```
julia> a = sinh(1)
1.1752011936438014

julia> asinh(a)
1.0

julia> a = sinh(0.5)
0.5210953054937474

julia> b = cosh(0.5)
1.1276259652063807

julia> c = tanh(0.5)
0.46211715726000974

julia> c1 = a/b
0.4621171572600098

julia> c == c1
false

julia> c-c1
-5.551115123125783e-17
```

The previous code outlines the similar problem in calculating $\tanh(x)=\frac{\sinh(x)}{\cosh(x)}$, as we observed when calculating *tan*. The result of a calculation from `tanh(x)` and calculating it as a ratio of `sinh(x)` and `cosh(x)` isn't the same since they are approximations limited by machine precision.

6.6 Iterative Algorithms to Calculate Mathematical Functions

It is worth understanding how mathematical functional values are calculated with a computer using algorithms. A computer *numerically approximates* the functional value using an algorithm based on a series expansion of a function.

6.6.1 Numerical Approximations

In the course of scientific investigation, finding exact answers may not be possible at times. Instead of devoting a lot of effort to find an exact answer by solving the problem *analytically*, another alternative is to develop methods for producing *approximate* answers *numerically*. The number of significant digits determined for a numerical approximation determines the accuracy of the answer. The degree of accuracy required for a result always depends on the targeted application. For example, measuring the length of a building does not need the answer to be accurate until the last length of an atom (Å). While measuring the body temperature of a human, you don't need to be accurate to more than two decimal places for most applications. In the era of faster and more efficient computers, higher accuracies of computations can be calculated by investing more time and storage, whenever required. But this facility must be used judiciously.

6.6.2 Tolerance

When an approximated answer or a set of approximated answers is available to the user, one of them must be chosen for a particular answer depending on the requirements of the applications. One way to make this decision is to define a *tolerance* limit. Tolerance can be defined as a single number or a range of numbers (having a maximum and a minimum). The rules to define tolerance limits are entirely application-oriented. For example, while measuring human height, you can define the tolerance to be 1 centimeter. However, at the same time, while measuring the diameter of a human hair, you would like to be more accurate by going down to 1 micron or less. While measuring the size of red blood cell, you would need to go further down to 1 nm. Whereas the decision to define tolerance is simpler while measuring sizes (that is, tolerance is one or two orders of magnitude smaller than the size of the object), it may not be a straightforward task in other applications. For example, the measurement of land for constructing a building would require a tolerance of a fraction of meters, whereas positioning a screw in a hole would require the accuracy of fraction of a centimeter.

In mathematical terms, if $\in$ is the tolerance limit, x is the real value, and x^* is approximated value, then the following is true:

$$\left|x - x^*\right| \leq \in \tag{6.18}$$

In this case, the absolute error (e_a) and relative error (e_r) in the measurements are given by the following:

$$e_a = \left|x - x^*\right| \tag{6.19}$$

$$e_r = \frac{\left|x - x^*\right|}{x} \tag{6.20}$$

Hence, if the absolute error is less than or equal to the tolerance limit, then the approximate solution/set of solutions is acceptable. However, if x is known, why do we need to calculate x^* (in other words, an approximate solution)?

In such cases where solutions of physical systems are unknown, x^* can be calculated and then be compared with physical measurements. The physical measurements constitute the value of x in this case and, consequently, errors can be calculated using Equation 6.20. Determining tolerance can then be determined around the fact that occasionally x^* will differ from x insignificantly; the errors won't matter much.

6.6.3 Taylor Series

Most mathematical functions would require very many complex operators other than the simpler ones (+, –, ×, and ÷) to be computed. However, a polynomial requires only the basic operators to be computed. Hence, if other mathematical functions can be represented in terms of polynomials, then they can be approximated with relative ease. The Taylor series expansion of a mathematical function performs this task.

A polynomial is defined as follows:

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \tag{6.21}$$

where $a_n \in R$ (the a's are called the *coefficients*). For the largest n that corresponds to $a_n \neq 0$, the degree of polynomial is defined as n.

6.6.4 Taylor Polynomials

Taylor's theorem shows the way to define a great many mathematical functions, which can be defined as polynomials called *Taylor polynomials*. The accuracy of the final answer shown by Taylor polynomial depends on its degree, that is, the number of terms defined in the polynomial. This provides a convenient methods to customize the polynomial as per desired tolerance.

Suppose a mathematical function `f (x)` needs to be approximated around $x = a$. A Taylor polynomial $p_n(x)$ of degree n centered at $x = a$ is that polynomial (of degree, at most, of n) that has the same value as n^{th} derivative at $x = a$.

The following points are true when deriving the formula for a Taylor polynomial:

- The zero order polynomial $p_0(x)$ has a degree, at most, of zero.
  - $p_0(x)$ must be a constant function (a horizontal line function graphically).
  - Approximating around $x = a$: $p_0(x) = f(a)$.
- The first order polynomial $p_0(x)$ has a degree, at most, of 1.
  - $p_1(x)$ must satisfy two conditions:

    $$p_1(a) = f(a)$$

    and

    $$p_1^{'}(a) = f'(a)$$

  - $p_1(x)$ must be of the form $p_1(x) = mx + c$ (a straight line with slope m and c as intercept).
  - Since $p_1^{'}(a) = f'(a)$ so $m = f'(a)$
  - So one can write $c = f(a) - f'(a)a$
  - Substituting back values of m and c, we get

$$p_1(x) = f'(a)x + f(a) - f'(a)a = f(a)(x - a)$$

Carrying forward the same arguments in a similar fashion, you can write the general form of Taylor polynomial of order *n* as follows:

$$p_n(x) = f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \frac{1}{3!}f'''(a)(x-a)^3 + \ldots + \frac{1}{n!}f^n(a)(x-a)^n$$

which can be rewritten in sigma notation as:

$$p_n(x) = \sum_{k=0}^{n} \frac{1}{k!} f^k(a)(x-a)^k \tag{6.22}$$

The previous definition requires that the polynomial must have *n* derivatives at $x = a$.

The Maclaurin series is simply the Taylor series defined for $a = 0$. Also using algebraic manipulations of The Taylor/Maclaurin series for basic functions such as *sin(x)*, *cos(x)*, e^x, and other complicated functions can also be defined in their series forms. These can be performed by simply using algebraical operators in addition to substitutions, derivatives, and integrations. This mathematical convenience comes in handy when formulating approximate solutions for physical systems defined by complicated functions. Let's study the Maclaurin series expansion of two of the most popular and widely used trigonometric functions—*sin(x)* and *cos(x)*.

6.6.5 Maclaurin Series for sin(x) and cos(x)

Both *sin(x)* and *cos(x)* are continuous and differentiable in the range given by any set of real numbers. Thus, their differentials exist in the same range. Consequently, they can be expanded in the form of a Maclaurin series.

Suppose *f(x)* = *sin(x)* needs to be approximated at $a = 0$.

Using Table 6-5 and Equation 6.22 results in the following:

$$sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \frac{1}{9!}x^9 - \ldots \pm \frac{1}{n!}x^n \tag{6.23}$$

***Table 6-5.** Calculating Coefficients for the Maclaurin Series of sin(x) at x = 0*

n	f (x)	f (a)
0	*sin*(*x*)	0
1	*cos*(*x*)	1
0	−*sin*(*x*)	0
1	−*cos*(*x*)	−1
0	*sin*(*x*)	0

Similarly *f(x)* = *cos(x)* needs to be approximated at *a* = 0.

Using Table 6-6 and Equation 6.22 results in the following:

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \frac{1}{8!}x^8 - \ldots \pm \frac{1}{n!}x^n \tag{6.24}$$

***Table 6-6.** Calculating Coefficients for the Maclaurin Series of cos(x) at x = 0*

n	f (x)	f (a)
0	*cos*(*x*)	1
1	−*sin*(*x*)	0
0	−*cos*(*x*)	−1
1	*sin*(*x*)	0
0	*cos*(*x*)	1

6.6.6 Series Expansion to Algorithms

A series expansion produces a series of terms that must be simply added to produce a functional approximated value. Algorithmically, one simply defines a general formula for calculation and loops over the calculations, each time adding the calculated value to the sum of values. This must be done until one satisfies the tolerance level. Users can set tolerance to a particular value.

This is how built-in Julia functions for mathematical functions are written. Apart from these simple ideas, users must also write smarter algorithms that give a wider range of operations and avoid overflow as well as underflow errors. Setting the functions confined to a particular data type depends on mathematical functions that need to be calculated.

6.6.7 How Many Numbers of Terms!

By increasing the number of terms from a series expansion, you reduce the error by many orders of magnitude. But does this trend mean that for achieving true values, one must include ∞ number of terms? After all, each time we add a new term, we invest in time and energy resources in our computation. In general, Maclaurin's series has the accuracy of a^{n+1} when n terms are used:

$$e^a = 1 + a + \frac{a^2}{2!} + \frac{a^3}{3!} + \frac{a^4}{4!} + \ldots + \frac{a^n}{n!} + O\left(a^{n+1}\right) \tag{6.25}$$

Analytically, one can choose n to be any large number, but this cannot be done on a computing machine. The chosen number is dictated by choosing tolerance to be closer to `eps` value.

6.7 Summary

In this chapter, we have summarized how mathematical functions are treated in Julia. A range of built-in functions performs mathematical operations on a range of data types. Using the `methods()` function, you can check which data types can be used with which functions. With predefined functions, performing mathematical calculations becomes easier and more organized, but users are free to write their own functions (user-defined functions) as per requirements, especially when built-in functions do not satisfy requirements. In order to do so, users would need knowledge to write loop structures and to define Julia functions. We will discuss these concepts in subsequent chapters.

6.8 Bibliography

[1] http://docs.julialang.org/en/stable/

[2] www.johndcook.com/blog/2010/06/02/whats-so-hard-about-finding-a-hypotenuse/

[3] www.johndcook.com/blog/2010/06/07/math-library-functions-that-seem-unnecessary/

CHAPTER 7

Arrays

7.1 Introduction to Arrays

Just as integers, floating point numbers, complex numbers, rational number data types, and irrational number data types define single values of different types, a host of other data types holding multiple values at the same time also exists. Such data types include `Array`, `Tuple` (tuples), `Dict` (dictionary), and `Set` (sets).

Arrays are particularly interesting since they are used for defining vectors, tables, and matrices for scientific computing:

- A 1D (one-dimensional) array acts as a vector or list.
- A 2D array can be used as a table or matrix.
- 3D and more-D arrays can represent multidimensional matrices.

An array is an ordered collection of elements. An array is an object that contains multiple data entries identified by their indices. Unlike many programming languages, the Julia `array` index starts at 1, not 0. An array is a collection of objects, where these collected objects are stored in a multidimensional grid. The dimension of an array is an abstract idea that we will discuss later in this chapter.

In the most general case, an array may contain objects of type `Any`, which essentially signifies it can store any variety of numeric data types. However, maintaining a uniformity of data structures helps manage the computational resources and avoid numerical computational errors. Hence, for most computational purposes, arrays should contain objects of a more specific type, such as `Float64` or `Int32`. Thus, `Array` data type objects can hold values of different data types or be restricted to values of a specific data type.

7.2 Construction

Arrays are often indicated with square brackets and comma-separated items. The following code shows the simplest way to handmake arrays:

```
julia> a = [1,2,3,4]
4-element Array{Int64,1}:
1
2
3
4

julia> b = [1 2 3 4]
1x4 Array{Int64,2}:
1  2  3  4

julia> size(a)
(4,)

julia> size(b)
(1, 4)

julia> ndims(a)
1

julia> ndims(b)
2
```

Issuing the command `a = [1,2,3,4]` in a REPL environment makes a 4-element array where each item is stored as `Int64` type (because input values of arrays are integers and the default data type of integers is `Int64`) and the array is defined to be a collection of four entries. The reference to an array object is depicted by the variable name `a`. On the other hand, when command `b = [1 2 3 4]` (each element is separated by a white space), we get a 1 × 4 array. The difference between the two can be probed by issuing the commands `size()` and `ndims()`, which give the size in terms of numbers of rows and columns. Also, when you create the array `a`, you observe information `4-element Array{Int64,1}:` signifying that it is a one-dimensional array storing four elements of data type `Int64`, whereas `b` informs that `1x4 Array{Int64,2}`. In other words, it is a two-dimensional array with the shape 1 × 2 (1 row and 2 columns) storing elements of data type `Int64`.

Alternatively, one can also define an array containing random numbers with the command structure `Array{data_type}(Number)` where the data type is defined within curly brackets `{}` and the number of elements is defined within simple brackets `()`. Please note that the elements are assigned randomly so the result will differ each time the command is executed. For example, when the command `a = Array{Int64}(3)` is issued two times, you obtain an array with three elements of random numbers for the type `Int64`:

```
julia> a = Array{Int64}(3)
3-element Array{Int64,1}:
4570398928
4509616976
4570398960
```

```
julia> a = Array{Int64}(3)
3-element Array{Int64,1}:
4510821744
4510825104
4507551632
```

Similarly, an array of three random numbers can be fabricated using the following:

```
julia> a = Array{Complex64}(3)
3-element Array{Complex{Float32},1}:
4.38586f-31+1.4013f-45im
2.76974f-31+1.4013f-45im
2.64668f-31+1.4013f-45im
```

7.2.1 Arrays of Multiple Dimensions

Just as you defined a 1D array, you can define multidimensional arrays by inputting the number of elements in each dimension. For example, `a = Array{Int64}(2,3,4)` will create an array of the size 2 × 3 × 4. The first dimension has two values, the second dimension has three values, and the third dimension has four values:

```
julia> a = Array{Int64}(2,3,4)
2x3x4 Array{Int64,3}:
[:, :, 1] =
4730481968  4730482096  4730597712
4730482032  4730597232  4730481168
```

```
[:, :, 2] =
4730481200  4730481264  4730481328
4730481232  4730481296  4730481360

[:, :, 3] =
4730481392  4730481456  4730481520
4730481424  4730481488  4730481552

[:, :, 4] =
4730481584  4730481648  4730481712
4730481616  4730481680  4730481744
```

The notation `[:,:,1]` will become clear in subsequent sections.

7.2.2 Arrays of Floats

If even one element of an object is defined as a floating point number, the data type of all number elements becomes `Float64`:

```
julia> a = [1,2.0,3,4]
4-element Array{Float64,1}:
1.0
2.0
3.0
4.0

julia> whos() # checking memeory usage
Base  34453 KB     Module
Core  12510 KB     Module
Main  41151 KB     Module
a     32 bytes  4-element Array{Float64,1}
ans     32 bytes  4-element Array{Float64,1}
```

It is worth noting that the array a uses 32 bytes for its storage since each element is `Float64` type, which uses 8 bytes $\Rightarrow 8 \times 4 = 32$ bytes. Hence, it is important to estimate the size required to store an array when it contains a huge number of elements. This is particularly important for devices and applications where memory is not a luxury like single board computers (Raspberry Pi, for example).

An array of random numbers can be created in a manner similar to that of integers:

```
julia> a = Array{Float64}(7)
7-element Array{Float64,1}:
2.32224e-314
2.32224e-314
2.32224e-314
2.32224e-314
2.32433e-314
2.32472e-314
2.32472e-314
```

```
julia> a = Array{Float64}(2,3,4)
2x3x4 Array{Float64,3}:
[:, :, 1] =
2.31516e-314  2.31516e-314  2.31516e-314
2.31516e-314  2.31516e-314  2.31531e-314

[:, :, 2] =
2.31531e-314  2.31538e-314  2.31538e-314
2.31531e-314  2.31538e-314  2.31563e-314

[:, :, 3] =
2.31563e-314  2.31538e-314  2.31538e-314
2.31561e-314  2.31538e-314  2.31554e-314

[:, :, 4] =
2.31538e-314  2.31553e-314  2.31553e-314
2.31553e-314  2.31553e-314  2.31538e-314
```

7.2.3 Array of Functions

Since elements of an array can be of any data type, even mathematical functions can themselves be an element of an array. Mathematical functions defined in Chapter 6 are defined under the data type `Function`. An array of mathematical functions can be defined as follows:

```
julia> a = [sin,cos,tan,log]
4-element Array{Function,1}:
sin
cos
tan
log
```

7.2.4 Arrays of Mixed Data Types

It is possible to create an array of mixed data types, too. For example, `a = [sin,1,1.5,2+5im,2//3]` creates an array were elements belong to `Function`, `Int64`, `Float64`, `Complex(Int64)`, and `Rational{Int64}` data types:

```
julia> a = [sin,1,1.5,2+5im,2//3]
5-element Array{Any,1}:
Sin
1
1.5
2+5im
2//3
```

7.2.5 Creating Arrays

Up to this point, we have just learned to make smaller arrays where the number of elements is small. What if you need to make an array of a large number of integers or floating point numbers separated by defined values, say odd integers from 1 to 1000? You would not like to feed these elements of arrays by hand. The `:` operator comes in handy in this case.

The operator `n:m` defines a range from `n` to `m` and, thus, can be used to create an array of a sequence of numbers. Using the `collect()` function, an array can be constructed for a predefined range of numbers. The start and stop numbers can be floating point numbers, too:

```
julia> collect(1:5)
5-element Array{Int64,1}:
1
2
```

```
3
4
5
```

```
julia> collect(1.1:5.6)
5-element Array{Float64,1}:
1.1
2.1
3.1
4.1
5.1
```

```
julia> collect(1:2:1000) # odd integers from 1 to 1000
500-element Array{Int64,1}:
1
3
5
7
.
.
.
995
997
999
```

It is worth noting that the increment (difference) between elements is set to 1 by default. This can be changed as necessary in the following way:

```
julia> collect(1:2:9)
5-element Array{Int64,1}:
1
3
5
7
9
```

```
julia> collect(1.5:2.2:9.9)
4-element Array{Float64,1}:
1.5
3.7
5.9
8.1
```

Arguments are presented as *start:increment:stop*. Also *stop* indicates the biggest number the array can contain. Elements of an array must be, at most, the *stop* number or less.

Increments can be negative numbers, too. For example:

```
julia> collect(10.5:-1.2:3.3)
7-element Array{Float64,1}:
10.5
9.3
8.1
6.9
5.7
4.5
3.3

julia> collect(10:-3.2:1)
3-element Array{Float64,1}:
10.0
6.8
3.6

julia> collect(10:-3:1)
4-element Array{Int64,1}:
10
7
4
1

julia> collect(1:-3:10)
0-element Array{Int64,1}
```

The command `collect(1:-3:10)` produces a null array (an array having zero elements) since negative increments cannot be implemented starting from 1 to 10.

7.2.6 Creating an Array Using the Ellipsis Operator

The ellipsis operator … can be used to create an array with a range of objects without using the `collect()` function:

```
julia> a = [1:5...]
5-element Array{Int64,1}:
1
2
3
4
5
```

```
julia> a = [1:2:9...]
5-element Array{Int64,1}:
1
3
5
7
9
```

```
julia> a = [1.2:2.2:9.9...]
4-element Array{Float64,1}:
1.2
3.4
5.6
7.8
```

7.2.7 Creating Arrays Using linspace

Another range object, namely `linspace()`, can be used to create arrays. `linspace` stands for *linearly spaced points*. It takes three arguments as *start:stop:number* where *number* defines the *integer* number of elements desired. For example:

```
julia> a = linspace(1,100,5)
5-element LinSpace{Float64}:
1.0,25.75,50.5,75.25,100.0

julia> a = linspace(1,100,3)
3-element LinSpace{Float64}:
1.0,50.5,100.0

julia> step(a) # Finding the step size
49.5
```

The function `step()` outputs the step size of a range object. Step size is easy to calculate. If `linspace(a,b,n)` is defined, then the step size *s* is the following:

$$s = \frac{b-a}{n} \tag{7.1}$$

Now, this `linspace` object can be fed to the `collect()` function to construct an array:

```
julia> a = linspace(1,100,3)
3-element LinSpace{Float64}:
1.0,50.5,100.0

julia> collect(a)
3-element Array{Float64,1}:
1.0
50.5
100.0
```

7.2.8 Creating Arrays Using logspace

Just like `linspace` produces linearly spaced points, `logspace` produces *logarithmically spaced* points. `logspace(1,100,2)` means to go from 10^1 to 10^{100} in two steps. Similarly, `logspace(2,5,5)` means to go from 10^2 to 10^5 in five steps:

```
julia> a = logspace(1,100,2)
2-element Array{Float64,1}:
10.0
1.0e100
```

```
julia> a = logspace(2,5,5)
5-element Array{Float64,1}:
100.0
562.341
3162.28
17782.8
100000.0
```

7.2.9 Similar Arrays

The built-in function `similar()`creates an array that is similar to a given array but that can be different in the data type of elements. For example, suppose one created an 2 × 3 array with data type `Float64` and saved in variable name `A`. Then a new array saved in variable name `A1` can be created of the same shape but with the data type `Int8`:

```
julia> A = Array{Float64}(2,3)
2x3 Array{Float64,2}:

2.25514e-314  2.25514e-314  2.25515e-314
2.25514e-314  2.25518e-314  2.25514e-314

julia> similar(A,Int8)
2x3 Array{Int8,2}:
64  -30  1
57   13  0
```

Similarly, one can create an array of `Float64` data type from an array of boolean numbers:

```
julia> A = Array{Bool}(2,3)
2x3 Array{Bool,2}:
false   true  true
true  false false

julia> similar(A,Float64)
2x3 Array{Float64,2}:
2.26024e-314  2.26024e-314  2.26024e-314
2.26024e-314  2.26024e-314  2.26024e-314
```

7.3 Properties of Arrays

A variety of built-in functions can be used to probe various properties of array objects. We covered two of them, `size()`and `ndims()`, in Section 7.2. Let's consider some more functions:

- `eltype`: Type of element

  ```
  julia> A = [1,2,3,4]
  4-element Array{Int64,1}:
  1
  2
  3
  4

  julia> eltype(A)
  Int64
  ```

- `length`: Number of elements

  ```
  julia> A = [1.1,-2.9,3.7,4.9]
  element Array{Float64,1}:
  1.1
  -2.9
  3.7
  4.9

  julia> length(A)
  4
  ```

- `ndims`: Number of dimensions

  ```
  julia> A = Array{Int64}(2,3,5)
  2x3x5 Array{Int64,3}:
  [:, :, 1] =
  4694973424  4694973552  4694972656
  4694973488  4694973616  4694972688

  [:, :, 2] =
  4694972720  4694972784  4694972848
  4694972752  4694972816  4694972880
  ```

```
[:, :, 3] =
4694972912  4694972976  4694973040
4694972944  4694973008  4694973072

[:, :, 4] =
4694973104  4694993040  4694973232
4694992976  4694973200  4694973264

[:, :, 5] =
4694973296  4719342608  4719342704
4694973328  4719342640           0

julia> ndims(A)
3

julia> A = Array{Float64}(6)
6-element Array{Float64,1}:
2.30585e-314
2.33172e-314
2.33172e-314
2.33172e-314
2.33172e-314
2.33172e-314

julia> ndims(A)
1
```

- `size()`: Size of the array (how many elements exist in each of its dimensions)

```
julia> A = Array{Int64}(2,4,3)

2x4x3 Array{Int64,3}:
[:, :, 1] =
4694973424  4694973552  4694972656  4694972720
4694973488  4694973616  4694972688  4694972752

[:, :, 2] =
4694972784  4694972848  4694972912  4694972976
4694972816  4694972880  4694972944  4694973008
```

```
[:, :, 3] =
4694973040  4694973104  4694993040  4694973232
4694973072  4694992976  4694973200  4694973264

julia> size(A) # No. of elements in each dimesnion
(2,4,3)

julia> size(A,3) # No. of elements in 3rd dimesnion
3

julia> size(A,2) # No. of elements in 2nd dimesnion
4

julia> size(A,1) # No. of elements in 1st dimesnion
2
```

- indices: Indices of the array

```
julia> A = [1,2,3,4,5]
5-element Array{Int64,1}:
1
2
3
4
5

julia> indices(A)
(Base.OneTo(5),)

julia> B = [1 2 3 4 5]
1x5 Array{Int64,2}:
1  2  3  4  5

julia> indices(B)
(Base.OneTo(1), Base.OneTo(5))
```

For array B, indices run from 1 to 1 in the first dimension and from 1 to 5 in the second dimension.

7.4 Indexing

The index of an element is the address of the same element within an array. Indexing in Julia starts at 1. For example, in a 1D array, the index of an element is the number of the element's position from the left. Julia does not have negative indexing. In other words, elements can only be approached from the left.

```
julia> a = [12,4,6,3,6]
5-element Array{Int64,1}:
12
4
6
3
6

julia> a[3]
6

julia> a[1]
12
```

7.4.1 Creating Subarrays Using : operator

Using indices and `:` operator, you can create subarrays. This is sometimes referred to as slicing an array. For example, if an array is stored in a variable named `a`, then `a[n:m]` will return another array with an element starting from the index `n` to `m`. Since the `:` defines a range of elements, it is sometimes referred to as *range* operator. The following example will make this concept clearer:

```
julia> a = [12,4,6,3,6]
5-element Array{Int64,1}:
12
4
6
3
6
```

```
julia> a[2:4]
3-element Array{Int64,1}:
4
6
3

julia> a[range(2,4)]
4-element  Array{Int64,1}:
4
6
3
6
```

The `range()` function can also be used instead of the `:` operator. If `n` or `m` exceeds the bounds of a defined array, you will encounter a `BoundsError`.

7.4.2 end

Using the keyword end, one can access the last element of an array as follows:

```
julia> A = [1,2,3,4,5]
5-element Array{Int64,1}:
1
2
3
4
5

julia> A[1]
1

julia> A[end]
5

julia> A[end-2]
3
```

This can also be used in making a subset of a given array:

```
julia> a = collect(1:7)
7-element Array{Int64,1}:
1
2
3
4
5
6
7

julia> a[2:2:end]
3-element Array{Int64,1}:
2
4
6
```

a = collect(1:7) creates an array having numbers from 1 to 7. Now a[2:2:end] creates a new array that starts with 2 and goes until the end of the original array a in steps of two (elements 2, 4, 6).

7.4.3 Slicing Multidimensional Arrays

Slicing a multidimensional array is one of the key skills in real-world data analytics. Finding the *part* of an array that you need to process and then slice it out of the main array as a separate entity would require the knowledge of accessing the elements of the array within a multidimensional framework:

```
julia> a = [[1,2,3] [4,5,6] [7,8,9]]
3x3 Array{Int64,2}:
1  4  7
2  5  8
3  6  9

julia> a[2,3] # 2nd row, 3rd column element
8
```

```
julia> a[3,2] # 3rd row,2nd column element
6

julia> a[3,end] # 3row, last column element
9

julia> a[end,3] # last row, 3rd column element
9

julia> a[end,end] # last row and column element
9
```

Now the : operator can be used within slicing operations to select an entire row or column, or particular parts of the same:

```
julia> a = [[1,2,3] [4,5,6] [7,8,9]]
3x3 Array{Int64,2}:
1  4  7
2  5  8
3  6  9

julia> a[:,3] # All elements of 3rd column
3-element Array{Int64,1}:
7
8
9

julia> a[3,:] # All elements of 3rd row
3-element Array{Int64,1}:
3
6
9

julia> a[3,1:2] # 3rd row and first to second rows
2-element Array{Int64,1}:
3
6
```

```
julia> a[2:3,2] # 2nd to third row and 2nd column
2-element Array{Int64,1}:
5
6

julia> a[2:3,2:3] # rows from 2nd to 3rd,
# columns from 2nd to 3rd
2x2 Array{Int64,2}:
5  8
6  9

julia> a[2:end,1:end] # rows from 2nd to last,
# columns from 1st to last
2x3 Array{Int64,2}:
2  5  8
3  6  9
```

7.5 Filling Arrays with Values

Automatically filling an array with data can be accomplished with a range of functions.

7.5.1 zeros()

An array of all elements as 0s can be constructed using the `zeros()` function as follows:

```
julia> a = zeros(7)
7-element Array{Float64,1}:
0.0
0.0
0.0
0.0
```

Syntax	Behavior
`zeros(A)`	an array of all zeros of the same element type and shape as `A`
`ones(A)`	an array of all ones of the same element type and shape as `A`
`trues(A)`	a `Bool` array with all values `true` and the shape of `A`
`falses(A)`	a `Bool` array with all values `false` and the shape of `A`
`rand(n)`	an array of `n` uniformly distributed random numbers in interval [0, 1)
`randn(n)`	an array of `n` normally distributed random numbers
`eye(n)`	an $n \times n$ identity matrix
`eye(n,m)`	an $n \times m$ identity matrix
`fill(x,n)`	an array of dimensions `n`, filled with value `x`

```
0.0
0.0
0.0

julia> eltype(a)
Float64

julia> a = zeros(2,3)
2x3 Array{Float64,2}:
0.0  0.0  0.0
0.0  0.0  0.0

julia> a[2,2]
0.0

julia> a[2,2] == a[2,3] == a[1,1]
true
```

7.5.2 ones()

An array of all elements as 1s can be constructed using the `ones()` function as follows:

```
julia> a = ones(7)
7-element Array{Float64,1}:
1.0
1.0
1.0
1.0
1.0
1.0
1.0

julia> eltype(a)
Float64

julia> a = ones(3,4)
3x4 Array{Float64,2}:
1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0

julia> a = ones(3,4,2)
3x4x2 Array{Float64,3}:
[:, :, 1] =
1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0

[:, :, 2] =
1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0
```

7.5.3 trues()

Just like numerical 0s and 1s can be filled in an array, boolean `true` can be filled in an array using the function `trues()` as follows:

```
julia> a = trues(2,3)
2x3 BitArray{2}:
true  true  true
true  true  true

julia> a = trues(3,4,2)
3x4x2 BitArray{3}:
[:, :, 1] =
true  true  true  true
true  true  true  true
true  true  true  true

[:, :, 2] =
true  true  true  true
true  true  true  true
true  true  true  true
```

7.5.4 falses()

In similar fashion, an array of boolean value `false` can be filled in an array using the function `falses()`:

```
julia> a = falses(2,3)
2x3 BitArray{2}:
false  false  false
false  false  false

julia>  a  =  falses(3,4,2)
3x4x2  BitArray{3}:
[:, :, 1] =
false  false  false  false
false  false  false  false
false  false  false  false
```

```
[:, :, 2] =
false  false  false  false
false  false  false  false
false  false  false  false
```

7.5.5 Arrays Filled with Random Numbers

Two functions provide arrays filled with random numbers. `rand()` provides uniformly distributed random numbers within the interval [0, 1). On the other hand, `randn()` provides an array filled with normally distributed random numbers:

```
julia> a = rand(8)
8-element Array{Float64,1}:
0.72864
0.203516
0.512295
0.449959
0.211407
0.348952
0.677256
0.585907

julia> a = rand(8)
8-element Array{Float64,1}:
0.591333
0.140416
0.127931
0.291892
0.0306536
0.0559765
0.959664
0.263331
```

It is important to note that just like the two instances when we ran the command `rand(8)`, we got a different set of random numbers, we should also expect to get a different set of random numbers while running this command:

```
julia> a = rand(8)
8-element Array{Float64,1}:
0.123916
0.577333
0.786042
0.19784
0.757978
0.481438
0.375539
0.949668
```

```
julia> a = randn(8)
8-element Array{Float64,1}:
-0.365407
-1.31341
-0.331167
-0.180398
-0.860501
0.831122
-0.223168
0.226383
```

The command works in a similar fashion for higher dimensional arrays:

```
julia> a = rand(2,3,5)
2x3x5 Array{Float64,3}:
[:, :, 1] =
0.0306414  0.767554  0.696444
0.0924386  0.334853  0.627763

[:, :, 2] =
0.505539  0.00991551  0.277056
0.553033  0.272472    0.381655
```

```
[:, :, 3] =
0.033432  0.826044   0.689259
0.927387  0.18994    0.517047

[:, :, 4] =
0.725888  0.261185   0.155774
0.623608  0.211425   0.237139

[:, :, 5] =
0.669267  0.659699   0.859842
0.691922  0.51326    0.156616

julia> a = randn(2,3,5)
2x3x5 Array{Float64,3}:
[:, :, 1] =
-1.45105    0.506138    0.607333
0.269298  -0.172373   -0.7592

[:, :, 2] =
-0.463605   0.459976   1.75424
-0.753155   0.043333   0.0107971

[:, :, 3] =
-0.470125   -0.426953   -1.02621
-0.0536346   1.10199     0.122024

[:, :, 4] =
1.00947     0.83089  0.395584
-0.0139497  -0.5233   0.405085

[:, :, 5] =
1.37339  -0.519548  0.556255
1.28814  -1.52136   0.0380677
```

7.5.6 eye()

An identity matrix is the one where diagonal elements are 1 and other elements are 0. This can be created in Julia using the `eye()` command as follows:

```
julia> a = eye(3)
3x3 Array{Float64,2}:
1.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  1.0

julia> a = eye(5)
5x5 Array{Float64,2}:
1.0  0.0  0.0  0.0  0.0
0.0  1.0  0.0  0.0  0.0
0.0  0.0  1.0  0.0  0.0
0.0  0.0  0.0  1.0  0.0
0.0  0.0  0.0  0.0  1.0

julia> a = eye(5,3)
5x3 Array{Float64,2}:
1.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  1.0
0.0  0.0  0.0
0.0  0.0  0.0
```

It is worth noting that with a singular argument, `eye(n)`, an $n \times n$ square identity matrix is created, whereas `eye(n,m)` creates a matrix of dimension $n \times m$.

7.5.7 fill()

Just like filling a matrix with 0s and 1s can be done with `zeros` and `ones`, the `fill` command produces an array filled with a desired numerical value as all of its elements:

```
julia> fill(5,3)
3-element Array{Int64,1}:
5
```

```
5
5

julia> fill(5,(2,3))
2x3 Array{Int64,2}:
5  5  5
5  5  5

julia> fill(5,(2,3,5))
2x3x5 Array{Int64,3}:
[:, :, 1] =
5  5  5
5  5  5

[:, :, 2] =
5  5  5
5  5  5

[:, :, 3] =
5  5  5
5  5  5

[:, :, 4] =
5  5  5
5  5  5

[:, :, 5] =
5  5  5
5  5  5
```

7.6 Reshaping Arrays

Reshaping an array means to change its dimensions. For example, a 1D array of the shape 1 × 20 can be reshaped in a variety of ways: 4 × 5, 5 × 4, 2 × 2 × 5, and so on. This can be done by the `reshape()` functions as shown below:

```
julia> A = Array{Int8}(4,5)
4x5 Array{Int8,2}:
-32  1  48  1  -16
```

```
-101  0  83  0  -101
-46  0  -7  0   -46
15  0   9  0    15

julia> A1 = reshape(A,(5,4))
5x4 Array{Int8,2}:
-32   0  -7     0
-101   0   9   -16
-46   0   1  -101
15  48   0   -46
1  83   0    15

julia> A1 = reshape(A,(20))
20-element Array{Int8,1}:
-32
-101
-46
15
1
0
0
0
48
83
-7
9
1
0
0
0
-16
-101
-46
15
```

```
julia> A1 = reshape(A,(2,2,5))
2x2x5 Array{Int8,3}:
[:, :, 1] =
-32  -46
-101   15

[:, :, 2] =
1  0
0  0

[:, :, 3] =
48  -7
83   9

[:, :, 4] =
1  0
0  0

[:, :, 5] =
-16  -46
-101   15
```

7.6.1 Flipping

Flipping a particular dimension of a matrix can be performed at dimension `n` for a matrix `A` using `flipdim(A,n)`:

```
julia> A = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
1  4
2  5
3  6

julia> flipdim(A,1) # flipping row (dim=1)
3x2 Array{Int64,2}:
3  6
2  5
1  4
```

```
julia> flipdim(A,2) # flipping column (dim=2)
3x2 Array{Int64,2}:
4  1
5  2
6  3
```

7.6.2 Squeezing and Arrays

Another built-in function named `squeez()` seems similar to reshaping, but it has quite different behavior. `squeeze(A, dims)` removes the dimensions specified by `dims` from array `A`:

```
julia> a = reshape(collect(1:9),(1,3,1,3))
# Array of dimesnion 1x3x1x3 is created
1x3x1x3 Array{Int64,4}:
[:, :, 1, 1] =
1  2  3

[:, :, 1, 2] =
4  5  6

[:, :, 1, 3] =
7  8  9

julia> squeeze(a,3)
# 3rd dimesnion is removed and from
# Array of 1x3x1x3 a new
# Array of dimesnion 1x3x3 is created
1x3x3 Array{Int64,3}:
[:, :, 1] =
1  2  3

[:, :, 2] =
4  5  6

[:, :, 3] =
7  8  9
```

```
julia> squeeze(a,1)
# 1st dimesnion is removed and from
# Array of 1x3x1x3 a new
# Array of dimesnion 3x1x3 is created
3x1x3 Array{Int64,3}:
[:, :, 1] =
1
2
3

[:, :, 2] =
4
5
6

[:, :, 3] =
7
8
9
```

7.7 Sorting

Sorting elements with a particular rule is an important aspect of matrix manipulation. The built-in function `sort(A),n` sorts a matrix `A` along a dimension `n`. By default, Julia picks reasonable algorithms and sorts in standard ascending order:

```
julia> sort([2,4,1,5,2,7,3])
7-element Array{Int64,1}:
1
2
2
3
4
5
7
```

If you need to sort in descending order, you can choose the argument `rev=true` (which is set to `false`) by default:

```
julia> sort([2,3,1,4,6,3,7], rev=true)
7-element Array{Int64,1}:
7
6
4
3
3
2
1
```

7.7.1 sortperm()

The built-in function `sortperm()` returns a permutation vector of indices of `v` that puts it in sorted order:

```
julia> A = [2,3,1,4,6,3,7]

7-element Array{Int64,1}:
2
3
1
4
6
3
7

julia> v = sortperm(A) # array indices for incremental values
7-element Array{Int64,1}:
3
1
2
6
4
5
7
```

```
julia> A[v] # Creating array with new vector of indices
7-element Array{Int64,1}:
1
2
3
3
4
6
7
```

7.7.2 Sort by Transformation

Within the sort() function, you can use a particular transformation to sort the elements in a particular fashion. For example, an array of positive and negative numbers is defined, but you wish to sort them by ignoring their sign. In this case, you can use the by=abs option in the sort() function to indicate that, while sorting, only *absolute value* must be considered, as shown in the following example Julia code:

```
julia> A = [-2,3,-4,-1,0,-5]
6-element Array{Int64,1}:
-2
3
-4
-1
0
-5

julia> sort(A,by=abs)
6-element Array{Int64,1}:
0
-1
-2
3
-4
-5
```

7.7.3 Sorting Algorithms

At present, there are four sorting algorithms to choose from:

- `InsertionSort`
  - It is used internally by `QuickSort`.
  - It is efficient for smaller arrays.
  - It has an order of $O(n2)$ and is stable.
- `QuickSort`
  - It has a default option for numeric values.
  - It has an order of $O(nlog(n))$ and, hence, it is very fast.
  - It is not stable.
    - Elements that are considered *equal* do not remain in the same order in which they originally appeared.
- `PartialQuickSort(k)`
  - It is similar to `QuickSort`, but the output array is only sorted up to index `k` where `k` has to be an integer.
- `MergeSort`
  - It is the default algorithm for non-numeric data.
  - It has an order of $O(nlogn)$ and is stable.
  - It is typically not as fast as `QuickSort`.

Let's do a small experiment to validate these claims about various algorithms at the Julia documentation web page. You can test the usage of the `alg` option within the `sort()` function and time each event for various algorithms using the `tic()` and `toc()` functions, which start and stop recording time as they appear.

It must be noted that this particular way of recording time of execution is not the best option for benchmarking performance of an algorithm because it has not been normalized for different processors, OS, and other parameters. Also a processor is free to run an observed process (here the Julia command we are interested in) for different intervals of time at different points of times. Hence, the time of execution will not even be the same for code running on the same computer at different points of times. But our aim is just to quickly check the time required for running the same sorting problem with a different algorithm. In addition, the random numbers will be different each time the command `rand()` is written. Thus, our present study is not enough to make judgments about sorting algorithms, but rather to just test their execution times for a very crude comparison.

The following Julia code is shown for this purpose. A Julia array is created with `rand(1:10000,100000)` having 100,000 random numbers between 1 and 10,000. `k` is set to 50,000 so that half of the numbers can be sorted. Then `sort()` is sandwiched between the `tic()` and `toc()` command to obtain time elapsed to run the execution of code. The results are compiled in Table 7-1.

```
julia> A = rand(1:10000,100000);

julia> k = 50000;

julia> tic();qs=sort(A;alg=InsertionSort);toc()
elapsed time: 1.844173231 seconds
1.844173231

julia> tic();qs=sort(A;alg=QuickSort);toc()
elapsed time: 0.035997547 seconds
0.035997547

julia> tic();ps=sort(A;alg=PartialQuickSort(k));toc()
elapsed time: 0.022802908 seconds
0.022802908

julia> tic();qs=sort(A;alg=MergeSort);toc()
elapsed time: 0.037192645 seconds
0.037192645
```

***Table 7-1.** Time Elapsed Study for Various Sorting Algorithms*

Algorithm	Time Elapsed (s)	Rank According to Speed
`InsertionSort`	1.844173231	4
`QuickSort`	0.035997547	2
`PartialQuickSort(500)`	0.022802908	1
`MergeSort`	0.037192645	3

Since `PartialQuickSort(500)` sorted only half the values, it came out the fastest; however, compared to `QuickSort`, it is not faster when normalized with a number of elements. So `QuickSort` is actually the fastest algorithm for the present crude experiment.

7.7.4 Lexicographical Order

The built-in functions `sortrows()` and `sortcolumns()` are in lexicographical order. Lexicographical order is sometimes called dictionary order because language dictionaries follow the same order. Let's test the same concept on an array of characters:

```
julia> A1 = [['b','a','c'] ['d','f','e']]
3x2 Array{Char,2}:
'b'  'd'
'a'  'f'
'c'  'e'

julia> sort(A1,1)
3x2 Array{Char,2}:
'a'  'd'
'b'  'e'
'c'  'f'

julia> sort(A1,2)
3x2 Array{Char,2}:
'b'  'd'
'a'  'f'
'c'  'e'
```

```
julia> sortrows(A1)
3x2 Array{Char,2}:
'a'  'f'
'b'  'd'
'c'  'e'

julia> sortcols(A1)
3x2 Array{Char,2}:
'b'  'd'
'a'  'f'
'c'  'e'
```

7.8 Finding Items in Arrays

Using the in() function, you can check if an item is a member of arrays, that is, if its value matches the value of the elements. This seemingly insignificant facility proves to be very powerful in writing comprehensions and loop structures, which makes Julia an excellent choice for numerical computations. Following are two versions of its usage:

```
julia> A = [2,4,1,5,6]
5-element Array{Int64,1}:
2
4
1
5
6

julia> 2 in A
true

julia> in(2,A)
true

julia> B = [A,A]
2-element Array{Array{Int64,1},1}:
[2,4,1,5,6]
[2,4,1,5,6]
```

```
julia> 2 in B
false

julia> A in B
true
```

It is clear from the previous example that elemental value is compared for `in()` member function. When an array `B` is made using array `A` as two of its elements, then the numeric value 2 is not found to be a member of `B`.

7.8.1 find(), findfirst(), and findnext()

Apart from just sensing the presence of a similar value, sometimes you need to find the exact position of a value inside an array. The positions are addressed by indices. The built-in function `find()` outputs the same. Another set of built-in functions, `findfirst()` and `findnext()`, finds a value for its first occurrence and next to a given index, respectively:

```
julia> A = collect(1:20); # Array having 1 to 20 numbers

julia> find(isodd,A) # Finding numbers which are odd
10-element Array{Int64,1}:
1
3
5
7
9
11
13
15
17
19

julia> findfirst(isodd,A)
1

julia> findnext(isodd,A,findfirst(A))
1
```

```
julia> findnext(isodd,A,findfirst(A)+1)
3

julia> findnext(isodd,A,findfirst(A)+10)
11
```

7.9 Copying an Array

While copying the contents of arrays seems a straightforward task, it has two varieties: copy (shallow copy) and deepcopy. Let's first study an example and then examine what the difference between the two is:

```
julia> a = eye(3)
3x3 Array{Float64,2}:
1.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  1.0

julia> b = [1,2,a] # b has array 'a' as its third element
3-element Array{Any,1}:
1
2
[1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0]

julia> c = copy(b); d = deepcopy(b);

julia> b[3][1]=10 # changing first element of
# third element as 10
10

julia> b
3-element Array{Any,1}:
1
2
[10.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0]
```

```
julia> c # 'c' shallow copies a 'b' and changes
3-element Array{Any,1}:
1
2
[10.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0]

julia> d # # d maintains deep copy of 'a'
3-element Array{Any,1}:
1
2
[1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0]
```

7.9.1 Deepcopy

From the previous example, it's clear that while `copy` makes merely a new reference (see Chapter 2, Section 2.6) to the same memory location as that of the original object, `deepcopy` makes an entirely new copy (hence, a new memory location). This is the reason that when the original object is changed, `copy` reflects those changes, while `deepcopy` does not. This also means that while using `copy` does not increase the memory footprint drastically, `deepcopy` does, especially for the cases when arrays occupy a significant percentage of available memory. As a result, the two options should be used judiciously as per requirements and available resources.

It is worth noting that while the command `similar()` (as explained in Section 7.2.9) copies only the size, `copy()` and `deepcopy()` copy both the size and content.

7.10 Comprehension

Comprehension means to create arrays by a defined rule. It provides a general and powerful way to construct arrays. The syntax is similar to a set of construction notation in mathematics:

```
A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

In other words, it is comprised of a list of values for variables `x` and `y`. For each value, `F (x, y)` is calculated and the element of the arrays is created. An example will make this concept clear:

```
julia> A = rand(1:100,7)
7-element  Array{Int64,1}:
24
3
90
80
20
78
57

julia> [A[i]^2 for i=1:length(A)]
# All elements of A are squared
7-element Array{Int64,1}:
576
9
8100
6400
400
6084
3249

julia> [n^2 for n in A]
# simpler way to perform the same
7-element Array{Int64,1}:
576
9
8100
6400
400
6084
3249
```

```
julia> [sqrt(A[i]) for i=1:length(A)]
# All elments of A are square rooted
7-element Array{Float64,1}:
4.89898
1.73205
9.48683
8.94427
4.47214
8.83176
7.54983

julia> [1//2*A[i]+1//3*A[i+1] for i=2:length(A)-1]
# one half of element is added to one third
# of next element of A for createing new element
5-element Array{Rational{Int64},1}:
63//2
215//3
140//3
36//1
58//1

julia> [complex(A[i],A[i+1]) for i=1:length(A)-1]
# A complex number is created with real part
# is the element and complex part is
# thr next element of A
6-element Array{Complex{Int64},1}:
24+3im
3+90im
90+80im
80+20im
20+78im
78+57im
```

The resulting array type depends on the types of the computed elements. As seen in the previous example, the `eletype()` of an output array changed to type `rational` or `complex numbers`, depending on the defined operation. For defining the type explicitly, you can define the type of an output array. For example, if you desire to get floating point numbers instead of rational numbers for the defined command `[1//2*A[i]+1//3*A[i+1] for i=2:length(A)-1]`, you can explicitly define the output array type by printing `Float64` at the beginning:

```
julia> Float64[1//2*A[i]+1//3*A[i+1] for i=2:length(A)-1]
5-element Array{Float64,1}:
31.5
71.6667
46.6667
36.0
58.0
```

A 2D and higher-dimension array can also be created simply by using comprehension. Just write the formula for creating elements and then assign ranges (separated by the `,` operator):

```
julia> [r^c for r in 1:5, c in 1:5]
5x5 Array{Int64,2}:
1   1    1    1     1
2   4    8   16    32
3   9   27   81   243
4  16   64  256  1024
5  25  125  625  3125

julia> [r^c+d for r in 1:3, c in 1:3, d in 3:5]
3x3x3 Array{Int64,3}:
[:, :, 1] =
4   4   4
5   7  11
6  12  30

[:, :, 2] =
5   5   5
6   8  12
7  13  31
```

```
[:, :, 3] =
6   6   6
7   9  13
8  14  32
```

7.11 Generator Expressions

The comprehension style of defining the arrays requires the formula for generating the elements to be written within square brackets. When it is written outside the square brackets, it generates an object called `Generator`. This object can then be used in defining the comprehension. `Generator` can be iterated to produce values on demand instead of allocating an array and storing them in advance. For example:

```
julia> collect(x^y for x in 1:3,y in 1:3)
# Array with element as x^y where
# x is from 1 to 3
# y is from 1 to 3
3x3 Array{Int64,2}:
1  1   1
2  4   8
3  9  27

julia> collect(sin(x)*min(y) for x in pi:4*pi, y in [-2,4,5])
# Array with element given by formula
# sin(x)* min(y)
10x3 Array{Float64,2}:
-2.44929e-16   4.89859e-16   6.12323e-16
1.68294      -3.36588      -4.20735
1.81859      -3.63719      -4.54649
0.28224      -0.56448      -0.7056
-1.5136        3.02721       3.78401
-1.91785       3.8357        4.79462
-0.558831      1.11766       1.39708
1.31397      -2.62795      -3.28493
1.97872      -3.95743      -4.94679
0.824237     -1.64847      -2.06059
```

7.12 Assignment Operator and Arrays

Assignment operator = usually assigns the value on the left-hand side to an argument on the right-hand side. This can be used to alter array values, too. For example:

```
julia> A = [1,2,3]
3-element Array{Int64,1}:
1
2
3
```

```
julia> A  =  [3,4,5] #  Changes  value  in same  size
3-element    Array{Int64,1}:
3
4
5
```

```
julia> A = [3,4] # changes value in different size
2-element Array{Int64,1}:
3
4
```

This behavior can be understood in terms of the concept of a variable being merely a reference to a memory location. The variable named A points to a memory location having a $3-elementArrayInt64,$ 1 object. When A is assigned to a different object, it simply points to a new object as per the new assignment. The new object can be very different from the original one.

An assignment operator can also be used to selectively assign new element values:

```
julia> A = rand(3,3)
3x3 Array{Float64,2}:
0.952371  0.0541676  0.957925
0.104845  0.168398   0.913292
0.571905  0.991414   0.0173661
```

```
julia> A[2:3, 3] = 0
# Assign the value zero to elements in
# rows from 2 to 3 and
# column number 3
0
```

```
julia> A
3x3 Array{Float64,2}:
0.952371  0.0541676  0.957925
0.104845  0.168398   0.0
0.571905  0.991414   0.0
```

7.12.1 Other Mathematical Operators

Chapter 8 is dedicated to explaining how mathematical functions can be operated on arrays and their elements. This chapter is critical for numerical experimentation as most of the data is converted into a matrix (stored in computer memory) and mathematical functions are used to define a *transformation equation*. This transformation equation operates on an input matrix and results in a new matrix (called *transformed matrix*). Simulating a real system involves defining transformation equations. These transformed matrices are converted back to the original form of data for visualization and interpretation. For this reason, Julia's abilities relating to speedy matrix transformation in a flexible manner must be understood in detail so that users can judge correctly which to choose and then define particular mathematical functions in the right manner.

7.13 Set Theory and Arrays

The `Array` data types can also be treated as equivalent to a mathematical set. The set operations like ∪ (Union) given by the built-in function `union()`, ∩ (Intersection) given by the built-in function `intersect()` and set difference (`setdiff(A-B)`) can be calculated. Union operation collects the unique occurrence of an element of both sets. Intersection collects common elements from both sets and set difference collects those elements that are present in `A` but not in `B`.

```
julia> A = [1,2,3,4,-1,-3]
6-element Array{Int64,1}:
1
2
3
4
-1
-3
```

```
julia> B = [2,4,1,3,1,10]
6-element Array{Int64,1}:
2
4
1
3
1
10

julia> union(A,B)
7-element Array{Int64,1}:
1
2
3
4
-1
-3
10

julia> intersect(A,B)
4-element Array{Int64,1}:
1
2
3
4

julia> setdiff(A,B)
2-element Array{Int64,1}:
-1
-3
```

7.14 Dictionary

An English dictionary maps a useful piece of information in the form of an illustrative paragraph and/or audio video files that can be found via a key. This kind of *associative collection* is used in computer science, too, where a *key-value* pair is stored for future use as a look-up table. By feeding a key, the value can be retrieved. A collection of such key-value pairs is called a *dictionary* for obvious reasons.

7.14.1 Creating a Dictionary

Creating dictionaries is quite straightforward where key-value pairs are associated using the => operator (called the `Pair()` function) and are separated by a comma. For example:

```
julia> dict = Dict("red"=>1,"blue"=>2,"green"=>3)
Dict{String,Int64} with 3 entries:
"blue"  => 2
"green" => 3
"red"   => 1

julia> dict = Dict("red"=>"Red","blue"=>"Red","green"=>"Green")
Dict{String,String} with 3 entries:
"blue"  => "Red"
"green" => "Green"
"red"   => "Red"

julia> dict = Dict(1=>"Red",2=>"Red",3=>"Green")
Dict{Int64,String} with 3 entries:
2 => "Red"
3 => "Green"
1 => "Red"
```

The data type for keys and values can be similar or dissimilar with the condition that keys must be unique.

If the data type of keys and values is known in advance, it can be alternatives defined as the following:

```
julia> dict1 = Dict{Integer,String}(1=>"A",2=>"b")
Dict{Integer,String} with 2 entries:
2 => "b"
1 => "A"
```

```
# An empty dictionary with known types
julia> dict1 = Dict{Integer,String}()
Dict{Integer,String} with 0 entries

# An empty dictionary with unknown types
julia> dict1 = Dict()
Dict{Any,Any} with 0 entries
```

7.14.2 Looking Up a Dictionary

We use the index of an element to find the element of an array. In a similar fashion, we use the key to find the value in a dictionary. Within square brackets, if a key is fed, the value is returned, Alternatively, we can use the `get()` function, which inputs the dictionary name and the key to output the value:

```
julia> dict = Dict(1=>"Red",2=>"Red",3=>"Green")
Dict{Int64,String} with 3 entries:
2 => "Red"
3 => "Green"
1 => "Red"

julia> dict[1]
"Red"

julia> dict[2]
"Red"

julia> dict[3]
"Green"

julia> get(dict,1,0)
"Red"

julia> get(dict,2,0)
"Red"

julia> get(dict,3,0)
"Green"
```

```
julia> get(dict,4,0)
0

julia> get(dict,4,"missing value")
"missing value"
```

The get() function uses the third argument (fed as a 0) in the previous code, which is the default value for output in case the key-value pair is missing. This is highlighted in the last two lines.

7.14.3 Finding Keys and Values

Keys of a dictionary can be found using the keys() function. The following Julia code gives one such example:

```
julia> dict = Dict(1=>"Red",2=>"Red",3=>"Green")
Dict{Int64,String} with 3 entries:
2 => "Red"
3 => "Green"
1 => "Red"

julia> keys(dict)
Base.KeyIterator for a Dict{Int64,String} with 3 entries. Keys:
2
3
1

julia> values(dict)
Base.ValueIterator for a Dict{Int64,String} with 3 entries. Values:
"Red"
"Green"
"Red"
```

A KeyIterator object is returned as an output of the keys() function, whereas the values() function outputs the ValueIterator object. They can be used to iterate over the keys using the loop structure, which will be discussed in Chapter 11.

7.14.4 Changing Values

A different value can be associated with a key using the = operator as follows:

```
julia> dict = Dict(1=>"Red",2=>"Red",3=>"Green")
Dict{Int64,String} with 3 entries:
2 => "Red"
3 => "Green"
1 => "Red"

julia> dict[2]="Blue"
"Blue"

julia> dict
Dict{Int64,String} with 3 entries:
2 => "Blue"
3 => "Green"
1 => "Red"
```

Here the value associated with key 2 is changed to the String value "Blue" and this change truly reflects the next time the dictionary is probed.

7.14.5 haskey()

Since the keys must be truly unique, the haskey() function comes in really handy because it checks if the key is present in the dictionary:

```
julia> dict = Dict(1=>"Red",2=>"Blue",3=>"Green")
Dict{Int64,String} with 3 entries:
2 => "Blue"
3 => "Green"
1 => "Red"

julia> haskey(dict,4)
false

julia> haskey(dict,3)
true
```

7.14.6 Checking a Key-Value Pair

To check if a particular key-value pair is part of a dictionary, you can use the `in` operator as follows:

```
julia> dict = Dict(1=>"Red",2=>"Blue",3=>"Green")
Dict{Int64,String} with 3 entries:
2 => "Blue"
3 => "Green"
1 => "Red"

julia> in((2=>"Red"),dict)
false

julia>  in((2=>"Blue"),dict)
true
```

7.14.7 Adding and Deleting a Key

Adding a key can be performed quite simply. In the following Julia code, a key `4` shall be associated with the value `"Orange"` and added to the dictionary. This will be reflected the next time the dictionary is printed:

```
julia> dict = Dict(1=>"Red",2=>"Blue",3=>"Green")
Dict{Int64,String} with 3 entries:
2 => "Blue"
3 => "Green"
1 => "Red"

julia> dict[4]="Orange"
"Orange"

julia> dict
Dict{Int64,String} with 4 entries:
4 => "Orange"
2 => "Blue"
3 => "Green"
1 => "Red"
```

```
julia> delete!(dict,4)
Dict{Int64,String} with 3 entries:
2 => "Blue"
3 => "Green"
1 => "Red"

julia> dict
Dict{Int64,String} with 3 entries:
2 => "Blue"
3 => "Green"
1 => "Red"
```

Using the `delete!()` function, you can delete a key from the dictionary. The exclamation mark signifies the version of function that changes the values of the input object while operating.

7.15 Summary

Arrays are the backbone of matrix computations, which has enabled the use of computers in the area of mathematics. Vectorizing a problem lets computers deal with complex tasks within a computing machine and this, in turn, lets us approximate a solution faster than achieving exact analytical solutions. Dynamically defining and manipulating arrays within a variety of data types makes Julia a good option for numerical computing. Fast operation is the key to Julia's preference in this area. Ease of defining vectorization of operations lets Julia work on arrays as matrices in a flexible manner. Effectively managing, copying, sorting, and generating arrays using comprehensions makes Julia a good choice for matrix-based mathematical methods to solve physical problems.

7.16 Bibliography

[1] http://docs.julialang.org/en/stable/

CHAPTER 8

Arrays for Matrix Operations

8.1 Defining an Array

A Julia array is equivalent to a mathematical matrix because, just like a Julia array, a matrix is an ordered collection of numbers. The simplest case for a matrix is the one storing component of a 3D vector. For example, a vector $\vec{a} = 2\hat{i} + 3\hat{j} - 4\hat{k}$ can also be represented as either a row matrix:

$$\begin{bmatrix} 2 & 3 & -4 \end{bmatrix}$$

or a column matrix:

$$\begin{bmatrix} 2 \\ 3 \\ -4 \end{bmatrix}$$

In both cases, the numbers 2, 3, and −4 are ordered in a fashion. Now this matrix can be represented by an array in Julia as follows:

```
julia> A = [2,3,-4]
3-element Array{Int64,1}:
2
3
-4
```

```
julia> size(A)
(3,)

julia> A'
1x3 Array{Int64,2}:
2  3  -4

julia> size(A')
(1,3)

julia> (A')'
3x1 Array{Int64,2}:
2
3
-4

julia> size((A')')
(3,1)
```

- `A` creates a 1D array object (having only one index).
  - This is not equivalent to a mathematical matrix as a matrix element must have at least two indices.
  - For practical purposes, this can be used as a vector.
  - This object is mostly used to represent a sequence or series of numbers.
- `A'` creates a 1 × 3 2D array object.
  - This is equivalent to a column matrix.
  - Each element has two indices, one depicting rows and the other depicting columns.
- `(A')'` creates a 3 × 1 2D array object.
  - This is equivalent to a row matrix.
  - Each element has two indices, one depicting rows and the other depicting columns.

It is important to note that Julia arrays are column majors—they are read columnwise, the same as BLAS [1] and LAPACK [2] libraries.

A Julia array doesn't have to contain only numbers. In fact, it can contain other arrays as its elements. Let's define an array having two arrays as its elements. The array named a has two elements; one of them is [1, 2, 3] and the other is [3, 4, 5]. Next, we create an array that has two a as its elements. Probing the type of elements using eltype() for a and b provides information that a has elements as an array of Int64, which is 1D in nature. On the other hand, b has a 1D array of arrays of 1D Int64.

```
julia> a = [[1,2,3],[3,4,5]]
2-element Array{Array{Int64,1},1}:
[1,2,3]
[3,4,5]

julia> b = [a,a]
2-element Array{Array{Array{Int64,1},1},1}:
Array{Int64,1}[[1,2,3],[3,4,5]]
Array{Int64,1}[[1,2,3],[3,4,5]]

julia> eltype(a)
Array{Int64,1}

julia> eltype(b)
Array{Array{Int64,1},1}
```

The importance of the comma operator , can be highlighted with the following example. A comma separates two elements of an array. When it is omitted and a whitespace character (pressing the space bar prints a whitespace character) is used instead, the elements belong to a separate column. We have already used this feature in Chapter 7 (Section 7.2).

```
julia> a = [[1,2,3],[4,5,6]]
2-element Array{Array{Int64,1},1}:
[1,2,3]
[4,5,6]
```

```
julia> b = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
1  4
2  5
3  6

julia> a = [[1,2,3], [4,5,6],[7,8,9]]
3-element Array{Array{Int64,1},1}:
[1,2,3]
[4,5,6]
[7,8,9]

julia> b = [[1,2,3] [4,5,6] [7,8,9]]
3x3 Array{Int64,2}:
1  3  6
2  4  8
3  5  9

julia> eltype(a) # elements are arrays
Array{Int64,1}

julia> eltype(b) # elements are numbers
Int64
```

8.2 Properties of a Matrix

Mathematical matrices have some properties associated with them. They can be evaluated by built-in Julia functions:

Syntax	Behavior
`det(A)`	determinant of a square matrix A
`inv(A)`	inverse of a square matrix A
`pinv(A)`	pseudo-inverse of a matrix A
`rank(A)`	rank of a matrix A

8.2.1 Determinants

Determinant of a 2 × 2 square matrix:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = (ad - bc) \tag{8.1}$$

Similarly, determinant of a 3 × 3 square matrix:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \times \begin{vmatrix} c & f \\ h & i \end{vmatrix} - b \times \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \times \begin{vmatrix} d & e \\ g & h \end{vmatrix} = aei + bfg + cdh - ceg - bdi - afh \tag{8.2}$$

In a similar manner, a bigger matrix can be solved for finding a determinant. The determinant of a square matrix can be evaluated using the command `det(A)` for an array referenced by variable named `A`.

```
julia> a = rand(3,3)
3x3 Array{Float64,2}:
0.00507492  0.305511  0.0548617
0.196032    0.444446  0.374534
0.461296    0.664772  0.260325

julia> det(a)
0.03241777361378804

julia> b = zeros(3,3)
3x3 Array{Float64,2}:
0.0  0.0  0.0
0.0  0.0  0.0
0.0  0.0  0.0

julia> det(b)
0.0
```

```
julia> c = ones(3,3)
3x3 Array{Float64,2}:
1.0  1.0  1.0
1.0  1.0  1.0
1.0  1.0  1.0

julia> det(c)
0.0
```

8.2.2 Rank

The rank of a matrix is related to the linear independence of rows/columns elements. The maximum number of linearly independent rows in a matrix A is called the row rank (R_r) of A, and the maximum number of linearly independent columns in A is called the column rank (R_c) of A. Hence, for a, $m \times n$, $R_r \leq m$. Similarly, $R_c \leq n$. Since there is no real reason to differentiate between rows and columns, $R_r = R_c = R$ (rank of matrix).

This matrix has rows and columns Number 1 and 2 as linearly dependent, which makes the rank 2:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 4 \\ 3 & 6 & 7 \end{bmatrix} \tag{8.3}$$

```
julia> A = [[1,2,3] [2,4,6] [3,4,7]]
3x3 Array{Int64,2}:
1  2  3
2  4  4
3  6  7

julia> rank(A)
2
```

8.2.3 Trace

The trace of a matrix is the sum of diagonals for a square matrix. For example, for matrix A:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 4 \\ 3 & 6 & 7 \end{bmatrix} \tag{8.4}$$

The diagonal elements are 1, 4, and 7, so their sum is 12.

```
julia> A = [[1,2,3] [2,4,6] [3,4,7]]
3x3 Array{Int64,2}:
1  2  3
2  4  4
3  6  7

julia> trace(A)
12
```

8.2.4 An Upper and Lower Triangular Matrix

`tril(A)` and `triu(A)` create a lower and upper triangular matrix from the matrix A.

```
julia> A = [[1,2,3] [2,4,6] [3,4,7]]
3x3 Array{Int64,2}:
1  2  3
2  4  4
3  6  7

julia> triu(A)
3x3 Array{Int64,2}:
1  2  3
0  4  4
0  0  7
```

```
julia> tril(A)
3x3 Array{Int64,2}:
1  0  0
2  4  0
3  6  7
```

It works in a similar manner for nonsquare matrices:

```
julia> A = rand(3,4)
3x4 Array{Float64,2}:
0.384402  0.322611  0.894988  0.839034
0.336801  0.949834  0.648842  0.0314278
0.717028  0.185107  0.684199  0.582574

julia> tril(A)
3x4 Array{Float64,2}:
0.384402  0.0       0.0       0.0
0.336801  0.949834  0.0       0.0
0.717028  0.185107  0.684199  0.0

julia> triu(A)
3x4 Array{Float64,2}:
0.384402  0.322611  0.894988  0.839034
0.0       0.949834  0.648842  0.0314278
0.0       0.0       0.684199  0.582574
```

To test if a given matrix is an upper and lower triangular matrix, the built-in function `istriu()` and `istril()` can be used:

```
julia> A = rand(3,3)
3x3 Array{Float64,2}:
0.912325  0.940698  0.768983
0.396439  0.555518  0.695407
0.961875  0.427829  0.987956
```

```
julia> triu_A = triu(A)
3x3 Array{Float64,2}:
0.912325  0.940698  0.768983
0.0       0.555518  0.695407
0.0       0.0       0.987956

julia> tril_A = tril(A)
3x3 Array{Float64,2}:
0.912325  0.0       0.0
0.396439  0.555518  0.0
0.961875  0.427829  0.987956

julia> istriu(triu_A)
true

julia> istril(tril_A)
true
```

8.2.5 Diagonal Elements

The built-in function diag(A,k) lists the diagonal elements with k as the offset for the diagonal whose positive value indicates the approaching right side and the negative value indicates the approaching left side:

```
julia> A = randn(3,4)
3x4 Array{Float64,2}:
0.171985   0.323654      -0.929096   0.237231
0.396988   0.000290637   -0.852227  -0.242657
1.52518   -0.721912      -1.40742    0.0488358

julia> diag(A)
3-element Array{Float64,1}:
0.171985
0.000290637
-1.40742
```

```
julia> diag(A,1)
3-element Array{Float64,1}:
0.323654
-0.852227
0.0488358

julia> diag(A,-1)
2-element Array{Float64,1}:
0.396988
-0.721912

julia> diag(A,-2)
1-element Array{Float64,1}:
1.52518
```

8.2.6 Norm

Following is the Euclidean norm where $a_n \in A$:

$$A = \sqrt{a_1^2 + a_2^2 + a_3^2 + \ldots}$$

The built-in function `norm()` computes the norm of a matrix. If a square complex or real matrix `A|` is given, then matrix norm `||A||` is a nonnegative number associated with `A` having the following properties:

1. `||A||` > 0 when `||A ≠ 0||` and `||A||` = 0 if $A = 0$
2. `k||A||` = `||k||||A||` for any scalar k
3. `||A + B||` ≤ `||A||` + `||B||`
4. `||AB||` ≤ `||A||||B||`

```
julia> A = [[1,2,3] [4,5,6] [7,8,9]]
3x3 Array{Int64,2}:
1  4  7
2  5  8
3  6  9
```

```
julia> norm(A)
16.84810335261421

julia> A = [[1//2,2//3,3//4] [4//5,5//6,6//7]]
3x2 Array{Rational{Int64},2}:
1//2  4//5
2//3  5//6
3//4  6//7

julia> norm(A)
1.8199543952941895

julia> a = complex(2,3)
2 + 3im

julia> b = complex(3,-2)
3 - 2im

julia> A = [[a,b] [-b,-a]]
2x2 Array{Complex{Int64},2}:
2+3im  -3+2im
3-2im  -2-3im

julia> norm(A)
5.099019513592785
```

In the case of vectors $\vec{x}$ and $\vec{y}$, the Euclidean distance =

$$norm(\vec{x}-\vec{y})$$

and the angle between them

$$= cos^{-1}\left(\frac{\vec{x}.\vec{y}}{norm(x) \times norm(y)}\right)$$

```
julia> x = [1,2,3]
3-element Array{Int64,1}:
1
2
3
```

```
julia> y = [2,4,6]
3-element Array{Int64,1}:
2
4
6

julia> dot_xy = dot(x,y)
28

julia> norm_x = norm(x)
3.7416573867739413

julia> norm_y = norm(y)
7.483314773547883

julia> angle = dot_xy/(norm_x * norm_y)
1.0

julia> (angle*180)/pi # converting to degrees
57.29577951308232
```

8.3 Matrix Operations

Matrix algebra entertains two varieties of each operation. The first one is where each element is operated upon (the operand is the element of an array). The second one is where the entire matrices are operated with each other (the operand is a matrix.) For example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} -1 & -2 \\ -3 & -4 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{8.5}$$

```
julia> a= [[1,2] [3,4]]
2x2 Array{Int64,2}:
1  3
2  4

julia> b= [[-1,-2] [-3,-4]]
2x2 Array{Int64,2}:
-1  -3
-2  -4
```

```
julia> a+b
2x2 Array{Int64,2}:
0  0
0  0
```

8.3.1 Multiplication

Addition and subtraction work in an elementwise fashion, but multiplication has many varieties:

- Scalar multiplication
  - $a \times \vec{A}$ where a is a scalar and $\vec{A}$ is a vector.
- Elementwise multiplication
  - Each element of $\vec{A}$ is multiplied by corresponding element of $\vec{B}$.
  - The shape of $\vec{A}$ and $\vec{B}$ must be identical.
- Vector multiplication
  - dot product ex. $\vec{A}.\vec{B}$
    - The shape of $\vec{A}$ and $\vec{B}$ must be identical.
  - cross product ex. $\vec{A} \times \vec{B}$
    - The inner dimension of $\vec{A}$ and $\vec{B}$ must be identical.
  - triple dot product ex. $\vec{A}.\left(\vec{B} \times \vec{C}\right)$
    - The shape of $\vec{A}$ and the resultant of $\vec{B} \times \vec{C}$ must be identical.
    - The inner dimension of $\vec{B}$ and $\vec{C}$ must be identical.
  - triple cross product ex. $\vec{A} \times \left(\vec{B} \times \vec{C}\right)$
    - The inner dimension of $\vec{B}$ and $\vec{C}$ must be identical.
    - The inner dimension of $\vec{A}$ and $\vec{B} \times C$ must be identical.

Scalar Multiplication

Scalar multiplication dictates multiplication of a scalar with each element of a matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times 2 = \begin{bmatrix} 2\times1 & 2\times2 \\ 2\times3 & 2\times4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix} \tag{8.6}$$

Julia code implementing the same can be written as follows:

```
julia> a= [[1,2] [3,4]]
2x2 Array{Int64,2}:
1  3
2  4

julia> 2*a
2x2 Array{Int64,2}:
2  6
4  8
```

This is usually accomplished as follows:

- A scalar 2 is converted to vector filled with scalar quantities with the same shape as it is multiplying with, in our case, 2 × 2:

$$\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

- Each element is multiplied elementwise with its corresponding element:

$$\begin{bmatrix} 2\times1 & 2\times2 \\ 2\times3 & 2\times4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Elementwise Multiplication

Elementwise multiplication between two matrices of the same size can be performed as follows:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 1\times2 & 2\times3 \\ 3\times4 & 4\times5 \end{bmatrix} = \begin{bmatrix} 2 & 6 \\ 12 & 20 \end{bmatrix} \tag{8.7}$$

This can be accomplished in Julia code using the elementwise multiplication operator (`.*`) as follows:

```
julia> a = [[1,3] [2,4]]
2x2 Array{Int64,2}:
1  2
3  4

julia> b = [[2,4] [3,5]]
2x2 Array{Int64,2}:
2  3
4  5

julia> a.*b
2x2 Array{Int64,2}:
2   6
12  20
```

When the shape of arrays do not match, one encounters a `DimensionMismatch` error. Hence, it is advisable that users check the dimensions of arrays (especially if they are big and/or dynamically modified during calculations) before performing this calculation.

Dot Products

The dot product of a matrix multiplies the row elements of one matrix with the column element of a second matrix and the sum all the numbers. Thus, the inner dimensions of the matrices must be identical. For example, $m \times n$ can be multiplied with $n \times p$ matrix. The result is a scalar, that is, a number:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} = (1\times1)+(1\times1)+(1\times1)+(1\times1)+(1\times1)+(1\times1)=6 \tag{8.8}$$

This can be accomplished in Julia with the following code:

```
julia> a = ones(2,3)
2x3 Array{Float64,2}:
1.0  1.0  1.0
1.0  1.0  1.0
```

```
julia> b = a'
3x2 Array{Float64,2}:
1.0  1.0
1.0  1.0
1.0  1.0

julia> vecdot(a,b)
6.0
```

Rational numbers can also be processed within this framework. The result is a rational number. The following example shows the process of working with rational numbers:

```
julia> A = [[2//3,3//4] [4//5,3//2]]
2x2 Array{Rational{Int64},2}:
2//3  4//5
3//4  3//2

julia> B = A'
2x2 Array{Rational{Int64},2}:
2//3  3//4
4//5  3//2

julia> vecdot(A,B)
701//180

julia> vecdot(B,A)
701//180
```

Similarly, complex numbers can also be used as matrix elements. The result is a complex number. The following example shows the process of working with complex numbers:

```
julia> a = complex(2,3)
2 + 3im

julia> b = complex(2,-2)
2 - 2im

julia> c = complex(-2,-2)
-2 - 2im
```

```
julia> d = complex(2,2)
2 + 2im

julia> A = [[a,b] [c,d]]
2x2 Array{Complex{Int64},2}:
2+3im  -2-2im
2-2im   2+2im

julia> B = A'
2x2 Array{Complex{Int64},2}:
2-3im  2+2im
-2+2im  2-2im

julia> vecdot(A,B)
-21 - 20im
```

Cross Product

The cross product of two matrices, say a `2 x 3` matrix named $\vec{A}$ with `3 x 2` matrix named $\vec{B}$, results in another matrix with the dimension `2 x 2`:

```
julia> A = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
1  4
2  5
3  6

julia> B = A'
2x3 Array{Int64,2}:
1  2  3
4  5  6

julia> A*B
3x3 Array{Int64,2}:
17  22  27
22  29  36
27  36  45
```

The cross product can be performed with `Rational` data types, too. In this case, the resulting matrix is composed of `Rational` data type:

```
julia> A = [[1//2,2//3,3//4] [4//5,5//6,6//7]]
3x2 Array{Rational{Int64},2}:
1//2  4//5
2//3  5//6
3//4  6//7

julia> B = A'
2x3 Array{Rational{Int64},2}:

1//2  2//3  3//4
4//5  5//6  6//7

julia> A*B
3x3 Array{Rational{Int64},2}:
89//100   1//1    297//280
1//1    41//36    17//14
297//280  17//14  1017//784

julia> eltype(A*B)
Rational{Int64}
```

Similarly, the cross product can be performed with `Complex` data types. In this case, the resulting matrix is composed of a `Complex` data type:

```
julia> a = complex(2,3)
2 + 3im

julia> b = complex(-1,2)
-1 + 2im

julia> c = complex(-2,-4)
-2 - 4im

julia> d = complex(2,4)
2 + 4im
```

```
julia> A = [[a,b,c] [b,a,c]]
3x2 Array{Complex{Int64},2}:
2+3im  -1+2im
-1+2im   2+3im
-2-4im  -2-4im

julia> B = A'
2x3 Array{Complex{Int64},2}:
2-3im  -1-2im  -2+4im
-1-2im   2-3im  -2+4im

julia> A*B
3x3 Array{Complex{Int64},2}:
18+0im    8+0im  -22-6im
8+0im   18+0im  -22-6im
-22+6im  -22+6im   40+0im

julia>  eltype(A*B)
Complex{Int64}
```

Another syntax that is used for multiplication is `*(A,B)` for arrays stored in `A` and `B`. It is equivalent to `A*B`:

```
julia> A = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
1  4
2  5
3  6

julia> *(A,A')
3x3 Array{Int64,2}:
17  22  27
22  29  36
27  36  45
```

8.4 Division

The division of two matrices is a peculiar operation. Let's first understand how this is done analytically. When matrices A and B are given, then

$$\frac{A}{B} = A \times B^{-1}$$

Consequently, it is important to understand what is the inverse of a matrix to perform division.

8.4.1 Inverse of a Matrix

The inverse of a square matrix A is such a matrix (depicted by A^{-1}) such that

$$A \times A^{-1} = 1$$

where *I* is the identity matrix. To calculate the inverse matrix from a given array, Julia uses the built-in function `inv(A)` for an array object referenced by variable name A. The usage is explained in the following code:

```
julia> A = rand(3,3) # A 3x3 matrix of random numbers
3x3 Array{Float64,2}:
0.0371386  0.382131  0.575963
0.920995   0.696674  0.897717
0.485728   0.705719  0.867646

julia> inv(A) # Inverse of matrix
3x3 Array{Float64,2}:
-0.733357   1.88985   -1.46853
-9.1587    -6.24467   12.5409
 7.85998    4.02126   -8.22573

julia> A*inv(A) # A*inv(A) = I
3x3 Array{Float64,2}:
 1.0          0.0  0.0
-8.88178e-16  1.0  0.0
-1.77636e-15  0.0  1.0
```

It is worth noting that the input array must depict a square matrix, that is, an array of dimensions $n \times n$. Julia will issue an error in the following situations:

- A is not square.
  - For a $n \times m$ matrix A, `pinv(A)` gives the pseudo-inverse:

```
julia> A = rand(2,3)
2x3 Array{Float64,2}:
0.844851  0.288378  0.568634
0.551517  0.83383   0.960742

julia> pinv(A)
3x2 Array{Float64,2}:
 1.59703   -0.753778
-0.838736   0.980266
-0.18884    0.622796

julia> A*pinv(A)
2x2 Array{Float64,2}:
1.0          -1.33206e-17
1.86956e-16   1.0
```

- A is not invertible.
  - A square matrix that is not invertible is called *singular* or *degenerate*.
  - A square matrix a is singular if |A| = 0.

```
julia> A = [[1,1] [1,1]]
2x2 Array{Int64,2}:
1  1
1  1

julia> det(A)
0.0
```

```
julia> inv(A)
ERROR: Base.LinAlg.SingularException(2)
Stacktrace:
[1] inv! at ./linalg/lu.jl:308 [inlined]
[2] inv(::Base.LinAlg.LU{Float64,
Array{Float64,2}}) at
./linalg/lu.jl:310
[3] inv(::Array{Int64,2}) at
./linalg/dense.jl:659
```

– An exception `Base.LinAlg.SingularException` is generated if Julia encounters a singular matrix.

The Inverse of a Matrix Made of Rational Numbers

The commands `inv()` and `pinv()` work well for square and nonsquare matrices, respectively, that are made of rational numbers, too:

```
julia> A = [[1//2,2//3] [3//4,4//5]]
2x2 Array{Rational{Int64},2}:
1//2  3//4
2//3  4//5

julia> inv(A)
2x2 Array{Rational{Int64},2}:
-8//1  15//2
20//3  -5//1

julia> A*inv(A)
2x2 Array{Rational{Int64},2}:
1//1  0//1
0//1  1//1

julia> A = [[1//2,2//3] [3//4,4//5] [5//6,6//7]]
2x3 Array{Rational{Int64},2}:
1//2  3//4  5//6
2//3  4//5  6//7
```

```
julia> pinv(A)
3x2 Array{Float32,2}:
-7.07495   6.77436
1.73306  -1.12989
3.88522  -3.04772
```

```
julia> A*pinv(A)
2x2 Array{Float32,2}:
1.0          -7.15256f-7
4.76837f-7   0.999999
```

The Inverse of a Matrix Made of Complex Numbers

The commands inv() and pinv() work well for square and nonsquare matrices, respectively, that are made of complex numbers, too. In most cases, instead of getting perfect zero at nondiagonal positions, we obtain extremely small numbers that can be *approximated* as 0:

```
julia> a = complex(2,3)
2 + 3im
```

```
julia> b = complex(3,4)
3 + 4im
```

```
julia> A = [[a,b] [-b,a]]
2x2 Array{Complex{Int64},2}:
2+3im  -3-4im
3+4im   2+3im
```

```
julia> inv(A)
2x2 Array{Complex{Float64},2}:
 0.0583333-0.075im        0.075-0.108333im
-0.075+0.108333im         0.0583333-0.075im
```

```
julia> A*inv(A)
2x2 Array{Complex{Float64},2}:
1.0+5.55112e-17im  1.66533e-16-2.77556e-17im
-5.55112e-17+2.77556e-17im                   1.0+0.0im
```

```
julia> A = [[a,b] [b,a] [-a,-b]]
2x3 Array{Complex{Int64},2}:
2+3im  3+4im  -2-3im
3+4im  2+3im  -3-4im

julia> pinv(A)
3x2 Array{Complex{Float64},2}:
-0.108108+0.101351im   0.141892-0.148649im
0.283784-0.297297im  -0.216216+0.202703im
0.108108-0.101351im  -0.141892+0.148649im

julia> A*pinv(A)
2x2 Array{Complex{Float64},2}:
1.0+1.38778e-16im   6.66134e-16+0.0im
0.0+1.11022e-16im   1.0-2.22045e-16im
```

8.4.2 Scalar Division

A scalar division of a matrix is elementwise division of a matrix with a scalar. For example:

```
julia> A = ones(3,2)
3x2 Array{Float64,2}:
1.0  1.0
1.0  1.0
1.0  1.0

julia> A/2
3x2 Array{Float64,2}:
0.5  0.5
0.5  0.5
0.5  0.5
```

This is similar for an array of rational number and complex numbers:

```
julia> a = [[1//2,2//3] [3//4,4//5]]
2x2 Array{Rational{Int64},2}:
1//2  3//4
2//3  4//5
```

```
julia> a/2
2x2 Array{Rational{Int64},2}:
1//4  3//8
1//3  2//5

julia> a = complex(2,3)
2 + 3im

julia> b = complex(-2,4)
-2 + 4im

julia> A = [[b,a] [-a,b]]
2x2 Array{Complex{Int64},2}:
-2+4im   -2-3im
2+3im    -2+4im

julia> A/2
2x2 Array{Complex{Float64},2}:
-1.0+2.0im    -1.0-1.5im
 1.0+1.5im    -1.0+2.0im
```

We can even make elements as rational numbers using \\ operators:

```
julia> A = rand(1:9, 2, 2)
2x2 Array{Int64,2}:

8  6
4  8

julia> A//2
2x2 Array{Rational{Int64},2}:
4//1  3//1
2//1  4//1
```

8.4.3 Left or Right Division

In the case of matrices, $A \times B \neq B \times A$. In the case of finding division, we would have two varieties:

- Left division

$$\frac{A}{B} = A^{-1} \times B$$

  - This is performed by Julia syntax `A\B`.

- Right division

$$\frac{A}{B} = A \times B^{-1}$$

  - This is performed by Julia syntax `A/B`:

```
julia> A = rand(2,2)
2x2 Array{Float64,2}:
0.0871932  0.403085
0.199973   0.611003

julia> B = rand(2,2)
2x2 Array{Float64,2}:
0.288173  0.691764
0.400971  0.457488

julia> A\B
2x2 Array{Float64,2}:
-0.528666  -8.71778
 0.829277   3.60196

julia> A/B
2x2 Array{Float64,2}:
0.836428  -0.383675
1.05474   -0.259307
```

8.4.4 Power of a Matrix

Just like numbers, matrices can be raised to a power and, just like other operators, this can be done elementwise or matrixwise. Elementwise raised to some power is simply replacing elements with new numbers after applying the operations:

```
julia> A = [[2,3] [4,5]]
2x2 Array{Int64,2}:
2  4
3  5

julia> A.^2 # elements sqaures
2x2 Array{Int64,2}:
4  16
9  25

julia> A.^0.5 # square root of elements
2x2 Array{Float64,2}:
1.41421  2.0
1.73205  2.23607

julia> A.^1//3 # elements with power 1/3
2x2 Array{Rational{Int64},2}:
2//3  4//3
1//1  5//3

julia> A.^complex(2,3) # elements raised
# to the power a complex number 2+3i
2x2 Array{Complex{Float64},2}:
-1.94798+3.49362im  -8.41077-13.611im
-8.89315-1.3827im    2.89163-24.8322im
```

On the other hand, matrix operations can also be defined in terms of power. If A is a matrix,

$$A^n = A \times A \times A \ldots (n\,times), n \in I^+ \tag{8.9}$$

$$A^{-n} = \frac{1}{A} \times \frac{1}{A} \times \frac{1}{A} \ldots (n\,times), n \in I^+ \tag{8.10}$$

$$A^0 = I \tag{8.11}$$

```
julia> A = [[2,3] [4,5]]
2x2 Array{Int64,2}:
2  4
3  5

julia> A^2 # raised to power of positive integer
2x2 Array{Int64,2}:
16  28
21  37

julia> A^(-2) # raised to power of negative integer
2x2 Array{Float64,2}:
 9.25  -7.0
-5.25   4.0

julia> A^(-2.5) # raised to power of fraction
2x2 Array{Complex{Float64},2}:
0.00211085-17.6309im  0.00371153+13.3696im
0.00278364+10.0272im   0.0048945-7.60367im

julia> A^0 # raised to power of zero
2x2 Array{Int64,2}:
1  0
0  1
```

```
julia> A^(complex(2,3)) # raised to
#power of complex number
2x2 Array{Complex{Float64},2}:
15.0873-5.16588im  26.5281-9.08321im
19.8961-6.81241im  34.9834-11.9783im
```

```
julia> A^(1//5) # raised to power of positive rational
2x2 Array{Complex{Float64},2}:
0.884712+0.317203im  0.45686-0.240537im
0.342645-0.180403im  1.22736+0.1368im
```

```
julia> A^(-2//5) # raised to power of negative rational
2x2 Array{Complex{Float64},2}:
0.49813-1.11379im   -0.034878+0.844594im
-0.0261585+0.633446im   0.471972-0.480345im
```

Square Root of a Matrix

sqrtm(A) is a dedicated built-in function for calculating the square root a matrix. Just like power operations, it requires A to be a square matrix:

```
julia> A = [[1,2] [3,4]]
2x2 Array{Int64,2}:
1  3
2  4
```

```
julia> A^(1//2) # using  power
2x2 Array{Complex{Float64},2}:
0.553689+0.464394im  1.21044-0.31864im
0.806961-0.212426im  1.76413+0.145754im
```

```
julia> sqrtm(A) # using sqrtm function
2x2 Array{Complex{Float64},2}:
0.553689+0.464394im  1.21044-0.31864im
0.806961-0.212426im  1.76413+0.145754im
```

8.4.5 Exponentiation of Matrices

The built-in function `expm(A)` computes the matrix exponential of `A`, as follows:

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!} \tag{8.12}$$

$$e^A = 1 + A + \frac{AA}{2!} + \frac{AAA}{3!} + \cdots \tag{8.13}$$

```
julia> a = [[1,2,3] [4,5,6] [7,8,9]]
3x3 Array{Int64,2}:
1  4  7
2  5  8
3  6  9

julia> expm(a)
3x3 Array{Float64,2}:
1.11891e6  2.53388e6  3.94886e6
1.37482e6  3.11342e6  4.85201e6
1.63072e6  3.69295e6  5.75517e6
```

The fact

$$e^A \times e^{-A} = I$$

can be verified using Julia code:

```
julia> a = rand(3,3)
3x3 Array{Float64,2}:
0.833095  0.295597  0.861936
0.748249  0.24969   0.63176
0.152565  0.227104  0.557303

julia> expm(a)*expm(-a)
3x3 Array{Float64,2}:
1.0          -5.55112e-17   0.0
6.93889e-17  1.0           -2.22045e-16
-2.77556e-17  5.55112e-17    1.0
```

Another fact, $e^A \times e^B = e^{A+B}$, can be verified using Julia code:

```
julia> a = rand(3,3)
3x3 Array{Float64,2}:
0.0487381  0.866547   0.00377073
0.391296   0.0638764  0.533699
0.425764   0.281759   0.626053

julia> tr_a = a'
3x3 Array{Float64,2}:
0.0487381   0.391296   0.425764
0.866547    0.0638764  0.281759
0.00377073  0.533699   0.626053

julia> sum1=(a+tr_a)
3x3 Array{Float64,2}:
0.0974763  1.25784   0.429535
1.25784    0.127753  0.815458
0.429535   0.815458  1.25211

julia> product = expm(a)*expm(tr_a)
3x3 Array{Float64,2}:
2.75234  2.44734  2.27168
2.44734  3.00784  3.09183
2.27168  3.09183  5.17031

julia> expm(sum1)
3x3 Array{Float64,2}:
2.65748  2.57752  2.3178
2.57752  3.20806  2.96962
2.3178   2.96962  5.08507
```

It is worth noting that due to numerical approximations, exact matrices might not be obtained. For example, instead of zeros in nondiagonal elements for an identity matrix, you might obtain very small numbers. Similarly, `product == expm(sum1)` would result in `false` for the previous code since the elements are not exactly equal. But they are close.

```
julia> product - expm(sum1)
3x3 Array{Float64,2}:
  0.0948575  -0.130178   -0.046114
-0.130178   -0.200223    0.122203
-0.046114    0.122203    0.0852462
```

Users are encouraged to verify more identities related to exponentiation of matrices while keeping in mind that approximations will result in inequalities where equality is expected.

8.4.6 Logarithm on Matrices

The built-in function `logm(A)` computes the logarithm of a matrix. Given the definition of exponentiation of a matrix by Equation 8.12, the logarithm of a matrix can be defined as follows:

$$e^{A} = B \Rightarrow log_{e}(B) = A \tag{8.14}$$

Matrix logarithms are not unique like logarithms of complex numbers. Furthermore, a matrix has a logarithm if and only if it is invertible.

The use of function `logm()` is explained in the following code:

```
julia> a = rand(3,3)
3x3 Array{Float64,2}:
0.202601  0.368547  0.304107
0.984077  0.77166   0.554232
0.526979  0.248144  0.534636

julia> logm(a)
3x3 Array{Complex{Float64},2}:
-1.07606+2.37668im   0.410897-0.794355im   0.382366-0.381245im
 1.15272-1.76414im  -0.52341+0.590431im    0.838622+0.281392im
 0.595192-1.09302im  0.411622+0.365746im  -0.926971+0.174484im
```

8.5 Broadcasting

When elementwise operations need to be performed on arrays of different sizes, Julia provides `broadcast()`, which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory and applies the given function elementwise. The following Julia code will make this clear where arrays `a` (size of 2 × 1) and `b` (size of 2 × 4) are multiplied using the `broadcast` function:

```
julia> a = rand(2,1)
2x1 Array{Float64,2}:
0.340869
0.864133

julia> b = rand(2,4)
2x4 Array{Float64,2}:
0.764798  0.716987  0.184377  0.483765
0.743202  0.808572  0.513173  0.839672

julia> broadcast(*,a,b)
2x4 Array{Float64,2}:
0.260696  0.244399  0.0628484  0.1649
0.642226  0.698714  0.44345    0.725588
```

8.6 Boolean Operations

Just like arithmetic operators, boolean operators can be applied to matrices. The simplest of them is the comparison of each element.

8.6.1 Comparison of Elements

Each element is compared with either a fixed value or a corresponding value of a matrix with the same size. The results are stored as a matrix made of boolean values:

```
julia> # Element-wise operations

julia> a = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
```

```
1  4
2  5
3  6

julia> b = 2*a
3x2 Array{Int64,2}:
2   8
4  10
6  12

julia> a.<b # Check if element of 'a'
# are smaller than those of 'b'
3x2  BitArray{2}:
true true
true true
true true

julia> a.== 3 # Check if elements of 'a'
# are equal to 3
3x2 BitArray{2}:
false  false
false  false
true  false

julia> a.<b & a.>3 # Checking logic statements
3x2 BitArray{2}:
false  false
false  false
false  false

julia> # Matrix operation

julia> b == (2*a) # If 'b' is two times 'a'
true
```

For matrix comparisons, the operators `<,>` do not work since their method does not include working with arrays. Hence, a `MethodError` is generated.

8.7 Concatenation

Apart from arithmetic and boolean-type mathematical operations, appending and truncating elements must also be defined to handle matrices in a flexible manner. This is achieved by a range of operators as mentioned in the following table:

Syntax	Behavior
`cat(k,A ...)`	concatenate input `n-d` arrays along the dimension k
`vcat(A)`	`cat(1,A)`
`hcat(A)`	`cat(2,A)`

```
julia> A = reshape(1:15,5,3)
5x3 Base.ReshapedArray{Int64,2,
UnitRange{Int64},Tuple{}}:
1   6  11
2   7  12
3   8  13
4   9  14
5  10  15

julia> B = reshape(15:29,5,3)
5x3 Base.ReshapedArray{Int64,2,
UnitRange{Int64},Tuple{}}:
15  20  25
16  21  26
17  22  27
18  23  28
19  24  29

julia> cat(A,B) #concatenating A and B
# along all dimesnions
5x3x1x1x1x1x1x1x1x1x1x1x1x1x1 Array{Int64,15}:
[:, :, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] =
15  20  25
16  21  26
```

```
17  22  27
18  23  28
19  24  29

julia> cat(1,A,B) # concatenating A and B row-wise
10x3 Array{Int64,2}:
1   6  11
2   7  12
3   8  13
4   9  14
5  10  15
15  20  25
16  21  26
17  22  27
18  23  28
19  24  29

julia> cat(2,A,B) # concatenating A and B coloumn-wise
5x6 Array{Int64,2}:
1   6  11  15  20  25
2   7  12  16  21  26
3   8  13  17  22  27
4   9  14  18  23  28
5  10  15  19  24  29

julia> cat(3,A,B) # Adding contents of B to new dimension
5x3x2 Array{Int64,3}:
[:, :, 1] =
1   6  11
2   7  12
3   8  13
4   9  14
5  10  15

[:, :, 2] =
15  20  25
16  21  26
```

```
17  22  27
18  23  28
19  24  29
```

A set of shortcuts for `vcat(A,B)` and `hcat(A,B)` are as follows:

Syntax	Alternate Syntax
`vcat(A,B)`	`[A;B]`
`hcat(A,B)`	`[A B]`
`hvcat(A,B,C,D)`	`A B;C D`

```
julia> A = reshape(1:15,5,3)
5x3 Base.ReshapedArray{Int64,2,UnitRange{Int64},Tuple{}}:
1   6  11
2   7  12
3   8  13
4   9  14
5  10  15

julia> [A A]
5x6 Array{Int64,2}:
1   6  11  1   6  11
2   7  12  2   7  12
3   8  13  3   8  13
4   9  14  4   9  14
5  10  15  5  10  15

julia> [A;A]
10x3 Array{Int64,2}:
1   6  11
2   7  12
3   8  13
4   9  14
5  10  15
1   6  11
2   7  12
```

```
3   8  13
4   9  14
5  10  15

julia> [A A;A A]
10x6 Array{Int64,2}:
1   6  11   1   6  11
2   7  12   2   7  12
3   8  13   3   8  13
4   9  14   4   9  14
5  10  15   5  10  15
1   6  11   1   6  11
2   7  12   2   7  12
3   8  13   3   8  13
4   9  14   4   9  14
5  10  15   5  10  15
```

The command vec() converts all matrices into a 1D matrix:

```
julia> A = rand(2,3,2)
2x3x2 Array{Float64,3}:
[:, :, 1] =
0.293696  0.336827  0.252549
0.999608  0.17789   0.718892

[:, :, 2] =
0.958808  0.408669  0.950778
0.996035  0.533242  0.310243

julia> vec(A)
12-element Array{Float64,1}:
0.293696
0.999608
0.336827
0.17789
0.252549
0.718892
0.958808
```

```
0.996035
0.408669
0.533242
0.950778
0.310243
```

8.7.1 repmat()

Given a matrix, if we wish to construct another matrix by repeating elements of the original matrix, `repmat()` comes in handy. The syntax of `repmat(A, n, m)`, which constructs a matrix by repeating `A` `n` times in dimension number 1 (rows) and `m` times in dimension number 2 (columns):

```
julia> A = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
1  4
2  5
3  6

julia> repmat(A,2,2)
6x4 Array{Int64,2}:
1  4  1  4
2  5  2  5
3  6  3  6
1  4  1  4
2  5  2  5
3  6  3  6

julia> repmat(A,2,3)
6x6 Array{Int64,2}:
1  4  1  4  1  4
2  5  2  5  2  5
3  6  3  6  3  6
1  4  1  4  1  4
2  5  2  5  2  5
```

8.7.2 repeat()

`repeat(A,inner,outer)` constructs an array by repeating the entries of `A`. The i_i element of `inner` specifies the number of times that the individual entries of the i_i dimension of `A` should be repeated. Similarly, the i_i element of outer specifies the number of times that a slice along the i_i dimension of `A` should be repeated. When inner or outer are not provided, repetitions are not performed:

```
julia> a = collect(2:4)
3-element Array{Int64,1}:
2
3
4

julia> repeat(a,inner=2)
6-element  Array{Int64,1}:
2
2
3
3
4
4

julia> repeat(a,outer=2)
6-element Array{Int64,1}:
2
3
4
2
3
4

julia> repeat(a,inner=2,outer=2)
12-element Array{Int64,1}:
2
2
3
3
```

```
4
4
2
2
3
3
4
4

julia> a = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
1  4
2  5
3  6

julia> repeat(a,inner=(1,3),outer=(3,1))
9x6 Array{Int64,2}:
1  1  1  4  4  4
2  2  2  5  5  5
3  3  3  6  6  6
1  1  1  4  4  4
2  2  2  5  5  5
3  3  3  6  6  6
1  1  1  4  4  4
2  2  2  5  5  5
3  3  3  6  6  6

julia> repeat(a,inner=(2,3),outer=(3,2))
18x12 Array{Int64,2}:
1  1  1  4  4  4  1  1  1  4  4  4
1  1  1  4  4  4  1  1  1  4  4  4
2  2  2  5  5  5  2  2  2  5  5  5
2  2  2  5  5  5  2  2  2  5  5  5
3  3  3  6  6  6  3  3  3  6  6  6
3  3  3  6  6  6  3  3  3  6  6  6
1  1  1  4  4  4  1  1  1  4  4  4
1  1  1  4  4  4  1  1  1  4  4  4
```

```
2  2  2  5  5  5  2  2  2  5  5  5
2  2  2  5  5  5  2  2  2  5  5  5
3  3  3  6  6  6  3  3  3  6  6  6
3  3  3  6  6  6  3  3  3  6  6  6
1  1  1  4  4  4  1  1  1  4  4  4
1  1  1  4  4  4  1  1  1  4  4  4
2  2  2  5  5  5  2  2  2  5  5  5
2  2  2  5  5  5  2  2  2  5  5  5
3  3  3  6  6  6  3  3  3  6  6  6
3  3  3  6  6  6  3  3  3  6  6  6
```

8.8 Rotating a Matrix

Rotating a matrix `A` by 180 degrees can be performed by the built-in function `rot180(A,n)` where `n` is the integer number of times the rotation needs to be performed. If *n* is an even number, the action is equivalent to `copy()`:

```
julia> a = [1 2 3 4 5 6 7 8]
1x8 Array{Int64,2}:
1  2  3  4  5  6  7  8

julia> a1 = reshape(a,(2,4))
2x4 Array{Int64,2}:
1  3  5  7
2  4  6  8

julia> rot180(a1) # defualt n=1
2x4 Array{Int64,2}:
8  6  4  2
7  5  3  1

julia> rot180(a1,1) # 1 rotation
# in forward direction
2x4 Array{Int64,2}:
8  6  4  2
7  5  3  1
```

```
julia> rot180(a1,-1) # 1 rotation
# in backward direction
2x4 Array{Int64,2}:
8  6  4  2
7  5  3  1

julia> rot180(a1,2) # 2 totations
# in forward direction
2x4 Array{Int64,2}:
1  3  5  7
2  4  6  8
```

8.9 Special Matrix

8.9.1 Symmetric Matrices

A symmetric matrix is a square matrix that is equal to its transpose. For example:

$$A = A'$$

The Julia function `issymmetric(A)` tests if array `A` represents a symmetric matrix and gives a boolean output `true` or `false`. The entries of a symmetric matrix are symmetric with respect to the main diagonal. So if the entries are written as $A = (a_{ij})$, then $a_{ij} = a_{ji}$, for all indices i and j. For this reason, every square diagonal matrix is symmetric since all off-diagonal elements are zero:

```
julia> A = rand(3,3)
3x3 Array{Float64,2}:
0.494451  0.65293   0.801365
0.775357  0.963112  0.535383
0.138436  0.206775  0.845183

julia> issymmetric(A)
false

julia> A = [[1,0] [0,1]]
2x2 Array{Int64,2}:
1  0
0  1
```

```
julia> issymmetric(A)
true

julia> A = [[1,7,3] [7,4,-5] [3,-5,6]]
3x3 Array{Int64,2}:
1   7   3
7   4  -5
3  -5   6

julia> issymmetric(A)
true
```

8.9.2 Positive Definite Matrix

A symmetric real matrix A is said to be positive definite if ∃*z* (a scalar) such that

$$z'Az > 0$$

is positive for every nonzero column vector *z* of *n* real numbers. For example:

$$\begin{bmatrix} a & b \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} a \\ b \end{bmatrix} = a^2 + b^2 > 0 \tag{8.15}$$

```
julia> A = [[2,-1,0] [-1,2,-1] [0,-1,2]]
3x3 Array{Int64,2}:
 2  -1   0
-1   2  -1
 0  -1   2

julia> b = [1,2,3]
3-element Array{Int64,1}:
1
2
3

julia> b'*A*b
1-element Array{Int64,1}:
12
```

```
julia> isposdef(A)
true
```

But this does not work with another matrix:

```
julia> A = [[1,7,3] [7,4,-5] [3,-5,6]]
3x3 Array{Int64,2}:
1   7   3
7   4  -5
3  -5   6

julia> b = [1,2,3]
3-element Array{Int64,1}:
1
2
3

julia> b'*A*b
1-element Array{Int64,1}:
57

julia> isposdef(A)
false
```

8.9.3 Hermitian Matrices

Hermitian is a complex square matrix that is equal to its own conjugate transpose. In other words, *i*th row and *j*th column are equal to the complex conjugate of the element in the *j*th row and *i*th column for all indices *i* and *j*:

$$a_{ij} = \bar{a}_{ji} \tag{8.16}$$

```
julia> A = [[2,2+im,4] [2-im,3,im] [4,-im,1]]
3x3 Array{Complex{Int64},2}:
2+0im  2-1im  4+0im
2+1im  3+0im  0-1im
4+0im  0+1im  1+0im

julia> ishermitian(A)
true
```

8.9.4 Sparse Matrices

When a matrix has a large number of 0 as its elements, storing them is a waste of precious computer memory and is an inefficient way of computing for in terms of time and computational resources Such matrices are called sparse matrices. For efficient computing framework, sparse matrices are stored in the *Compressed Sparse Column (CSC)* format.

The built-in function `speye()` creates a sparse matrix of a given dimension. Its inputs are vectors for denoting indices of rows and columns and a third vector denoting the nonzero values:

```
julia> a = [1,4,3,5,8,3];

julia> b = [4,7,18,9,7,3];

julia> c = [1,2,-5,3,-100,0.5];

julia> s = sparse(a,b,c)
8x18 sparse matrix with 6 Float64 nonzero entries:
[3 ,  3]  =     0.5
[1 ,  4]  =     1.0
[4 ,  7]  =     2.0
[8 ,  7]  =  -100.0
[5 ,  9]  =     3.0
[3 , 18]  =    -5.0

julia> findn(s)
([3,1,4,8,5,3],[3,4,7,7,9,18])

julia> findnz(s)
([3,1,4,8,5,3],[3,4,7,7,9,18],[0.5,1.0,2.0,-100.0,3.0,-5.0])
```

The vectors `a` and `b` contribute to making the indices for nonzero elements and the values of these nonzero elements are given by the third vector `c`. The dimension of the sparse matrix is obviously the maximum values of `a` and `b`. The function `findn()` finds the indices for the rows and columns of a sparse matrix as two separate arrays. Another function `findnz()` finds the same plus the nonzero values as a third array.

8.10 Summary

It is now clear that arrays can be used to define matrices. Matrix algebra is encoded in a way the arrays can be manipulated. Vectorized versions of operations and corresponding nonvectorized matrix operations can be executed with good speed. Data crunching involves a flexible manner in which arrays can be defined as matrices and mathematical operations can be done in quickly. Julia provides an upper hand in this arena and is fast becoming favorite option of data analytics.

8.11 Bibliography

[1] www.netlib.org/blas/

[2] www.netlib.org/lapack/

CHAPTER 9

Strings

9.1 Introduction

The handling of text-based data is an important feature of all programming frameworks. *Strings* are simply defined as a *set of characters*. These include characters and words (group of characters) made up of the following:

- Uppercase alphabets, for example, `A,B,C ...`
- Lowercase alphabets, for example, `a,b,c ...`
- Hindu-Arabic numerals, for example, `1,2,3 ...`
- Some special symbols, for example, `!,@,#,$,%,^,&,*`

They can be found on most English-language-based keyboards. What about other languages? They must also be included within a computational framework. However, natural languages that humans use are not the preferred language of computation in computer science. Computers, instead, use the language of binary numbers, where all entities are defined as a group of bytes made up of two bits, either 1 or 0. Hence, these characters and their groups must be mapped with binary numbers within a specific protocol that must be internationally accepted. Thus, the ASCII and Unicode systems were developed.

9.2 ASCII System

The ASCII [1] system for characters maps English characters, numbers (as characters), and some special characters to integer values between 0 and 127. These 128 sets of combinations encompasss most of the required characters for English-related work. A 7 (2^6 – 128) bits could store a unique ASCII character.

The ASCII system works quite well but is limited in scope. As more nations joined the computing community, more language symbols needed to be incorporated into computing. A range of mathematical symbols also needed to be incorporated.

9.3 Unicode System

The Unicode system [2] system is an extension of ASCII that increases the number of bytes for storing a character and, thus, increases the number of characters that can be uniquely defined. This system incorporates many languages and special symbols for mathematical notations. Julia supports the Unicode definition of characters, meaning that they can be used just like any other character while computing. This is a great advantage for the mathematical environment as straightforward usage of mathematical symbols makes it easy to understand. For example, π can be written as the symbol itself, rather than as `pi` as a pnuemonics for the symbol. Those familiar with LATeX formulation would understand that these symbols are written by proceeding their command by the `\` operator; the same is done in Julia. To write π, you write `\pi` and press the Tab key on the keyboard. This results in displaying the Unicode symbol π.

Now let's look at how Julia understands and interprets textual information. We will start with characters and then graduate to groups of characters called strings. The characters are primarily fed using the keyboard, but they can originate from a file both within a machine and from outside-world interfacing instrument(s). Julia provides versatile capabilities to deal with all the facilities with respect to handling strings.

9.4 Characters

Since Julia is an object-oriented programming language, characters must also be defined as objects. The Julian data type for characters is `Char`. Let's first understand characters with an example:

```
julia> a = 'a'
'a'

julia> typeof(a)
Char
```

Here a variable references to a memory location storing an object of the type Char whose value is a. Incidentally, the name of the variable is also a. While a means a reference name to Julia, 'a' refers to a Char (character) object.

9.5 Corresponding Integer Value

Since a character is stored as a set of binary digits, these binary digits can be interpreted as numbers. Thus, each character has a corresponding integer value. This can be illustrated by defining the character as an integer object as follows:

```
julia> Int32('a')
97

julia> Int32('z')
122

julia> Int32('!')
33

julia> Int32('#')
35
```

The reverse is also true; integers also correspond to a particular character:

```
julia> Char(121)
'y'

julia> Int32('y')
121
```

All integer values are not valid Unicode characters. The valid Unicode code points (in hexadecimal digits) from *U+00 - U+d7ff* and *U+e000 - U+10ffff*. All of these numbers have not been assigned intelligible meaning yet, but are valid Unicode characters. Julia uses a machine's locale and language settings to determine characters that must be printed.

Since integers are associated with characters, they can be used with some arithmetic operators. For example, one can calculate 'a'-'b' and Int32('a') - Int32('b') and find that they are similar since `'a'` and `'b'` correspond to an integer value (given by `Int32('a')` and `Int32('b')`, respectively):

```
julia> 'a' - 'b'
-1

julia> Int32('a') - Int32('b')
-1

julia>  Int32('a')
97

julia>  Int32('b')
98

julia> 'A' == 'a' # Capitalized alphabets
# hold different ineteger values than
#  small  alphabets
false

julia> 'A' <    'a'
true

julia> Int32('A')
65

julia> 'A'+1
'B': ASCII/Unicode U+0042
(category Lu: Letter, uppercase)

julia> 'A'+2
'C': ASCII/Unicode U+0043
(category Lu: Letter, uppercase)

julia> 'A'+58 # results corresponds to
# integer value corresponding to the
# symbol "}"
'{': ASCII/Unicode U+007b
(category Ps: Punctuation, open)
```

When we input commands like `'A'+1`, we obtain a character as a result that corresponds to the integer value that, in turn, corresponds to the result of the calculation. Since `'A'` corresponds to integer value 65, adding one to it results in 66, which corresponds to `'B'`. The output also displays additional information—ASCII/Unicode U+0042 (category Lu: Letter, uppercase). This defines that the output is an ASCII/Unicode object whose category is Lu—a letter that is defined as uppercase.

9.6 + Operator and Characters

What happens when we concatenate two `Char` objects? In most programming languages, the + works like a concatenation operator for characters and strings. When a character a and `!` need to be made into a string `a!`, we usually write `'a'+'!'` or `+('a','!')`. Let's check if this can be done in Julia:

```
julia> char1 = 'a' # definig first character
'a': ASCII/Unicode U+0061
(category Ll: Letter, lowercase)

julia> char2 = '!' # defining second character
'!': ASCII/Unicode U+0021
(category Po: Punctuation, other)

julia> typeof(char1) # verifying type of object
Char

julia> typeof(char2) # verifying type of object
Char

julia> +(char1,char2) # Operator + operated on char1 and char2
ERROR: MethodError: no method matching +(::Char, ::Char)
Closest candidates  are:
+(::Any, ::Any, ::Any, ::Any...) at operators.jl:424
+(::Char, ::Integer) at char.jl:40
+(::Integer, ::Char) at char.jl:41
```

As seen in the previous Julia code, we obtain a `MethodError` since the concatenation operator does not handle `Char` objects. All operators are defined as Julia functions (see Chapter 10). A method is a function associated with an object to probe its property. Julia has a feature—multiple dispatch—which enables different functional definitions as per data type. So while the + function recognizes objects like `Int64` and `Complex64`, it does not recognize `Char` because this was not defined in its source code. Hence, two characters cannot be concatenated using the + operator.

If strings are sets of characters, then how will characters make strings if concatenation is not allowed? We will explore this idea in the next section.

9.6.1 Characters and Strings Are Two Data Types

Julia defines a character using single quotes. String definitions need double or triple quotes enclosing a single character or a set of characters:

```
julia> a1 = 'a' # character 'a' is referenced by a1
'a'

julia> typeof(a1) # type of a1 is Char
Char

julia> a2 = "a" # String "a" is referenced by a2
"a"

julia> typeof(a2) # type of a2 is String
String

julia> a2 == a1 # a2 and a1 are not equal since
# they store different objects
false
```

Triple quotes are used in those cases when we wish to print a single quote or double quote as part of the string:

```
julia> """You've been "warned" alread !Don't repeat!"""
"You've been \"warned\" alread !Don't repeat!"
```

9.7 + Operator and Strings

Code line `a2 == a1` is important to understand. Even though the value of the `Char` and `String` objects use the same alphabet, namely 'a', they are not actually the same. Characters and strings are quite different in their behavior. Can strings be concatenated using + operator?

```
julia> a1 = "Hello "
"Hello "

julia> a2 = "world"
"world"

julia> a1+a2
ERROR: MethodError: no
method matching +(::String, ::String)
Closest candidates are:
+(::Any, ::Any, ::Any, ::Any...)
at operators.jl:424
```

We can now see that the + operator does not concatenate strings. Julia must provide an alternative way to add or cut elements from strings.

9.8 Concatenation

Concatenation of strings is performed using the function `string()`, which concatenates multiple strings that are separated by the `,` separator:

```
julia> string('a','!') #  inputs are  characetrs
"a!"

julia> string("a","!") #  inputs are strings
"a!"

julia> string("a",'!') # inputs are stings and character
"a!"

julia> string("a",'!',' ', "wow", "#") # Third character
# is a whitespace
"a! wow#"
```

Another example will make it clearer:

```
julia> h = "hello"
"hello"

julia> w = "world"
"world"

julia> ws = " "# A white space as a string
" "

julia> str = string(h,ws,w)
"hello world!"
```

Alternatively, the * operator also performs concatenation actions:

```
julia> "a"*"!"
"a!"

julia> *("a","!")
"a!"

julia> h = "hello"
"hello"

julia> w  =  "world"
"world"

julia> ws = " "
" "

julia> *(h,ws,w)
"hello world"
```

9.9 Interpolation

If variables store some values (in this case, a `String` object), their verbose calls are performed as follows:

```
julia> h = "hello"
"hello"
```

```
julia> ws = " "
" "

julia> e = "!"
"!"

julia> "$h$ws$e"
"hello !"
```

This is a better way of concatenation as it's more convenient, particularly in the case when we need to fill the value of a variable inside a string as output:

```
julia> a = 10
10

julia> int = "Integer"
"Integer"

julia> b = 10.0
10.0

julia> float_number = "Floating point number"
"Floating point number"

julia> "$a is stored as $int"
"10 is stored as Integer"

julia> "$b is stored as $float_number"
"10.0 is stored as Floating point number"
```

What if we need to print the character $ itself? We then should precede it with a backlash character in this case:

```
julia> statement = "I have "
"I have "

julia> currency = "US Dollars"
"US Dollars"
```

```
julia> value = 100
100

julia> print("$statement \$100 ($currency) in my account")
I have  $100 (US Dollars) in my account
```

9.10 Strings Are Like Arrays

Just like arrays, a string's characters are indexed. The index in Julia always start from 1 and the last index can be accessed by using end. This can be understood in the following example:

```
julia> str = """You've been "warned" alread !Don't repeat!"""
"You've been \"warned\" alread !Don't repeat!"

julia> str[1]
'Y'

julia> str[end]
'!'

julia> str[end-20]
'a'

julia> length(str)
42

julia> str[10:20]
"en \"warned\""

julia> str[21:end]
alread !Don't repeat!"
```

The variable `str` references to the defined string. The following operations are performed successively in the previous code:

- `str[1]` outputs the first character of the string object.
- `str[end]` outputs the last character of the string.
- `str[end-20]` outputs the twentieth character from the end.

- The total length of string can be calculated using `length(str)`.
- Using an index less than 1 or greater than `end` raises a `BoundsError`.
- Slicing can be performed using the `:` operator. For example, `str[10:20]` outputs from the tenth character to the twentieth character. Similarly, `str[21:end]` outputs from the twenty-first character to the end of the string (the last character).

What should we expect if we write `str[10]` and `str[10:10]`? 10*th* character is `e`:

```
julia> str[10]
'e'

julia> str[10:10]
"e"

julia> typeof(str[10])
Char

julia> typeof(str[10:10])
String

julia> str[10:10] == str[10]
false
```

It is observed that the output of `str[10]` is an object of `Char` type, whereas the output of `str[10:10]` is an object of `String`. Even if their values are same, they are not similar objects. As a result, the equality operator == shows `false` as its output.

9.10.1 search()

The built-in function `search()` can be used to search for the index of a particular character in a string. The first argument is the string that needs to be probed, and the second argument is the character/string that needs to be probed. In the case of searching a character, the output is the index number. In the case of searching a string, the output is the range object. If it does not find the input character in the input string, the output is `0`. If it does not finds the input string within the given string, it outputs a range object `0:-1` signifying that the input string cannot be found in any given string. This is demonstrated in the following Julia code:

```
julia> search("Sandeep Nagar", 'N')
9

julia> search("Sandeep Nagar", 's')
0

julia> search("Sandeep Nagar", 'a',3) # offset by 3
10

search("Sandeep Nagar", "Sandeep")
1:7

julia> search("Sandeep Nagar", "sandeep")
0:-1

julia> search("Sandeep Nagar", "randeep")
0:-1
```

9.10.2 contains()

The built-in function `contains()` can be used to test if a particular character or string is contained inside a test string. The difference of output when compared to the `search()` function is that the output of the `contains()` function is a boolean object. (Either it is `true` if the input string is found, or it is `false` for the other case.) Also, this function works only with `String` objects, not with `Char` objects, and hence throws a `MethodError`. Thus, if a character needs to be searched (in the previous case `S`), it must be input as a string (`"S"`). This is shown in the following Julia code:

```
julia> contains("Sandeep Nagar", "Sandeep")
true

julia> contains("Sandeep Nagar", " ")
true

julia> contains("Sandeep Nagar", "S")
true
```

```
julia> contains("Sandeep Nagar",'S')
ERROR: MethodError: no method matching contains
(::String, ::Char)
Closest candidates are:
contains(::Function, ::Any, ::Any)
at reduce.jl:664
contains(::AbstractString, ::AbstractString)
at strings/search.jl:378
```

An alternate way is to use the member function `in()` as follows:

```
julia> in('S',"Sandeep Nagar")
true
```

9.11 Common String Functions

A variety of string functions can operate on strings to perform specific tasks. Some of them are discussed in the following sections.

9.11.1 repeat()

A string can be repeated a specific number of times by using the `repeat()` function:

```
julia> repeat("Hi",3)
"HiHiHi"
```

Alternatively, the ^ operator also performs the same job. This is quite obvious mathematically too since power is merely successive multiplication; for example, $2^3 = 2 \times 2 \times 2$. Since the operator * is used for concatenation, ^ performs successive operations of similar nature for a specified number of times:

```
julia> "Hi"^3
"HiHiHi"
```

```
julia> ^("Hi",3)
"HiHiHi"
```

9.11.2 join()

The built-in function `join(io, items, delim, [last])` prints elements of items to `io` with `delim` (delimiter) between them. If `last` is specified, it is used as the final delimiter instead of `delim`:

```
julia> a = join(["Beginner","Intermediate","Advanced"],","," and ")
"Beginner,Intermediate and Advanced"
```

```
julia> println("""Three stages of julia learner are \n $a""")
Three stages of julia learner are
Beginner,Intermediate and Advanced
```

9.11.3 start(), endof(), and next()

The built-in function `start()` gives the first valid index. This is typically 1. The built-in function `endof()` gives a maximal (byte) index that can be used to index. Another built-in function, `next()`, returns the next character at or after the index `i` and the next valid character index following that. We encounter a `BoundsError` if we attempt to probe beyond the maximum index found in the input string:

```
julia> start("Sandeep Nagar")
1
```

```
julia> endof("Sandeep Nagar")
13
```

```
julia> length("Sandeep Nagar")
13
```

```
julia> a,b = next("Sandeep Nagar",5)
('e', 6)
```

```
julia> a,b = next("Sandeep Nagar",13)
('r', 14)
```

```
julia> a,b = next("Sandeep Nagar",14)
ERROR: BoundsError: attempt to access
"Sandeep Nagar" at index [14]
Stacktrace:
[1] next(::String, ::Int64) at ./strings/string.jl:197
```

These functions can be used to iterate over strings using loop structures. (See Chapter 11.)

9.11.4 split()

The built-in function `split()` takes an input as a `String` object and returns an `Array` object where the elements are individual `String` objects for each word, that is, an array of substrings:

```
julia> str = String("Hi, How are you")
"Hi, How are you"

julia> split(str)
4-element  Array{SubString{String},1}:
"Hi,"
"How"
"are"
"you"
```

If substrings needs to be made exactly at the occurrence of a specified character, then an additional argument can be inserted. For example:

```
julia> str = String("Hi, How are you")

julia> split(str,"w")
2-element Array{SubString{String},1}:
"Hi, Ho"
"are you"
```

```
julia> split(str,"o")
3-element Array{SubString{String},1}:
"Hi, H"
"w are y"
"u"

julia> split(str,"are")
2-element Array{SubString{String},1}:
"Hi, How "
" you"

julia> split(str,"ow")
2-element Array{SubString{String},1}:
"Hi, H"
" are you"

julia> split(str,"")
15-element Array{SubString{String},1}:
"H"
"i"
","
" "
"H"
"o"
"w"
" "
"a"
"r"
"e"
" "
"y"
"o"
"u"
```

As seen in the prevous example, if an empty string is used for splitting, then each character makes the substring.

9.11.5 uppercase() and lowercase()

The built-in functions `uppercase()` and `lowercase()` convert the characters of a string to uppercase or lowercase characters, respectively.

```
julia> uppercase("Sandeep Nagar")
"SANDEEP NAGAR"
```

```
julia> lowercase("Sandeep Nagar")
"sandeep nagar"
```

In the case of `uppercase()` functions on the string `Sandeep Nagar`, all characters of the string get converted to uppercase characters. Those that are already uppercase remain so. Similar actions happen for `lowercase()` functions where lowercase characters remain so after operation.

9.11.6 replace()

The built-in function `replace()` returns a new string with a substring of characters replaced with something else:

```
julia> name = "Sandip Nagar"
"Sandip Nagar"
```

```
julia> replace(name,"i","ee") # replace "i" with "ee"
"Sandeep Nagar"
```

```
julia> replace("sandeep","e",uppercase)
"sandEEp"
```

As seen in the last command, a function can also be supplied to the `replace()` function to output a string in a desired form replacing the original string.

9.11.7 lpad() and rpad()

Padding a string from the left and right side with a specific character for a specific number of times can be done using `lpad()` and `rapd()`:

```
julia> name = "Sandeep Nagar"
"Sandeep Nagar"
```

```
julia> length(name)
13

julia> lpad("Sandeep Nagar",15,"a")
"aaSandeep Nagar"

julia> rpad("Sandeep Nagar",15,"a")
"Sandeep Nagaraa"
```

In the following section, the variable name references to a `String` object that has length of 13 characters. When `lpad(name,15,"a")` is used, the character a is padded from the left two times to make the desired length of 15. A similar task is accomplished by the `rpad()` function but from the right-hand side.

9.11.8 reverse()

The built-in function `reverse()` reverses a string:

```
julia> reverse("Sandeep Nagar")
"ragaN peednaS"
```

9.11.9 strip(), lstrip(), and rstrip()

Stripping a string from undesirable characters is an important function while dealing with strings in a programmatic way. The built-in function `strip()` performs the same:

```
julia> name = " Sandeep Nagar "
# Two white spaces at start and end of string
Sandeep Nagar "

julia> length(name)
15

julia> strip(name)
# Two white spaces (start and end) have been stripped
"Sandeep Nagar"

julia> lstrip(name)
# One white space (left) is stripped
"Sandeep Nagar "
```

```
julia> rstrip(name)
# One white space (right) is stripped
" Sandeep Nagar"
```

9.11.10 randstring()

Creating a random string is as important while testing the code as creating a single or set of random numbers. This can be accomplished by the built-in function `randstring()` as follows:

```
julia> randstring(20)
"8bXPxczPGOv2EDweJBtX"

julia> randstring(20)
"3BMgodnhVbgVQnW6h9EO"

julia> randstring(20)
"nMDC8x4yc8UYMsNPdQrx"

julia> randstring(20)
"cZDVojcv2RYjzyZCLJ6B"
```

Please note that the previous example shows that a different string is obtained each time the same command, `randstring()`, is used, verifying its random nature. Users might obtain different sets of strings from those mentioned for the same reason.

9.12 Reading Data as Arrays from Strings

Sometimes the data may be formatted as a `string` object. For mathematical manipulation, this data must be converted as an element of an array. To read from a string into an array, you can use the `IOBuffer()` function, which creates a read-only `IOBuffer` on the data underlying the given string. To understand this object, try exploring the same using `help?> IOBuffer`. (These `I-O` (Input- Output) objects are discussed in Chapter 12. Hence, the details are not discussed here, but the primary usage is demonstrated for sake of simplicity.)

```
julia> str = "1 2 3 4 5 6 7 8 9" # a string object
# where entries are separated by white space
"1 2 3 4 5 6 7 8 9"

julia> a = IOBuffer(str) # IOBuffer object
IOBuffer(data=UInt8[...],
readable=true,
writable=false,
seekable=true,
append=false,
size=17,
maxsize=Inf,
ptr=1,
mark=-1)

julia> readdlm(a) # Read with delimiter
1x9 Array{Float64,2}:
1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0

julia> str = "1,2,3,4,5,6,7,8,9" #  String where
#entities are separated by commas
"1,2,3,4,5,6,7,8,9"

julia> a = IOBuffer(str)
IOBuffer(data=UInt8[...],
readable=true,
writable=false,
seekable=true,
append=false,
size=17,
maxsize=Inf,
ptr=1,
mark=-1)

julia> readdlm(a) # No white space, no columns
# Also commas are part of element
# So element is not numeric type
1x1 Array{Any,2}:
"1,2,3,4,5,6,7,8,9"
```

The function readdlm() (read with delimiter) converts this IOBuffer object into an array object. The columns are assumed to be separated by one or more whitespace characters. If all data are numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned with data type Any.

When the data is made of string objects, an array of substrings is created as follows:

```
julia> str = "Hi How are you"
"Hi How are you"

julia> a = IOBuffer(str)
IOBuffer(data=UInt8[...],
readable=true,
writable=false,
seekable=true,
append=false,
size=14,
maxsize=Inf,
ptr=1,
mark=-1)

julia>  readdlm(a)
1x4 Array{Any,2}:
"Hi"  "How"  "are"  "you"
```

Since the array is composed of Any object, it does not pose a problem if data are made of a mixture of strings and numerals.

```
julia> str = "Hi How are you number 1"
"Hi How are you number 1"

julia> a = IOBuffer(str)
IOBuffer(data=UInt8[...],
readable=true,
writable=false,
seekable=true,
append=false,
size=23,
maxsize=Inf,
ptr=1,
mark=-1)

julia> readdlm(a)
1x6 Array{Any,2}:
"Hi"  "How"  "are"  "you"  "number"  1
```

9.13 Lexicographical Comparison of Strings

Standard comparison operators compare strings by lexicographical rules [3]. A lexicographical comparison is the kind of comparison generally used to sort words alphabetically in dictionaries. Thus, it is sometimes called dictionary order.

Lexicographical comparison involves comparing sequentially the elements that have the same position in both ranges against each other until one element is not equivalent to the other. The result of comparing these first nonmatching elements is the result of the lexicographical comparison:

```
julia> str1 = "abcdefg"
"abcdefg"

julia> str2 = "abcdefh"
"abcdefh"

julia> str1 < str2
true

julia> str3 = "abcdefhi"
"abcdefhi"

julia> str3 > str2
true

julia> str4 > str3
false
```

```
julia> "20 March 2017" > "19 March 2017"
true

julia> "20 March 2016" > "19 March 2017"
true

julia> "superhero" == "Superhero"
false

julia> "superhero" > "Superhero"
true
```

9.14 Summary

The ability to deal with textual data is an important feature of Julia. To understand how characters are defined and dealt with in a computing machine, check out Chapter 12, Section 12.6. Dealing with characters in a programmable way lets a computational linguist explore tasks under natural language processing (NLP) and derive meaningful patterns within human and nonhuman languages. Julia's ability to define a string and flexibly manipulate it in a variety of manners makes it a good candidate for NLP. The field of bio-informatics demands these abilities as well. As a result, Julia is a good candidate to define code under these domains for faster execution.

9.15 Bibliography

[1] https://en.wikipedia.org/wiki/ASCII#ASCII_control_characters.

[2] https://en.wikipedia.org/wiki/List_of_Unicode_characters.

[3] https://en.wikipedia.org/wiki/Lexicographical_order.

CHAPTER 10

Functions

10.1 Introduction to Julia Functions

A procedural programming language involves breaking down a bigger program into smaller chunks of functional blocks and then stitching them together as desired for the task undertaken. A block of code performs a specific task when called by the master program. This block of code is defined as a *function*. A function maps its *input* to *output* according to the set definition as dictated by a (set of) statements called the *body* of function. This definition is quite similar to the definition of a mathematical function.

The function object is referenced by a name that points to memory location where function `object` is stored. When a function needs to be called, the name, along with a set of inputs (in parentheses), is called during the execution of the program.

Alternatively, a function can also be defined as a first-class object that inputs an argument list (`arglist`), processes the list of arguments as per definition of function `body`, and returns none, one, or more values as outputs. Multiple arguments form a tuple. So does the output if it is made up of multiple entities. For this reason, they are separated by commas. The *type* of arguments can be set or the arguments can be determined using the kind of operations in the function `body`. It is recommended to set the type for optimized utilization of computational resources. However, in this chapter, we will ignore this recommendation for ease of understanding.

The syntax of a function is as follows:

```
julia> function fname(arglist)
                   #function body...
                   return values
         end
```

The tabbing of a body part is optional, but it makes the appearance neater for visual clarity. Hence, it is strongly recommended. Function names are usually defined in lowercase characters as a convention. They can contain Unicode characters, too. The `return keyword` is optional. In general, the value of the last expression is returned. While calling a function, the name of the function is written with the `arglist` within parentheses. If an assignment operator is used for the same (`a = fname(arglist)`), then the return value is assigned to `a`. When the return is used without a value, the function returns nothing; it just does a calculation but returns nothing.

10.2 Defining a Simple Julia Function

Let's study Julia functions using some examples. Consider a function for calculating the value of the hypotenuse of a right-angled triangle with the dimensions of its perpendicular (say `p`) and base (say `b`) given. Let's name the function `f`. The output (hypotenuse) is referenced by variable `h`. This is defined as follows:

$$h = \sqrt{p^2 + b^2} \tag{10.1}$$

The following Julia code performs this task and the function is called as `f(p,b)` where the values of `p` and `b` are given as inputs:

```
julia> function f(p,b)
                  h = sqrt(p^2 + b^2)
                  return h
           end
f (generic function with 1 method)

julia> f(3,4)
5.0

julia> f(3.0,4.0)
5.0
```

When `f(3,4)` is written at the Julia prompt, `f` is called with `p=3` and `b=4`. Consequently, `h` is calculated to be 5.0 and is returned. Thus, the output of `f(3,4)` is 5.0.

Note that when we define a function and end its definition by the keyword `end` and press Enter, REPL outputs the information that a generic function with one method has been created. This has to do with the kind of data types allowed by the operators used. The operators + and ^ operate on numerical data types only. Julia allows multiple dispatches, that is, a different definition of a function for different data types. This will be discussed later in Section 10.3.

It's always useful to name a function meaningfully, so instead of naming the previous function as `f`, we can name it `hypotenues`. When called, we write `hypotenues(p,b)`:

```
julia> function hypotenues(p,b)
                 h = sqrt(p^2 + b^2)
                 return h
         end
hypotenues (generic function with 1 method)

julia> hypotenues(3,4)
5.0
```

When we inspect the state of the computer's memory at this stage (after executing the Julia function at least once) by issuing the command `whos()`, we can observe that two new objects are shown, `f` and `hypotenues`, in the main namespace. For this reason, we can press the Tab key after filling in a few characters of its name and the tab competition will work for the function's name:

```
julia> whos()
Base  33853 KB    Module
Core  12333 KB    Module
Main  40714 KB    Module
...
f      0 bytes  #f
hypotenues      0 bytes  #hypotenues
```

It is worth noting that the scope of the variable `h` is local to the function `hypotenues`. In other words, it is valid only within the function body. These type of variables are called *local variables*. The local nature of the variable can be verified as follows:

```
julia> hypotenues(3,4) # function is called
5.0

julia> h = 5.5 # h is set to be 5.5
5.5
```

```
julia> hypotenues(3,4) # calculation of h is unaffected
#by previous setting of h value
5.0

julia> h # h still holds the set vaue after
# calling the function
5.5
```

10.2.1 Shorthand Notation

A function can also be written in shorthand notation:

```
fname(x,y,...) = a_function(x,y,...)
```

For example, in our case of calculating the hypotenuse, we can write a function in shorthand notation as follows:

```
julia> hypotenues(p,b)=sqrt(p^2 + b^2)
hypotenues (generic function with 1 method)

julia> hypotenues(3,4)
5.0
```

This syntax matches with the analytical way of writing mathematical functions. Thus, it is more intuitive. Moreover, not only ASCII but also Unicode characters can be used for naming the function. This makes it particularly easy while translating a mathematical expression into Julia code for scientific computation. But this notation has one severe limitation. You can only define a *single expression* within this format. In other words, only one mathematical rule can be defined in the body of the function.

10.2.2 Multiple Input

As is the case of the previous example of the function named `hypoteneus`, we can define a Julia function with multiple inputs. Let's consider another case. For example, let's consider the case when we need to find the length of a vector from the origin using its three components on *x, y,* and *z* axes:

$$l = \sqrt{x^2 + y^2 + z^2} \tag{10.2}$$

The following Julia code defines a function named `length_vec` that takes three inputs (*x, y,* and *z* values) and gives output as `length`, which calculates for Equation 10.2.

```
julia> function length_vec(x,y,z)
length = sqrt(x^2 + y^2 +z^2)
return length
end
length_vec (generic function with 1 method)

julia> length_vec(2,3,4)
5.385164807134504

julia> length_vec(-2,-3,-4)
5.385164807134504
```

From the previous code, it is worth noting that the vector

$$2\hat{i}+3\hat{j}+4\hat{k}$$

and

$$-2\hat{i}-3\hat{j}-4\hat{k}$$

have the same length from (0, 0, 0), which is ≈ 5.4 units.

However, we sometimes need a variable list of arguments in the multiple input scenario. In this case, we would need to have a flexibility in the number of inputs while calling the function. The next section discusses a variable argument list as input to the function.

Variable Argument List

While defining a function, optional arguments can be defined so that the function can use sensible defaults if specific values aren't supplied. As an example, we shall modify the already defined function `length_vec` (Section 10.2.2) and modify the same:

```
julia> function length_vec1(x,y,z=0)
length = sqrt(x^2 + y^2 +z^2)
return length
end
length_vec1 (generic function with 2 methods)
```

```
julia> length_vec1(-2,-3) # z=0
3.605551275463989

julia> length_vec1(-2,-3,-4) #z=-4
5.385164807134504
```

A new function named `length_vec1` is defined with inputs *x, y, z* = 0. Values for *x, y* are taken from the user and, if the value of *z* is not supplied by the user, it is taken to be equal to 0 (default value).

A function defined this way will ensure that we do not encounter an error if the right number of arguments is not supplied by the user, and it also defines a default behavior of a Julia function. Both of these features prove useful during numerical computations as well as software developments to avoid annoying error messages.

Positional Arguments

Until now, when we defined multiple inputs, they were used in the order in which they were defined. What if the order of input is a critical factor in computation? For example:

```
julia> function sumprod(a,b,c)
answer = (a+b)*c
return answer
end
sumprod (generic function with 1 method)

#(1+2)*3=0

julia> sumprod(1,2,3)
9

#(3+2)*1=5
julia> sumprod(3,2,1)
5
```

If the order of input is (1, 2, 3), we obtain 9. When the order of input is (3, 2, 1), the result is 5. This is because the values of the variables are `a=1`, `b=2`, and `c=3` in the first case, while the values of the variables are `a=3`, `b=2`, and `c=1` in the second case. In such cases where users can make a mistake that can result in erroneous calculation, we need to find a way to avoid this issue.

In addition, when we have a large number of inputs, it is difficult to keep track of their order. This can be avoided if we use a positional argument method. Keywords can be labeled for arguments in the form of a keyword=value pair:

```
julia>julia> function data_type(a,b;c="Int64",d="Complex64")
                println(typeof(a))
                println(typeof(b))
                return "Type of c is $c and Type of d is $d"
      end
data_type (generic function with 1 method)

julia> data_type(2,3)
Int64
Int64
"Type of c is Int64 and Type of d is Complex64"

julia> data_type(2.5,3)
Float64
Int64

julia> data_type("Hi",3)
String
Int64
"Type of c is Int64 and Type of d is Complex64"

julia> data_type("Hi",3,c=typeof('a'),d=typeof(1//2))
String
Int64
"Type of c is Char and Type of d is Rational{Int64}"

julia> data_type(c=typeof('a'),"Hi",d=typeof(1//2),3)
String
Int64
"Type of c is Char and Type of d is Rational{Int64}"
```

In the present context, three categories of inputs can be defined as follows:

- Normal
  - Arguments that must be specified and also must be in a specific order
- Optional
  - Arguments that may be skipped, but if specified, they must be in order
- Keywords
  - Arguments that can be skipped and need not be specified in an order

Let's understand these three kinds of inputs with the following Julia code. We define a function named `function1` with three inputs (of three finds discussed before: normal, optional, and keywords). The function is called successively with these arguments to illustrate their usage:

```
julia> function function1(normal,optional=1;keyword=0.001)
                 println("normal argument is $normal")
                 println("optional argument is $optional")
                 println("keyword argument is $keyword")
       end
function1 (generic function with 2 methods)

#Only normal argument is specified
julia> function1("Hi")
normal argument is Hi
optional argument is 1
keyword argument is 0.001

#Normal and Optional arguments
# both are specified and the value of optional
# argument is changed from 1 to 2
julia> function1("Hi",2)
normal argument is Hi
optional argument is 2
keyword argument is 0.001
```

```
#Keyword argument is explicitly specified
#with a different value
julia> function1("Hi",2,keyword=0.1)
normal argument is Hi
optional argument is 2
keyword argument is 0.1

#Keyword argument do not follow orders
julia> function1(keyword=0.1,1,2)
normal argument is 1
optional argument is 2
keyword argument is 0.1
```

Variable Arguments List

Users might need the ultimate flexibility in providing a variable list of arguments in some cases. For this purpose, a list of arguments is supplied from which values can be picked as required. This can be achieved by using the `splat` operator (`...`). Using the *help* mode, one can find information about using the `splat` operator. The following Julia code will illustrate the use of the variable argument list:

```
julia> function variable_arguments(args...)
answer = length(args)
println("number of arguments is $answer")
end
variable_arguments (generic function with 1 method)

julia> variable_arguments(1)
number of arguments is 1

julia> variable_arguments(1,2)
number of arguments is 2

julia> variable_arguments(1,2,"Hi")
number of arguments is 3

julia> variable_arguments(1,2,'a')
number of arguments is 3
```

Another example shows the use of this facility for printing the second value among arguments:

```
julia> function second_value(args...)
return args[2]
end

julia> second_value([1,2],3,4) 3

julia> second_value([1,2],[2,1])
2-element Array{Int64,1}:
2
1
```

10.2.3 Multiple Outputs

A function returns objects, which are termed as its outputs. A Julia object outputs just one object. So how can we obtain multiple output values? We have to understand that there is a difference between obtaining multiple values in the output and obtaining multiple output objects.

Multiple outputs are returned as a tuple of values instead of a single value. In this manner, a function still returns a single object. It is important to remember that tuples can be created and destructured without parentheses, which gives an illusion of returning multiple values. Let's explore multiple output functions with a simple example: a Julia function named `power` takes two inputs `a,b`, performs calculations a^b and $b^{a,}$, and outputs them. When called with inputs 2 and 3, it outputs $2^3 = 8$ and $3^2 = 9$ as a tuple:

```
julia> function power(a,b)
a^b,b^a
end
power (generic function with 1 method)

julia> power(2,3)
(8,9)

julia> x,y=power(2,3)
(8,9)
```

```
julia> x
8

julia> y
9
```

Two variables, x and y, can be used with an assignment operator when functions are called so that they are assigned the corresponding element of the output tuple. An equivalent syntax for the same function definition including a return statement is as follows:

```
julia> function power(a,b)
return a^b,b^a
end
```

10.2.4 Anonymous Functions

When functions are defined without names, they are called *anonymous* functions. For example, `x->x^3-3x^2+4x-21` defines a mathematical function:

$$f(x) = x^3 - 3x^2 + 4x - 21$$

They are used to pass them to functions that take other functions as arguments. Anonymous functions can be defined as `map()`, which will map the anonymous function to the values supplied as the second argument. The second argument can be a single value or multiple values as a list. Also the data type of inputs must be workable with operators used in the definition. The Julia code for the same can be written as follows:

```
julia> map(x->x^3-3x^2+4x-21,3)
-9

julia> map(x->x^3-3x^2+4x-21,[3,2,1])
3-element Array{Int64,1}:
-9
-17
-19

julia> x
ERROR: UndefVarError: x not defined
```

It is important to note that after the execution of map() function, *x* disappears from the namespace as it was a local variable for the function map(). Hence, when *x* is probed, it shows an UnderVarError since it is not defined in the present namespace.

Mapping Multiple Values

Multiple values can be mapped when inputs are provided as a tuple, that is, inputs are separated by a comma. Here the order of input values will matter:

```
julia> map((x,y,z) -> sqrt(x^2+y^2+z^2),[1,1,1],[-1,-1,-1],
[0,0,1])
3-element   Array{Float64,1}:
1.41421
1.41421
1.73205
```

Its is important to note that for the calculation

$$\sqrt{x^2+y^2+z^2}$$

the first element of each array is taken to perform the calculation and then output the first element. Then the same happens with the second and third to produce the result:

$$\sqrt{(1)^2+(-1)^2+(0)^2}=\sqrt{2}=1.41421$$
$$\sqrt{(1)^2+(-1)^2+(0)^2}=\sqrt{2}=1.41421$$
$$\sqrt{(1)^2+(-1)^2+(1)^2}=\sqrt{3}=1.73205$$

10.2.5 map() Function

The built-in function map() can be used for nonanonymous functions, too. If you have a function and an array, the function can be called for each element of the array by using the map() function. The function is called for each element of the array. The results are collected as an array that is then returned. The whole process is termed *mapping*:

```
julia> map(sin,0)
0.0
```

```
julia> map(sin,[0,pi])
2-element Array{Float64,1}:
0.0
1.22465e-16

julia> map(sind,[0,pi])
2-element Array{Float64,1}:
0.0
0.0548037
```

The elementwise operation of the `map()` function is built in in most functions. They are, in fact, optimized for faster operations:

```
julia> @time map(sin,1:100000);
0.005181 seconds (11 allocations: 781.625 KB)
```

```
julia> @time sin(1:100000);
0.021231 seconds (4.06 k allocations: 959.704 KB)
```

As is clear from the previous example, `sin()` gives faster results than mapping.Mapping from one array to another can be performed (it is done elementwise) for a given function, provided both of them have the same size. For example:

```
julia> map(//,1:10,2:11)
10-element Array{Rational{Int64},1}:
1//2
2//3
3//4
4//5
5//6
6//7
7//8
8//9
9//10
10//11
```

```
julia> map(^,[1,2,3],[2,4,3])
3-element Array{Int64,1}:
1
16
27
```

In the first example code, the command `map(//,1:10,2:11)` maps the built-in function `//` from the array `1:10` to the array `2:11` in an elementwise fashion; the first element is `1//2`, the second element is `2//3`, and so on.

In the second example code, the command `map(^,[1,2,3],[2,4,3])` maps the built-in function ^ from array `[1,2,3]` to the array `2,4,3` elementwise. The elements of the resultant array are the following:

$$(1)^2 = 1$$

$$(2)^4 = 16$$

$$(3)^3 = 27$$

10.2.6 reduce(), foldl(), and foldr() Functions

The `map()` function *collects* the results by operating a function elementwise on an iterable object, such as an array of numbers. On the other hand, the built-in function `reduce()` does a similar task, but after every element has been checked and processed by the function, only one is left. The function should take two arguments and return one. The array is reduced by continual application so that just one is left. For example, it can be used to sum up the contents of an array:

```
julia> reduce(+,[1,2,3])
6
```

In first case of command `reduce(+,[1,2,3])`, operator + performs the following:

$$1+2 \Rightarrow 3+3 \Rightarrow 6$$

What if we use the subtraction operator? There is an issue surrounding the property of associativity:

```
(1-2)-3 \Rightarrow -1-3 \Rightarrow -4 \\
1-(2-3) \Rightarrow 1-(-1) \Rightarrow 2
```

```
julia> reduce(-,[1,2,3])
-4
```

The `reduce()` function starts clubbing elements for operating from the left. The `foldl()` and `foldr()` functions will determine the direction of folding the given array for a particular operation to obtain a single valued output:

```
julia> reduce(-,[1,2,3])
-4

julia> foldl(-,[1,2,3])
-4

julia> foldr(-,[1,2,3])
2
```

10.2.7 mapreduce()

Mapping and folding can be performed simultaneously using the `mapreduce()`, `mapfoldl()`, and `mapfoldr()` functions:

```
julia> mapreduce(+,-,[1,2,3])
-4

julia> map(+,[1,2,3])
3-element Array{Int64,1}:
1
2
3
```

```
julia> reduce(-,map(+,[1,2,3]))
-4

julia> mapfoldl(+,-,[1,2,3])
-4

julia> foldl(-,map(+,[1,2,3]))
-4

julia> mapfoldr(+,-,[1,2,3])
2

julia> foldr(-,map(+,[1,2,3]))
2
```

10.3 Multiple Dispatches

Until now, our function definitions included defining just one method. For example, when we construct a function:

```
julia> f(x,y)=x+y
f (generic function with 1 method)
```

The output of the REPL prompt at which the function is defined in a shortcut notation says that we have constructed a generic function with one method. The output is type-sensitive in the sense that the + works on numbers, but not on strings and characters, and hence will give a `MethodError` since the method is not defined for `Char` and/or `String` data type as is evident for the following code:

```
julia> f(x,y)=x+y
f (generic function with 1 method)

julia> f(2,3)
5

julia> f('a','b')
ERROR: MethodError: no
method matching +(::Char, ::Char)
Closest candidates are:
```

```
+(::Any, ::Any, ::Any, ::Any...) at
operators.jl:424
+(::Char, ::Integer) at char.jl:40
+(::Integer, ::Char) at char.jl:41
Stacktrace:
[1] f(::Char, ::Char) at ./REPL[365]:1
```

To avoid such scenarios, multiple dispatch facilities can be given while defining functions so that functions can perform computations workable for various data types.

10.3.1 Defining Multiple Function Definitions

Let's look at the concept of multiple dispatches by taking a simple example of a function named `typeInfo` defined below:

```
#Defining first method for Int64
julia> function typeInfo(a::Int64)
println("Input's type is Int64")
end
typeInfo (generic function with 1 method)

#Defining second method for Float64
julia> function typeInfo(a::Float64)
println("Input's  type  is  Float64")
end
typeInfo (generic function with 2 methods)

#Defining third method for Char
julia> function typeInfo(a::Char)
println("Input's type is Char")
end
typeInfo (generic function with 3 methods)

#Defining fourth method for String
julia> function typeInfo(a::String)
println("Input's  type  is  String")
end
typeInfo (generic function with 4 methods)
```

```
julia> typeInfo(2)
Input's type is Int64

julia> typeInfo(2.0)
Input's type is Float64

julia> typeInfo('a')
Input's type is Char

julia> typeInfo("Hello")
Input's type is String
```

The `::` operator is used to attach a particular type to the variables.

The defined methods are probed using the `methods()` functions. As an example, let's probe the methods for the function `typeInfo()` we defined earlier:

```
julia> methods(typeInfo)
# 5 methods for generic function "typeInfo":
typeInfo(a::String) in Main at REPL[372]:2
typeInfo(a::Char) in Main at REPL[371]:2
typeInfo(a::Float64) in Main at REPL[370]:2
typeInfo(a::Complex{Float32}) in Main at REPL[369]:2
typeInfo(a::Int64) in Main at REPL[368]:2
```

It is important to note that the previous definitions of functions and their methods will have a life inside the present REPL session.

10.4 Operators Defined as Functions

In Julia, most operators are just functions with support for special syntax where symbols are used instead of names. But remember that symbols are simply Unicode characters that are valid function names. For example, the operator + can be called just like a function on two numbers. The infix form is exactly equivalent to the function application form. Hence, the operator can be assigned to another name and used just like any other function:

```
julia> 1+1.0
2.0
```

```
julia> +(1,1.0) # called like a function
2.0

julia> f=+ # name can be assigned to a variable
+ (generic function with 163 methods)

julia> f(1,1.0) # Assigned vairable name can be
# used for calling
2.0
```

The exception to this case is `&&` and `||` operators since they require that their operands are not evaluated before the evaluation of the operator.

10.4.1 Functions Returning Functions

A function (let's call this a *minor* function) can be nested inside another function (let's call this a *major* function). In this case, the minor function returns an object that is used by a major function to return another object in return.

As an example, let's consider a major function named `expo()` that takes one input *x*, which is essentially the power of exponentiation. A minor function defines the functionality of calculating y^x where *y* defines the values for which the power *x* is calculated. Apart from calculating y^x, the minor function also prints the *type* of object returned by the minor function:

```
julia> function expo(x)
       expo1  = function(y)
                answer=y^x
                answer_type = typeof(answer)
                println(answer_type)
                return answer
                end
       end

julia> sq(2)
4

julia> sq = expo(2)
(::#58) (generic function with 1 method)
```

```
julia> sq(2)
Int64
4

julia> sq(2.0)
Float64
4.0

julia> cu = expo(3)
(::#58) (generic function with 1 method)

julia> cu(2)
Int64
8

julia> cu(2.0)
Float64
8.0

julia> sq_root = expo(0.5)
(::#58) (generic function with 1 method)

julia> sq_root(2)
Float64
1.4142135623730951

julia> sq_root(3)
Float64
1.7320508075688772

julia> cu_root(2)
Float64
1.2599210498948732

julia> cu_root(3)
Float64
1.4422495703074083
```

To calculate squares of numbers, you can write `sq=expo(2)`, which calculates the square of any number that is provided as input. Similarly, `cu = expo(3)` finds cubes, `sq_root = expo(0.5)` finds square roots, and `cu_root = expo(1//3)` finds the cube roots.

10.5 Summary

In this chapter, we have discussed functions within a Julian framework. Defining a bigger code into a group of functions makes the task modular and manageable in a flexible manner. The ability to pass arguments strictly or with flexibility allows us to write specific functions for specific requirements. Functions also let us define methods for particular objects. Thus, this ability forms the core of the OOP concept and Julia uses functions quite effectively.

CHAPTER 11

Control Flow

11.1 Introduction to Control Flow

When you evaluate Julia code line-by-line for execution, you sometimes need to shift the flow of execution out of this line-by-line manner to a different point of the code. In other words, the flow of the program needs to be altered. There are many reasons why you would need to make this shift. For example, a condition needs to be checked and then the flow can be directed to one of many directions, some parts of the calculation are repetitive so the program needs to be altered before returning back, or an error might happen and the flow needs to redirected. Julia provides powerful constructs for these situations, which will be described in this chapter.

11.2 Ternary Expression

When flow needs to be chosen for two directions based on a simple *yes* or *no* answer for a condition, ternary operators ? and : can be used efficiently within a one-line statement. For example, suppose one constructs a vector (referenced by `a`) as `[1,2,3,4,5]`. If the length of this vector is less than 3, then all the elements must be squared. If the length of this vector is more than 3, then the square root of all elements must be taken. The following code can be implemented for this purpose:

```
julia> a = [1,2,3,4,5]
5-element Array{Int64,1}:
1
2
3
4
5
```

```
julia> length(a)<3 ? a.^2 : sqrt.(a)
5-element Array{Float64,1}:

1.0
1.41421
1.73205
2.0
2.23607

julia> a=[1,2]
2-element Array{Int64,1}:
1
2

julia> length(a)<3?a.^2:sqrt.(a)
2-element Array{Int64,1}:
1
4
```

Since the length of a in the first case is actually 5 (it has five elements), the condition is not satisfied and, hence, the square root of the elements is taken. In the second case, the length of a becomes 2 and the condition is satisfied, which is why elements are squared. This simple construct provides a powerful means for directing the flow within just one line of code.

11.3 Boolean Switching

Similar to ternary expressions, boolean switching is presented in the cases where a single or compound logical sentence presents `true` or `false` boolean data type as output. The operators && represent the `and` operator, whereas || represents the `or` operator. Their usage can be understood in the following Julia code:

```
julia> a = 1
1

julia> isodd(a) && true
true
```

```
#verification
julia> isodd(a)
true

julia> true && true
true

julia> isodd(a) || true
true

#verification
julia> true || true
true
```

These operators are used to evaluate a particular logical condition, following which a decision can be made to direct the flow in a desired direction.

11.4 if-else

To check a logical condition, the keyword `if` is used following the set of expressions that is executed when the condition is found to be `true`. If the condition is found to be `false`, the set of expressions following the `else` keyword is executed. The keyword `elseif` provides an option for checking another condition. The number of `elseif` blocks is not limited to a particular number, which makes this option a powerful one for complex systems. Conditions are made using boolean expressions as described in Section 11.3.

A simple example can be used to check if an element type of an input is an integer or a float. The following Julia code accomplishes this:

```
julia> function el_type(a)
        if eltype(a)==Int64
                println("$a is an integer")
        else
                println("$a is a float")
        end
end
el_type (generic function with 1 method)
```

```
julia> el_type(2)
2 is an integer

julia> el_type(2.0)
2.0 is a float
```

The code can be improved to contain other data types using the `elseif` keyword. For example, complex data types can also be included:

```
julia> function el_type(a)
        if eltype(a)==Int64
                println("$a is an integer")
        elseif eltype(a)==Complex64
                println("$a is a complex number")
        else
                println("$a is a float")
        end
end

julia> el_type(complex(2,3))
2 + 3im is a float

julia> el_type(2)
2 is an integer

julia> el_type(2.0)
2.0 is a float
```

In a similar fashion, a number of `elseif` conditions can be inserted. An important aspect of Julia syntax is that you don't need to worry about defining blocks of statements within brackets (as in C and C++) or with indents (as in Python). The indents used here are for visual clarity. The Julia compiler does not demand the same syntactically, so you also don't have to worry about using whitespace, braces, indentation, brackets, semicolons, and so on, for defining blocks of code. However, you need to remember to finish the conditional construction with `end`. Furthermore, the `elseif` and even the `else` parts are optional.

11.5 for Loop

The `for ... end` construct helps in working through a list or a set of values, or from a start value to a finish value. The following example will help in understanding its usage. Suppose we simply want to print the elements of an array `[1,2,3]`:

```
julia> a = [1,2,3]
3-element Array{Int64,1}:
1
2
3

julia> for i in a
               println(a[i])
       end
1
2
3
```

The elements of an array are iteratively valued by `i` over the array `a` using the membership operator `in`, that is, using the syntax `for i in a`. Since Julia uses a very intuitive syntax structure, it is easier for users to conceive and understand code.

It is important to note that while `a` was an array object, the `for` loop does not return an array. Instead, a `Void` type object is returned. The `Void` object has just one instance called `nothing`, which is used by convention when there is no value to return. At the end of the loop, no value is returned as the loop simply performs whatever the tasks it is defined to do. Some functions and/or parts of code are used only for their side effects and do not need to return a value. The `for` loop structure is one of them. After the loop ends, it must not create an object; hence, it creates a `Void` object. As a result, REPL does not print anything for it:

```
julia> a=for i in a
println(a[i])
end
1
2
3

julia> typeof(a)
Void
```

Complex mathematical structures can be included within the block of the loop for performing desired operations. For example, let's consider a Julia code that prints `Odd` when it encounters a odd number and prints `Even` otherwise. The numbers can be checked using the function `isodd()`:

```
julia> a=[1,2,3,4,5]
5-element Array{Int64,1}:
1
2
3
4
5

julia> for i in a
       if isodd(i)
              print("Odd \n")
       else
              print("Even \n")
       end
end

Odd
Even
Odd
Even
Odd
```

11.5.1 Scope of a Loop Variable

The existence of a looping variable (`i` in the previous code) is independent of the loop in which it is used. It can exit beforehand, in which case the loop's behavior affects it. If it did not exist before, it is destroyed as soon as the loop is exited. The following example will make this clear:

```
julia> i=1
1
```

```
julia> for i in 1:5 #i already existed
println(i)
end
1
2
3
4
5

julia> i # final value is governed by loop
5

julia> for j in 1:5 #j did not existed before
println(j)
end
1
2
3
4
5

julia> j # j does not exist after loop
ERROR: UndefVarError: j not defined
```

11.5.2 continue

When you wish to skip certain values during a loop, `continue` comes in handy. You make a rule using a logical expression and skip the values using an `if` statement. For example, suppose you want to extract only the odd numbers from arrays of numbers from 1 to 10:

```
julia> for i in 1:10
                if i%2 == 0
                        continue
                end
        println(i)
        end
```

```
1
3
5
7
9
```

When condition `i%2==0` is checked, it proves true for all even numbers. Hence, they are skipped using a `continue` statement. Others are simply fed to the `println()` function, which prints them.

11.5.3 Comprehensions

Comprehensions are convenient ways of defining arrays using `for` loops. Here a rule is defined within [ ] brackets (which removes the necessity of writing the `end` keyword to end the `for` loop). For example, if you need to define an array of the square root of numbers from 1 to 5, then you can write Julia code in one line as follows:

```
julia> [sqrt(i) for i in 1:5]
5-element Array{Float64,1}:
1.0
1.41421
1.73205
2.0
2.23607
```

You can define the type of elements by defining the type array outside [ ] brackets. For example, writing `Complex64` outside square brackets ensures the elements are of the type `Complex64`. (Complex numbers are stored in 64 bits of memory.) The complex numbers are defined as follows:

$$\sqrt{k}+(2\times k)i$$

```
julia> a = Complex64[sqrt(k)+(2k)im for k in 1:5]
5-element Array{Complex{Float32},1}:
1.0+2.0im
1.41421+4.0im
```

```
1.73205+6.0im
2.0+8.0im
2.23607+10.0im

julia> eltype(a)
Complex{Float32}
```

An array of tuples, valued by a comprehension rule, can also be created as follows:

```
julia> [(sqrt(k),k,k^2) for k in 1:5]
5-element Array{Tuple{Float64,Int64,Int64},1}:
(1.0,1,1)
(1.41421,2,4)
(1.73205,3,9)
(2.0,4,16)
(2.23607,5,25)
```

In this example, each element is a tuple consisting of the square root of a number, the number itself, and its square. Numbers range from 1 to 5.

Two iterators can also be defined in a comprehension format as follows:

```
julia> [(a,b) for a in 1:5,b in 2:4]
5x3 Array{Tuple{Int64,Int64},2}:
(1,2)  (1,3)  (1,4)
(2,2)  (2,3)  (2,4)
(3,2)  (3,3)  (3,4)
(4,2)  (4,3)  (4,4)
(5,2)  (5,3)  (5,4)
```

The first elements of tuples range from 1 to 5; the second element ranges from 2 to 4.

11.5.4 Generators

Generators, which are new to Julia, were introduced in version number 0.5. Just like comprehensions, generators can be used to produce values from iterating a variable. But they pose a striking difference. Unlike comprehensions, the values are produced on demand. Let's look at example Julia code:

```
julia> collect(x for x in 1:100 if x%7==0 && x%3==0)
4-element Array{Int64,1}:
21
42
63
84
```

This example makes an array of numbers (within the range of 1 to 100 and that are multiples of both 7 and 3).

11.5.5 enumerate

Using the built-in function `enumerate()`, you can produce a tuple of values with their index:

```
julia> a = ["3",3,3.0]
3-element Array{Any,1}:
"3"
3
3.0

julia> for(index,value) in enumerate(a)
println("$index $value")
end
1 3
2 3
3 3.0
```

In thsi example, an array is defined with three values: `"3"` (a string valued 3), `3` (an integer valued 3), and `3.0` (a floating point number valued 3.0). This list is iterated with `enumerate()` to produce index and value pairs and store them in a defined tuple `(index,value)`. Each value is printed using the `println()` function.

11.5.6 Zipping Arrays

Zipping arrays involves taking corresponding elements from each array as a member of a tuple. For example:

```
julia> for i in zip(1:5,5:10,10:15)
println(i)
end
(1,5,10)
(2,6,11)
(3,7,12)
(4,8,13)
(5,9,14)
```

The first element of each tuple is taken from the rule `1:5`, the second element is taken from the rule `5:10`, and the third element of each tuple is taken from the rule `10:15`.

11.6 while Loop

The `while ... end` construct is used when a particular expression or a set of expressions needs to be calculated while a condition is `true`.

```
julia> x = 0
0

julia> while x<5
       println(sqrt(x))
       x+=1
end

0.0
1.0
1.4142135623730951
1.7320508075688772
2.0
```

This code first initializes the variable named x to value 0. Now the condition $x < 5$ is checked. When $x = 0$, then this condition is `true` and `println(sqrt(x))` is executed. The `x++1` increments the value of x to 1 and again the condition is checked. If found `true`, the `println()` statement is executed. This is done until the condition is `true`, that is, until $x = 4$.

11.7 Nested Loops

One of the most convenient aspects of defining loops in the Julia programming language is the simplicity of defining nested loops. Nested loops are written by simply separating loops with a comma `(,)` operator. For example, let's consider the following Julia code where first x is spanned from 1 to 3 and, inside each step of this loop, y is spanned from 1 to 3. For each step within the y loop, z is defined as a tuple where the current value of x and y are fed. So when `x=1`, coordinates `(1,1), (1,2), (1,3)` are created iteratively and then x is incremented to 2 and same is done again:

```
julia> for x in 1:3,y in 1:3
        z = (x,y)
        println(z)
        end
(1,1)
(1,2)
(1,3)
(2,1)
(2,2)
(2,3)
(3,1)
(3,2)
(3,3)
```

Similarly, another Julia code can be written where three coordinates on *x, y,* and *z* axes of a vectors are defined using nested loops in `x,y,` and `z`, respectively (in the same order). The coordinates are created as follows:

- First, x = 1 and y = 1, and z takes values 1a dn, then 2.
  - Coordinates are created as (1, 1, 1) and (1, 1, 2).

- Next, x = 1 and y increments to 2, and z takes values 1a dn, then 2.
  - Coordinates are created as (1, 2, 1) and (1, 2, 2).
- Next x = 1 and y increments to 3, and z takes values 1a dn, then 2.
  - Coordinates are created as (1, 3, 1) and (1, 3, 2).
- Now y and z loops have been exhausted to outer loop for x incremented its value to x = 2 and the process is repeated for this new value of x.
  - First, (2, 1, 1) and (2, 1, 2) are created.
  - Next, (2, 2, 1) and (2, 2, 2) are created.
  - Next, (2, 3, 1) and (2, 3, 2) are created.
- Again y and z loops have been exhausted to outer loop for x incremented its value to x = 3 and the process is repeated for this new value of x.
  - First, (3, 1, 1) and (3, 1, 2) are created.
  - Next, (3, 2, 1) and (3, 2, 2) are created.
  - Next, (3, 3, 1) and (3, 3, 2) are created.
- In each case, the length is calculated by the following formula:

$$\sqrt{x^2 + y^2 + z^2}$$

This is stored in the variable name distance and is used in the println() function for printing.

```
julia> for x in 1:3,y in 1:3, z in 1:2
distance = sqrt(x^2+y^2+z^2)
println("Length for vector ($x,$y,$z) is $distance")
end
```

```
Length for vector (1,1,1) is 1.7320508075688772
Length for vector (1,1,2) is 2.449489742783178
Length for vector (1,2,1) is 2.449489742783178
Length for vector (1,2,2) is 3.0
Length for vector (1,3,1) is 3.3166247903554
Length for vector (1,3,2) is 3.7416573867739413
Length for vector (2,1,1) is 2.449489742783178
Length for vector (2,1,2) is 3.0
Length for vector (2,2,1) is 3.0
Length for vector (2,2,2) is 3.4641016151377544
Length for vector (2,3,1) is 3.7416573867739413
Length for vector (2,3,2) is 4.123105625617661
Length for vector (3,1,1) is 3.3166247903554
Length for vector (3,1,2) is 3.7416573867739413
Length for vector (3,2,1) is 3.7416573867739413
Length for vector (3,2,2) is 4.123105625617661
Length for vector (3,3,1) is 4.358898943540674
Length for vector (3,3,2) is 4.69041575982343
```

11.8 do ... end

The `do ... end` construct can be used just like comprehension. Suppose we have an array A having five numbers. When we write an anonymous function ($x- > x == 1||x == 4|$ where the value of x is either 1 or 4), to `find()` function (which returns the index of resultant element), we can also use the `do ... end` construct as an alternate. Here, we just avoid defining an anonymous function, as illustrated in the following Julia code:

```
julia> A = [1,2,3,4,5]
5-element Array{Int64,1}:
1
2
3
4
5

julia> find(x-> x==1||x==4,A)
2-element Array{Int64,1}:
1
4
```

```
julia> find(A) do x
x==1||x==4
end
2-element Array{Int64,1}:
1
4
```

11.9 Exceptions

Sophisticated level programming involves error (or exception) handling. It is an essential feature of writing Julia code and coders are encouraged to write their code with this feature for better usage and understanding by the general community as well as for code stability. When an unexpected condition occurs while executing a Julia program, a defined function may not be able to return a reasonable value to its caller. This will usually issue an error message, but it is advisable to use the exceptional condition to perform one of the following:

- Terminate the program.
- Print a diagnostic error message.
- If the programmer has provided code to handle such exceptional circumstances, allow that code to take the appropriate action.

This way, you control the way the program proceeds rather than just getting an error message.

11.9.1 Built-in Exceptions

There are a set of built-in exceptions in Julia, which are produced when an unexpected condition has occurred. We have already encountered some error messages in a similar fashion. The messages that we saw printed on REPL were defined because the Julia code that generated them was written for handling exceptions.

Going inside the `help` mode (by writing `?` on a Julia terminal and writing `Exception`), you can obtain a list of built-in exceptions. Some of them are listed in Table 11-1.

Table 11-1. Built-in Exceptions in Julia

Syntax	Illustration
`ArgumentError`	The parameters to a function call do not match a valid signature.
`BoundsError`	An indexing operation into an array, `a`, tried to access an out-of-bounds element, `i`.
`DivideError`	Integer division was attempted with a denominator value of 0.
`DomainError`	The arguments to a function or constructor are outside the valid domain.
`EOFError`	No more data was available to read from a file or stream.
`InexactError`	Type conversion cannot be done exactly.
`InterruptException`	The process was stopped by a terminal interrupt (CTRL+C).
`MethodError`	A method with the required type signature does not exist in the given generic function.
`OutOfMemoryError`	An operation allocated too much memory for either the system or the garbage collector to handle properly.
`ReadOnlyMemoryError`	An operation tried to write to memory that is read-only.
`OverflowError`	The result of an expression is too large for the specified type and will cause a wraparound.
`TypeError`	A type assertion failure occurred, or an intrinsic function was called with an incorrect argument type.
`UndefRefError`	The item or field is not defined for the given object.
`UndefVarError`	A symbol in the current scope is not defined.
`DimensionMismatch`	The objects called do not have matching dimensionality.
`AssertionError`	The asserted condition did not evaluate to `true`.

11.9.2 Custom-Built Exceptions

Built-in exceptions can be used by the built-in function `throw()`, which throws an exception as per a defined rule. For example, a `DomainError` can be thrown when the user inputs a non-negative number defined exclusively for negative numbers, as shown in the following Julia code:

```
julia> f(x) = x<0 ? exp(x) : throw(DomainError())
f (generic function with 1 method)

julia> f(-1)
0.36787944117144233

julia> f(1)
ERROR: DomainError:
Stacktrace:
[1] f(::Int64) at ./REPL[31]:1
```

The condition `x<0` ensures that `x` should be smaller than 0.

Another function named `error()` is used to produce an `ErrorException` that interrupts the normal flow of control. For example, the following Julia code is written where we wish to entertain only integers for inputting to our defined function. Any other type of data should display a descriptive error message so that the user can input correctly:

```
julia>f(x)=typeof(x)==Int64?e^x:error("Input only integer")
f (generic function with 1 method)

julia> f(1)
2.718281828459045

julia> f(2)
7.38905609893065

julia> f(2.0)
ERROR: Input only integer
Stacktrace:
[1] f(::Float64) at ./REPL[34]:1
```

11.9.3 catch...try Construct for Testing Exceptions

The `try .. catch` statement allows for exceptions to be tested for. The catch clause is not strictly necessary; when omitted, the default return value is `nothing` (the singleton instance of type `Void`, used by convention when there is no value to return).

The `try ... catch` construct let's us handle exceptions, both generally and dependent on a variable. The general structure is as follows:

- `try`
  - The main body of the function should be written within this block.
  - Julia will *try* to execute the code within this block.
- `catch`
  - This block *catches* the errors.
  - It is advised to use a variable, to which the exception can be assigned.
  - This variable can be used in the `if ... elseif ... else` construct to check the exception.

Let's consider the example of a custom-built function to calculate the square root of a quantity `x` where `x` has to be a positive number (`Integer`, `Float`, `Complex` type). We would encounter an error if the user feeds a string as input. Hence, the error is stored in `err` variable and then uses `if ... else` construct. If the `err` value is `MethodError` (that is, the data type isn't defined within the method of the built-in `sqrt()` function), then an error message is printed on the terminal. This is verified in the following Julia code:

```
julia> function square_root(x)
        try
                sqrt(x)
        catch err
                if isa(err,MethodError)
        println("Input a number")
                end
        end
end

square_root (generic function with 1 method)

julia> square_root("a")
Input a number

julia> square_root("alpha")
Input a number
```

```
julia> square_root(complex(2,3))
1.6741492280355401 + 0.8959774761298381im

julia> square_root(2)
1.4142135623730951
```

The code can be improved to include the `DomainError` using the `elseif` option where, if a negative number is given as input, it will be treated like a complex number and the square root will be calculated:

```
julia> function square_root(x)
        try
                sqrt(x)
        catch err
                if isa(err,MethodError)
                println("Input a number")
                elseif isa(err,DomainError)
                        sqrt(complex(x))
                end
        end
end

square_root (generic function with 1 method)

julia> square_root(-2)
0.0 + 1.4142135623730951im

julia> square_root("a")
Input a number

julia> square_root(2)
1.4142135623730951
```

11.9.4 finally

Once the `try ... catch` construct has finished, Julia executes the code whether the operation has succeeded or not. The `finally` keyword executes whether there was an exception or not. This is particularly important because the Julia code that affects the state changes or uses resources like files, involving typical clean-up work (such as closing files). These tasks needs to be done when the code is finished (finally!). Exceptions are not a good choice for these tasks since they can cause a block of code to exit before reaching its normal end. The `finally` keyword provides a way to run a particular Julia code when a given block of code exits (regardless of the fact that it exited). This is particularly important for file handling so it will be discussed in Chapter 12.

11.10 Summary

In this chapter, we illustrated Julia's control flow structures. The control flow of computational tasks is essential to any computational tasks. Defining loops, checking comparison, and defining expressions for complex logical statements are critical for writing code to solve a physical problem. The ability to run these tasks effectively with different data types in a timely manner makes Julia a prime candidate for numerical computation. The tasks' ease of usage is a prominent feature in Julia. One-line definitions of such features can be seen in very few programming languages. Furthermore, there aren't many strict rules about using indents and brackets to define the blocks of code, which avoids syntactical errors on the part of the programmer.

CHAPTER 12

Input Output

12.1 Introduction

Input and output of data are such basic operations for a programming language that they are mostly taken for granted and powerful constructs existing for the same are mostly ignored. Julia provides a variety of ways to input and output data. It is important to differentiate them and use the appropriate one for a particular application.

Julia provides a rich ecosystem of interfaces to deal with streaming I/O objects such as terminals, pipes, and TCP sockets. Julia prefers to handle data in terms of streams; data is streamed continuously to the Julia program rather than working on it as a block of memory. There are several varities of streaming (for both inputing and/or outputting purposes). This chapter will deal with a variety of ways in which Julia can handle data input and/or output streams.

12.2 Console I/O

The Julia `in` this mode is, in fact, a medium for inputting and outputting data. This comes under the category of *console I/O*. We have already used two functions, `print()` and `println()`, for this purpose. They print the their inputs on the Julia terminal where `println()` also appends a character to the output. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details:

```
julia> println("Hello World")
Hello World

julia> print("Hello World")
Hello World
```

The terminal input can be achieved using the built-in `readline()` function. It reads the keyboard input until the first occurrence of a newline character. The newline character `\n` is also stored when the `readline()` function is used for input:

```
julia> a = readline();
Hi

julia> a
"Hi\n"
```

12.3 Basic Stream I/O

The basic Julia functions `read()` and `write()` take the streams as their first argument. Let's first explore the built-in writing `write()` function:

```
julia> write(STDOUT,"Hello")
Hello5

julia> write(STDOUT,"Hello\n")
Hello
6

julia> write(STDOUT,"Hello\nWorld")
Hello
World11

julia> write(STDOUT,"Hello\tWorld")
Hello   World11

julia> write(STDOUT,"Hello World")
Hello World11
```

`STDOUT` is a global variable referring to the standard out stream:

```
julia> a = STDOUT
Base.TTY(RawFD(14) open, 0 bytes waiting)
```

TTY represents the computer terminal. By default, `STDOUT` is the set of output data stream to the computer terminal. When we wrote the following:

```
write(STDOUT,"Hello")
```

we got the output `Hello5`. Here `Hello` is the content of the input string to the `write()` function and `5` is the number of bytes in the stream. When we wrote the following:

```
write(STDOUT,"Hello\n")
```

we added a new character called a *newline* character to the stream (represented by `\n`); hence, the count of the number of bytes increased to `6`. Also, when it is printed at TTY (computer terminal), you can observe that the newline character is also printed i.e between `Hello` and `6` being printed, a newline character is also printed (a gap of one line exists). Other test commands also show the same capabilities.

Returning the number of bytes can be suppressed with the `;` character at the end of the command. This can be easily verified in the following Julia code:

```
julia> write(STDOUT,"Hello")
Hello5

julia> write(STDOUT,"Hello");
Hello #supresses \n too
julia>
```

The built-in function `read()` works in a similar fashion. As the name suggests, this function will input a stream. Just like `STDOUT`, there is a global variable referring to the standard input stream, `STDIN`, which will be used in the `read()` function. Let's first use it in the following Julia code and look at its usage and behavior from the output:

```
julia> read(STDIN,Char)

'\n': ASCII/Unicode U+000a
(category Cc: Other, control)
```

The `read()` function would treat the stream as character data type from standard input set by default on the keyboard and stored at the global variable `STDIN`. In this case, the Enter key is pressed, which represents a newline character, `\n`, that is in turn printed. Similarly, you can print alphabets like `a` as follows:

```
julia> read(STDIN,Char)
a
'a': ASCII/Unicode U+0061
(category Ll: Letter, lowercase)
julia>
```

Within this same command structure, even a number key will produce a character data type:

```
julia> read(STDIN,Char)
1
'1': ASCII/Unicode U+0031
(category Nd: Number, decimal digit)
julia>
```

When two numbers are fed as input and then the Enter key is pressed, the second number is fed to the Julia command prompt:

```
julia> read(STDIN,Char)
10
'1': ASCII/Unicode U+0031
(category Nd: Number, decimal digit)

julia> 0 # Fed to REPL from previous input
0

julia> typeof(ans)
Int64
```

Here, the keys for numbers 1 and 0 are pressed and then the Enter key is pressed. In this case, 1 (the number `1` as a character) becomes part of the stream and duly displayed, after which 0 is fed to the next command prompt, which *evaluates* it to `0` and displays the same. Since the last evaluated entity is stored in the variable `ans`, checking its data type for 0 confirms that the data type is `Int64` and not `Char` as for 1:

```
julia> read(STDIN,Char)
ab
'a': ASCII/Unicode U+0061
(category Ll: Letter, lowercase)

julia> b
ERROR: UndefVarError: b not defined
```

Doing a similar task with a set of characters results in an error for `julia>b` since `b` variable is not defined and cannot be evaluated.

Using the command `methods(read)`, you can scan the kinds of data types that can be fed to the function `read()`. Also, it is worth noting that `write()` takes the data to write as its second argument, while `read()` takes the type of the data to be read as its second argument.

12.4 Byte Array Streaming

Just like single characters (bytes) are streamed, a byte array can also be streamed as follows:

```
julia> x = rand(UInt8,3)
3-element Array{UInt8,1}:
0x72
0x5a
0x37

julia> read!(STDIN,x)
abc
3-element Array{UInt8,1}:
0x61
0x62
0x63

julia> x
3-element Array{UInt8,1}:
0x61
0x62
0x63
```

During the first command, three random numbers of the data type `Unit8` (8 bits = 1 byte) are stored in a variable named `x`. These numbers are fed to the `read!()` function. The addition of `!` forces the function to change stored values with new values. Now the byte array `x` is open to read three bytes. When characters `a`, `b`, and `c` are fed at the keyboard, these bytes are stored in the byte array. Their hexadecimal representation can be verified to be present as values of elements in x. Conversely, another way to stream in byte values in byte arrays is the following:

```
julia> read(STDIN,3)
abc
3-element Array{UInt8,1}:
0x61
0x62
0x63

julia> ans
3-element Array{UInt8,1}:
0x61
0x62
0x63
```

Here three values are streamed in the array default output variable ans:

```
julia> read(STDIN,3)
123
3-element Array{UInt8,1}:
0x31
0x32
0x33

julia>

julia> read(STDIN,3)
1s3

3-element Array{UInt8,1}:
0x31
0x73
0x33

julia> read(STDIN,3)
#$%
```

```
3-element Array{UInt8,1}:
0x23
0x24
0x25
```

Notice that the output of streamed bytes is shown in their hexadecimal representation. Hence, the inputs can be any kind of Unicode characters in the input stream.

12.5 Streaming a Line of Characters

When a line of characters must be streamed in, a simpler built-in function can be used (`readline()`), which can take inputs of characters to make a line. Pressing the Enter key on a keyboard prints a newline character, which declares the end of the line. The output of the `readline()` function is a string. This is verified by issuing `typeof(ans)` at the Julia command prompt:

```
julia> readline(STDIN)
Hi, How are you
"Hi, How are you\n"

julia> readline(STDIN)
123.345
"123.345\n"

julia> readline(STDIN)
#$%^yY&*(
"#\$%^yY&*(\n"

julia> typeof(ans) #probing last input at REPL
String
```

Please note that depending on a particular terminal's settings, the TTY may be line-buffered and might thus require an additional enter before the data is sent to Julia REPL.

12.6 Text I/O

The capability to write textual data depends on the output media. For example, some softwares and associated hardwares will only understand ASCII characters, while others can also understand Unicode characters. For example, a seven-segment display can display only English alphabets and Roman numerals using ASCII code. Hence, it is important for a developer to keep in mind the kind of target hardware-software combination for a particular application. Generally, one writes a Julia code that needs to display on a graphical monitor (such as your desktop's or laptop's screen).

The data to monitor is streamed via a software channel that is responsible for interpreting it as per the monitor's configuration. Since it is a graphical terminal, apart from textual data, it can also handle a lot of graphical formats. In a general sense, printing textual objects and printing graphical objects are similar tasks and can be generalized for a computer-programming environment. Julia code enables handling these objects and interpreting them as per their defined properties. How many types of objects can one Julia function handle simply depends on how many methods have been defined for the same. (See Chapter 10.)

We have already seen the primary usage of the built-in function `write()`. The function `write()` operates on binary streams and text representations are written as is. For example, the character `a` is stored in one byte and is given a Unicode represented by the hexadecimal number 0*x*61. To print `a`, you can use the `write()` method:

```
julia> write(STDOUT,'a')
a1

julia> write(STDOUT,'a');
a
julia> write(STDOUT,0x61)
a

julia> write(STDOUT,0x61);
a1
```

It is worth noting that the `;` operator suppresses the printing of the number of bytes (one here) when used at the end of the command. Some other functions exist to handle textual objects in a more structured manner:

- `show()`
- `print()`

- `println()`
- `display()`

12.6.1 show()

Most of the display functions, ultimately called `show()` for writing an object `x`, are given a mime type to a given I/O stream (usually a memory buffer), if possible. The default mime type is plain text. The function `show()` requires two input arguments—type of I/O stream and data. The `show()` function can handle a variety of textual data and represents them using its defined mime. For example, a complex number must be displayed accordingly to its mime, which dictates textual information in the following order:

1. A number depicting the real part
2. A whitespace
3. The symbol for signs `+` or `-` (as is the case for the defined number)
4. A whitespace
5. A number for the imaginary part
6. The alphabets `im` immediately after the number for the imaginary part

Let's print some data types using the `show()` function. In each case, the data type is defined in the variable `a`, which is given as the second argument to `show()` since the first argument is `STDOUT`, which is set to a graphic terminal by default. This is the Julia terminal in the present case:

```
julia> show(STDOUT,'a')
'a'
julia> show(STDOUT,'1')
'1'
julia> a = "sandeep"
"sandeep"

julia> a = "sandeep nagar"
"sandeep nagar"
```

```
julia> show(STDOUT,a)
"sandeep nagar"

julia> show(STDOUT,a)
"sandeep"
julia> a = complex(2,3)
2 + 3im

julia> show(STDOUT,a)
2 + 3im
julia> a = 2//3
2//3

julia> show(STDOUT,a)
2//3
```

Similarly, mimes of characters dictate using single quotes around the characters and mimes for strings dictate the use of double quotes around the group of characters defining the string. Special characters like a whitespace are not displayed but *interpreted* for their behavior and displayed accordingly. Similarly, a rational number is printed with the symbol `\\` between the numbers for the numerator and the denominator.

The `IOContext` option can be used to pass the contextual information about output from the `show()` function. It can be the first argument for the `show()` function specifying output format properties. For example, :`compact` specifies that small values should be printed in a compact form. In the case of numbers, they should be printed with fewer digits. Similarly, :`displaysize` can be used to set the number of rows and columns for displays of textual data, overriding the information dictated by the calling function. This can be useful when you are using LCD displays that have a fixed number of rows and columns for handling alphanumeric data. Here, the display size can be set to a given number of rows and columns of a particular LCD display unit. In a similar fashion, data display can be truncated by using the :`limit` option for `IOContext` where displaying textual information can be truncated as per defined values. In all cases, it is worth noting that data are stored in memory and these functions only affect the display behavior of the same; they do not alter the data in memory. Even if the data display is truncated, it is not truncated by these functions for storage purposes.

12.6.2 print()

The built-in function `print()` simply prints an input to an output stream (by default, it is set as a Julia terminal) with minimal formatting. It calls the `show()` function if it cannot handle the features of formatting. Let's take some examples. When we set a variable `a` as numeric value 1, it prints the same. But when we set `a` as character value `'1'`, the `print()` still displays its numeric value. This behavior is different from the `show()` function:

```
julia> a = 1
1

julia> print(a)
1

julia> show(STDOUT,a)
1

julia> a = '1'
'1'

julia> print(a)
1

julia> show(STDOUT,a)
'1'
```

Similarly, characters and strings can also be printed on the terminal as the following:

```
julia> a = 'z'
'z'

julia> print(a)
z
julia> show(STDOUT,a)
'z'
julia> a = "sandeep nagar"
"sandeep nagar"
```

```
julia> print(a)
sandeep nagar
julia> show(STDOUT,a)
"sandeep nagar"
```

For other kinds of data like complex numbers, rationals, and strings with special characters, the `print()` function can be used. It calls the `show()` function when special considerations for formatting the display needs to be taken care of. Hence, the outputs of `print()` and `show()` are equivalent:

```
julia> a = complex(2,3)
2 + 3im

julia> print(a)
2 + 3im
julia> show(STDOUT,a)
2 + 3im
julia> a = 2//3
2//3

julia> print(a)
2//3
julia> show(STDOUT,a)
2//3
julia> a = "sandeep@nagar"
"sandeep@nagar"

julia> print(a)

sandeep@nagar
julia> show(STDOUT,a)
"sandeep@nagar"
```

The value of variables can be printed within strings using $, as has been explained earlier. The outputs of `print()` and `show()` are obviously different as numeric values are shown as strings in the `show()` function because the second argument was fed as a string data type:

```
julia> a = 2
2

julia> print("$a")
2
julia> show(STDOUT,"$a")
"2"
```

12.6.3 println()

The function `println()` prints the input plus a newline character. This allows users to avoid defining newline characters while giving input and seeing output always in a newline each time a new instance of the `print()` function is called:

```
julia> a = 1
1

julia> print(a)
1
julia> println(a)
1

julia> a = 'z'
'z'

julia> print(a)
z
julia> println(a)
z

julia> a = "sandeep@nagar"
"sandeep@nagar"
```

```
julia> print(a)
sandeep@nagar
julia> println(a)
sandeep@nagar
julia>
```

It is worth noting that in each case, though the output looks similar for both functions, `println()` inserted a newline character after printing the input. This can be observed by the fact that the Julia prompt appears in the next-to-next line after the output from `println()`, but it appears in the next-line when `print()` is used.

12.6.4 display()

The built-in function `display()` simply displays the input using the topmost applicable display in the display stack, typically using the richest supported multimedia output for *x*, with plain-text `STDOUT` output as a fallback. This kind of display can be chosen when the `display(d, x)` variant attempts to display `x` on the given display `d` only. It `throws` a `MethodError` if `d` cannot display objects of a given type. The display units can be connected to the machine on which Julia is installed. Each display unit gets a position as a memory location among display stacks. Thus, they can be called as per requirements. By default, the display is the Julia console:

```
julia> a = 1
1

julia> display(a)
1

julia> a = '1'
'1'

julia> display(a)
'1'

julia> a = "1"
"1"

julia> display(a)
"1"
```

```
julia> a = "sandeep@nagar"
"sandeep@nagar"

julia> display(a)
"sandeep@nagar"
```

12.7 Different Display Units

Section 12.6 concentrated on textual data only, but there can be many kinds of objects that need to be displayed on a variety of display units. Objects like tables of data, graphs, 2D and 3D drawings, maps, photographs, and movies are handled by a variety of display units. Present-day computers are equipped with multimedia consoles that have appropriate hardware and software to stream and interpret multimedia data objects. These objects are usually defined as a file or set of files in a particular format. To deal with these kinds of data objects, Julia must be able to interpret them for inputting and generate them in the right format for outputting. But even after doing the same, the stream must be inputted and/or outputted from the right kind of input and/or output device (which is able to handle that particular kind of data). Hence, knowledge of display units is essential.

In addtion to multimedia consoles, you can have monochrome display units that do not output color and data with a simple black-and-white screen. Some of them cannot handle graphical objects. In some cases, graphics are created using textual information. For example, a line can be made using the `-` symbol, and circles can be represented by the `o` or `O` symbol.

LCD screens exist in a variety of formats ranging from colorful, big billboards to LCD TVs tosimple LCD units with just eight characters in one row. No matter what kind of LCD unit is attached to the machine, its ports of connections, memory locations for its display driver, and so on, must be known to Julia in advance so that they can be used to communicate with Julia code. LED screens follow the same pattern as LCD screens.

3D printers, which can print a 3D object for display, are the newest kind of display units. They are connected using simple USB ports or specialized hardware to a machine. Connecting driver(s) that interpret data from a machine to a 3D printer must be installed properly and then Julia code can be written to give commands to a 3D printer to get streamed data for printing. The data essentially is in the form of a machine code, having information about the controlling printer's motor and filament feeding mechanism (for filament-based 3D printers).

Listing all kinds of display units and their functioning is out of the scope of this book, but we have provided a general description of the basic philosophy pertaining to display units. The major takeaway lesson from this section is that, most often, files are exchanged by Julia code and the machine's operating system for the purpose of interaction with a local or remote machine from which data are streamed in or out. Thus, it is important to study how Julia handles files for this purpose.

12.8 File I/O

Handling files is an essential part of the process of computation. Julia provides many features to perform this act. A file is a group of symbols clubbed together as a unit in a particular format. Files exist in a variety of formats and, hence, any programming language enjoying the capabilities of handling files must provide the functionalities for handling a variety of file formats as well as opening, making, editing, and deleting them as desired, with ease.

UNIX and similar systems treat all computing resources as files, which comprise a computer's peripherals, including the keyboard. Reading keystrokes to input values into a program remains a critical functionality of any programming language. We have already learned that a keyboard is set to be the default value for the global variable `STDIN`. We have also learned that keystrokes can be read by the functions `read()` and `readline()`. The function `read()` can also be used to read files, but each file must be opened first and must be closed after the operation so that it can be opened again. To perform these operations, `open()` and `close()` operations exists.

Before you perform these actions, it is important to know where the file is located (its path), what kind of permissions are allowed for users (permissions to `read`, `write`, and/or `execute`), and whether users have permissions to create, edit, and delete files. It is assumed that readers already have basic knowledge of these concepts since they will not be discussed in great detail here. For learning the same you can refer reference number(s) [1]. Knowledge of Linux commands [2] for handling files also comes in handy when working with Julia programming.

12.8.1 open(), close(), and read()

The `open()` function takes a filename and returns an `IOStream` object that can be used for reading/writing data from/to the file. To work with a file, we need to either create one or open an existing one. The function `open()` needs a string with a path to the file. If a file by that name does not exist,it can be created if the second argument is given as `"w"` for *writing* the file. Please note that following code is tested on MacOS and I believe that it works uniformly on all Unix-based OS.

The following Julia code will write a file in the directory `\tmp` and name it `t.txt` with writing permission. Then ii will create a variable `f`, which references to this file object. Then `f` is used to write into this file object. The string `"A, B, C, D\n"` is fed to this text file object:

```
julia> open("/tmp/t.txt", "w") do f
       write(f, "A, B, C, D\n")
end

11
```

It is important to note that the code has not closed the file after opening the same. Instead, the following code does the same. You must close a file after performing the required actions so that it can be opened again in the same or different session:

```
julia> open("/tmp/t.txt", "w") do f
write(f, "A, B, C, D\n")
close(f)
end
```

Opening the Unix/Linux console and probing the contents of the directory `tmp` allows you to verify that the file has been created by the user (who is logged it, in your case):

```
$ cd /tmp
$ ls
t.txt
$ head t.txt
A, B, C, D
```

The command `ls -l` will let you know what kind of permissions are assigned for this file. Depending on the kind of user, you may have assigned read and/or write and/or execute permissions by default, but you can change these assigments using the Linux command `chmod` [2].

A file with multiple data points in multiple lines can also be created. Suppose you wish to create a file containing five lines comprised of random numbers in a defined string:

```
julia> open("/tmp/t.txt", "w") do f
       for i in 1:5
               random_number = rand()
               write(f, "random number is $random_number \n")

       end
end
```

This will modify the existing file `t.txt` having the data as required. Suppose you wish to write two random numbers per line. They must be separated by a symbol for visual clarity as well as data integrity. Such symbols that perform this task are called delimiters. A very commonly used delimiter is the symbol for a comma (`,`). Let's modify the previous code to include a comma between two random numbers per line:

```
julia> open("/tmp/t.txt", "w") do f
      for i in 1:5
               num1 = rand()
               num2 = rand()
               write(f, "num1= $num1,num2=$num2 \n")
      end
end
```

The same results, except when printing formatted strings, can be achieved by the following code using the `writedlm()` function, which writes with a defined delimiter symbol as one of the arguments:

```
julia> writedlm("/tmp/test.txt", rand(5,2), ", ")
```

12.8.2 Array Reading and Writing

The functions `writedlm()` and `readdlm()` can be used to write arrays into a file and make arrays from contents to the file as follows:

```
julia> random_num = rand(2,3)
2x3 Array{Float64,2}:
0.742948  0.346532  0.385416
0.567728  0.345581  0.553949

julia> writedlm("/tmp/t.txt",random_num)
```

This modifies (and creates in case `t.txt` did not exist in the directory `\\tmp`) the file `t.txt` with contents of the array referenced by th evariable name `random_num`. It is worth noting that a delimiter can be any symbol or group of symbols.

Just as the `writedlm()` function was used to write an array into a file, `readdlm()` reads a file into an array:

```
julia> read_file = readdlm("/tmp/t.txt")
2x3 Array{Float64,2}:
0.742948  0.346532  0.385416
0.567728  0.345581  0.553949

julia> read_file
2x3 Array{Float64,2}:
0.742948  0.346532  0.385416
0.567728  0.345581  0.553949
```

When the delimiter is not defined, a whitespace is used for the same in the previous case since array entries of the same row are delimited by a whitespace. For the `reddlm()` function, the columns are assumed to be separated by one or more whitespaces. The end-of-line delimiter is taken as `\n`. Moreover, if the whole data is numeric, the resultant array is also numeric. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned:

```
julia> writedlm("/tmp/t.txt",random_num,",")

julia> random_num = rand(2,3)
```

```
2x3 Array{Float64,2}:
0.899567  0.195336  0.330642
0.951103  0.915678  0.986845

julia> writedlm("/tmp/t.txt",random_num,",")

julia> read_file = readdlm("/tmp/t.txt")
```

When `readdlm()` is not used with the delimiter argument in the previous case, each row element becomes a string and, as a result, a 2 × 1 array is stored in `read_file`. The actual delimiter in the file `t.txt` is a `,` (comma):

```
julia> read_file = readdlm("/tmp/t.txt",',')
2x3 Array{Float64,2}:
0.899567  0.195336  0.330642
0.951103  0.915678  0.986845
```

The type of data can also be specified as third arguments to ensure the data type of all elements is uniform:

```
julia> read_file = readdlm("/tmp/t.txt",',',Float64)
2x3 Array{Float64,2}:
0.899567  0.195336  0.330642
0.951103  0.915678  0.986845

julia> eltype(read_file)
Float64
```

Since most often data points are separated by a comma, *csv (comma separated vaules)* versions of the functions are used as `writecsv()` and `readcsv()`to handle csv files.

```
julia> read_file = readcsv("/tmp/t.txt")
2x3 Array{Float64,2}:
0.899567  0.195336  0.330642
0.951103  0.915678  0.986845

julia> read_file
2x3 Array{Float64,2}:
0.899567  0.195336  0.330642
0.951103  0.915678  0.986845
```

12.9 Summary

In this chapter, we have dealt with defining the concept of I/O steams of data and Julia's ability to deal with a variety of streams in a flexible manner. Streams of data can be input from a variety of input devices including keyboards, microphones, and video cameras. The behavior of an input device can be objectified in Julia and then defined using methods. The behavior of most output devices—such as a Julia terminal, a graphic terminal, printers, plotters, LCD panels, LED panels, and even 3D printers—is similar. Numerical computing deals with defining tasks in files. Julia's ability to treat data in a file as an I/O stream defines an abstraction of layers that lets a developer define the code with ease. File I/O operations define the backbone of numerical computations. For this reason, this chapter is quite important for users who are serious about making a career in data crunching using Julia.

12.10 Bibliography

[1] https://en.wikipedia.org/wiki/File_system

[2] https://training.linuxfoundation.org/free-linux-training

CHAPTER 13

Plotting

13.1 Introduction to Plotting in Julia

Plotting is an essential part of science and engineering studies. Visualization of an engineering concept leads to better understanding of the phenomenon. Also, in today's world, publications are becoming benchmarks of academic success, and good publications require attractive graphs and animations for showcasing results. For these reasons, plotting 2D and 3D graphs as well as making animations for a given process/equation is an essential part of computational processing and post-processing investigation.

The basic Julia package does not include any functions to make plots. Thus, users need to use a variety of packages for this purpose. Following is a list of packages users can take advantage of:

- Plots
- Pyplots
- GR
- UnicodePlots
- Plotly
- Gadify
- Bokeh

Most packages define plots in the same way. Plotting functions usually takes arrays as inputs and offers a variety of options for decorating the plots such as changing the marker style, marker size, and marker colors; connecting data points with lines of varying sizes; giving a title, label, and legend to graphs; writing equations for labels, legends, or somewhere in between the graphs; and so. This chapter begins with the package `Plots()` and gives a brief introduction to some other packages and their unique features.

13.2 A Plot as an Object

A plot is a graphical object that requires a graphical terminal. On a nongraphical terminal, plots are usually created by placing characters in a specific series. This method is usually not preferred nowadays when graphical terminals have increasingly become widely available at reasonable costs and decent processing speeds, and they have a lot of power. Hence, most Python graphing packages do make plot objects for dealing with graphic terminals at computing machines.

A graphic object has many properties. Following are some of them:

- **Size**: Length and breadth of a plot as it appears on a graphing terminal
- **Shape**: Aspect ratio (the ratio of length and breadth of a plot)
- **Title**: The string declaring the subject of a plot
- **Axes**: The axes that show reference data points
- **Labels**: The labels associated to axes showing a descriptive string
- **Markers**: Symbols of various shapes depicting data points
- **Resolution**: Measured in dpi (dots per inch) because, while printing, the resolution of an image is determined by the ability of the printer to define individual pixels as a dot and the number of dots per inch will define the degree of pixelation of the figure
- **Format of file**: A plot can be stored in many formats reserved for photographs and other kind of media.
- **File name**: A file name provides identification marker for an object in a computing system.

A Julia plot object has a method to deal with all these properties. Using this method, you can modify the features. It is possible to save a profile so that all objects can derive values for their properties from the profile, thus avoiding the need to define them each time a new object needs to be created and also to maintain uniformity.

Furthermore, a plot object is rendered by a graphical engine. Again a variety of graphing engines of varying capabilities can be used. Some are more powerful in terms of their abilities to produce plots of richer properties and producing them faster. While working with different packages, some graphing engines provide a way to define a particular engine for rendering at the back end. Fixing this aspect alleviates worries about uniformity of configurations, quality, and formats of graphical objects.

13.3 Plots Package

`Plots` is a high-level plotting package. It provides powerful graphing capabilities that are usually desired for most of the high-performance computing requirements. It interfaces with other plotting packages (referred to as *back ends* or *graphic engines*) to produce graphics files in a flexible manner. Each of these graphic back ends also can perform as stand-alone plotting packages, but `Plots` provides a user-friendly, simple, and consistent interface.

Before usage, a package must be imported into the present Julia session using the `import` <*packageName*> command. The following command is one of the first commands for using the `Plots` package:

```
# If Plots package is not installed
julia> using Plots
ERROR: ArgumentError: Module Plots not found in current path.
Run `Pkg.add("Plots")` to install the Plots package.
Stacktrace:
[1] _require(::Symbol) at ./loading.jl:428
[2] require(::Symbol) at ./loading.jl:398

julia> Pkg.add("Plots") # Plots is installed
#Long list of outputs is suppressed here.

julia> using Plots # First time
INFO: Precompiling module Reexport.
INFO: Precompiling module StaticArrays.
INFO: Precompiling module RecipesBase.
```

```
INFO: Precompiling module PlotUtils.
INFO: Precompiling module PlotThemes.
INFO: Precompiling module Showoff.
INFO: Precompiling module StatsBase.
INFO: Precompiling module NaNMath.
INFO: Precompiling module Requires.

julia> using Plots # Second time onwards

julia> x = Array([1,2,3,4,5])
5-element Array{Int64,1}:
1
2
3
4
5

julia> y = x.^2 #vectorized power to array x
5-element Array{Int64,1}:
1
4
9
16
25

julia> plot(x,y)
```

Figure 13-1 shows the result, which encompasses the screenshot of a figure window that appears on the graphic terminal of a computer.

It is worth noting that the colors and resolution of windows will be dictated by the configuration of the graphic terminal inside the operating system of the user's machine, but it will appear similar to that in Figure 13-1. The top bar presents the Close, Expand, and Hide buttons while the lower bar presents the Home, Move left, Move right, Shift with mouse, Zoom in and zoom out, Change plot properties, and Graphic object properties buttons. Users are encouraged to click the tabs and explore each one. Explaining them is a wasteful exercise because they are quite intuitive.

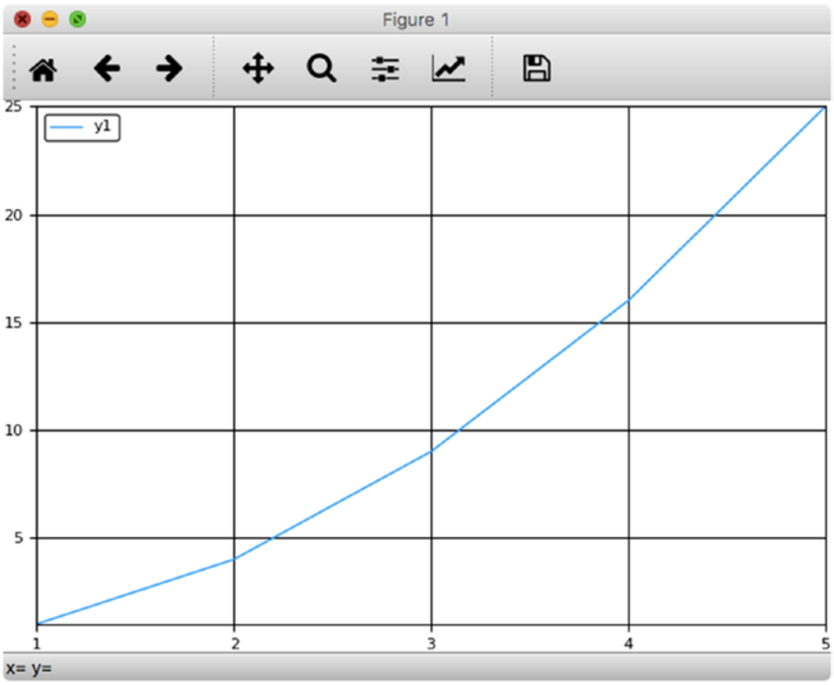


Figure 13-1. *Plot of x vs.* $y = x^2$

Since there are only five data points, the plotted graph isn't a very smooth, straight line connected with the data points. Ideally

$$y = x^2$$

is a smooth curve. To produce a graphically accurate curve, more data points are needed, so the Julia code is modified as follows:

```
julia> x = 1:10e4;

julia> y = x.^2;

julia> plot(x,y)
```

Now instead of five data points, we have 10^4 data points, which results in Figure 13-2.

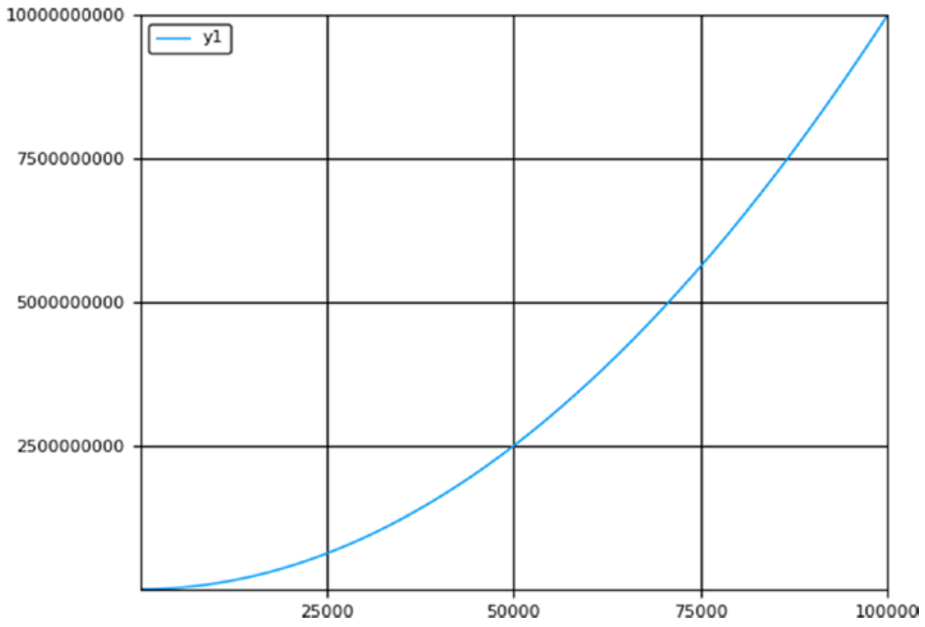


Figure 13-2. *Plot of x vs. y = x^2*

The `plot()` function can also evaluate a mathematical expression, so the following code would also result in the same figure as Figure 13-2.

```
julia> x=1:10e4;
```

```
julia> plot(x,y)
```

13.3.1 Default Behavior of Plots

Figure 13-2 is obtained using the default behavior of plot properties with the default engine named `PyPlots`. Other plotting engines can also be chosen for changing the behavior. As an example, the `unicodeplots()` can be invoked and used as follows:

```
julia>Pkg.add("UnicodePlots")
```

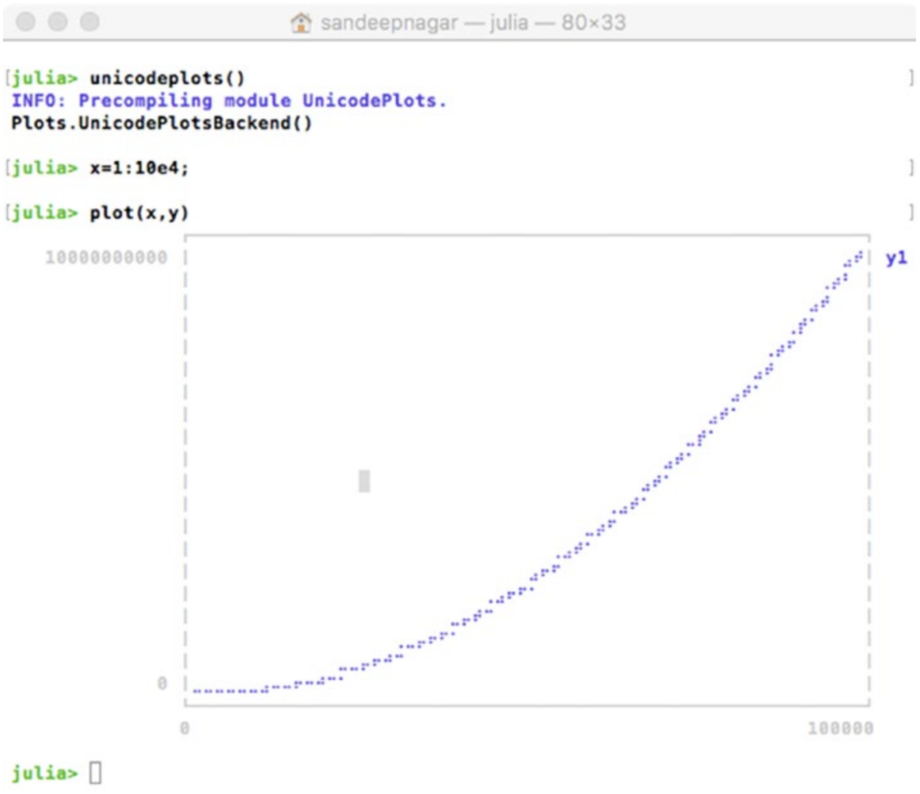
```
julia>Pkg.build("UnicodePlots")
```

```
julia> unicodeplots()# first time usage
INFO: Precompiling module UnicodePlots.
Plots.UnicodePlotsBackend()
```

```
julia> unicodeplots()# second time onwards
Plots.UnicodePlotsBackend()

julia> x=1:10e4;
julia> plot(x,y)
```

The plot is not shown in a separate window, but on the Julia terminal itself, as depicted in Figure 13-3.



```
julia> unicodeplots()
INFO: Precompiling module UnicodePlots.
Plots.UnicodePlotsBackend()

julia> x=1:10e4;

julia> plot(x,y)
```

```
julia>
```

Figure 13-3. *Unicode Plot of x vs. $y = x^2$*

Since PyPlots is a good option for general purpose usage of producing good quality plots, it is a good idea to include the following line of code at the beginning to ensure PyPlot is set as the back end:

```
julia> pyplot()
Plots.PyPlotBackend()
```

13.3.2 Simpler Way to Plot Equations

The `plot()` command presents a simple way to plot mathematical equations. Suppose you wished to plot the graph depicted in Figure 13-4.

$$y = sin(x) + sin(2x)$$

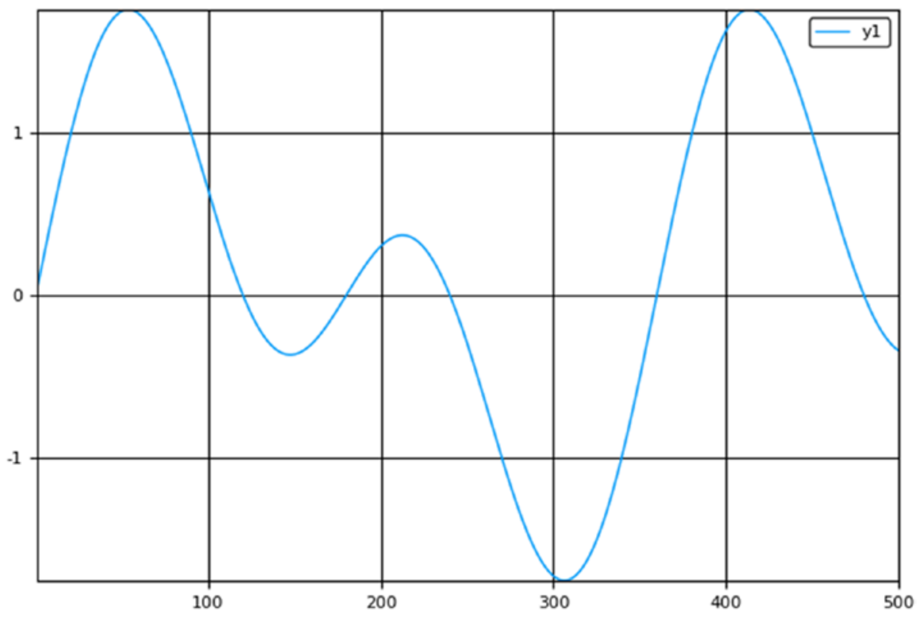


Figure 13-4. *x vs. y = sin(x) + sin(2x)*

The Julia code for performing this task is as follows:

```
julia> eq(x) = sind(x) + sind(2x)
eq (generic function with 1 method)

julia> plot(equation, 1:500)
```

The resulting graphs using this code looks like Figure 13-4

13.3.3 Implicitly Passing a Second Plot

Julia provides a way to implicitly pass an argument for the second plot arguments to exist by using `plot!()` syntax, which changes the original plot since the `!` version modifies the existing object produced by the `plot()` command. The following code will produce the graph shown in Figure 13-5:

```
julia> eq(x) = sind(x) + sind(2x)
eq (generic function with 1 method)

julia> plot(eq,1:500)

julia> eq1(x) = sind(2x)+sin(3x)
eq1 (generic function with 1 method)

julia> plot!(eq1,1:500)
```

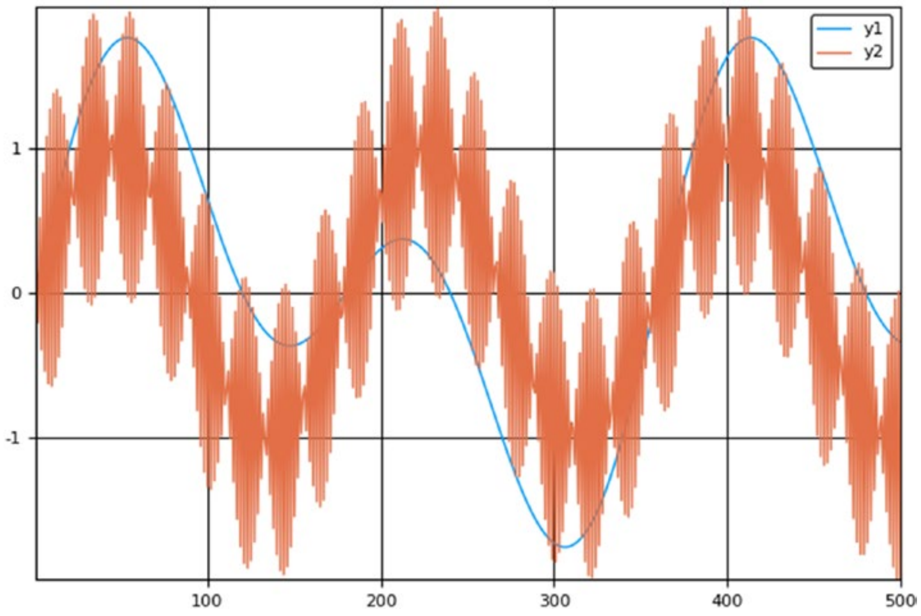


Figure 13-5. *x v.s y = sin(x) + sin(2x) and y = sin(2x) + sin(3x)*

Thus, two plots are produced together on the same figure window where the second plot command is passed to the exiting plot window implicitly.

13.3.4 Decorating the Plots

Documentation on the `Plots` web page [1] gives a detailed overview of various options for decorating the plot with information in a meaningful manner. The following Julia code performs some of the most relevant tasks and produces the graph in Figure 13-6. Users are encouraged to read documentation to learn more.

```
julia> eq1(x) = sind(x) + sind(2x)
eq1 (generic function with 1 method)

julia> eq2(x) = sind(x) + sind(3x)
eq2 (generic function with 1 method)

julia> plot(eq1,
        1:10:500,
        label = "sin(x)+sin(2x)",
        line =(:black,0.9,3, :dot))

julia> plot!(eq2,
       1:10:500,

label = "sin(x)+sin(3x)",
line =(:black,0.7,3, :solid)
size=(800, 600)
)
```

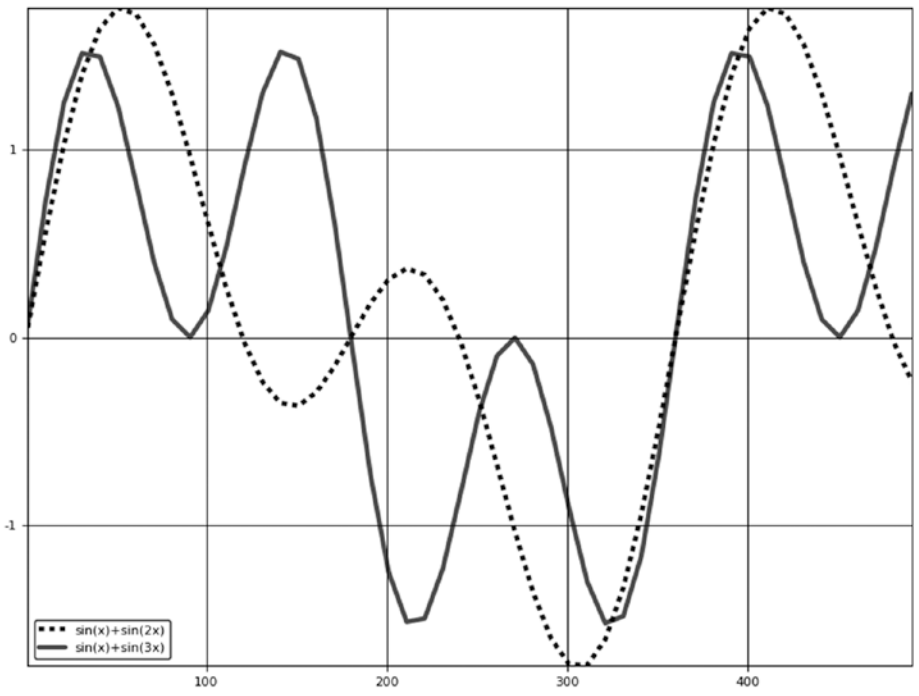


Figure 13-6. *x vs. y = sin(x) + sin(2x) and y = sin(2x) + sin(3x) with some decoration*

13.3.5 Many Plots in the Same Window Using subplot()

A number of plots can be plotted within the same plot window using the `subplot()` window. Plots with these configuration are treated as a matrix of graphs. For example, let's consider plotting four graphs as a matrix of 2 × 2 graphs. Suppose we wish to plot `sin(x)` in the first graph (the graph element indexed for subplot as `subplot(221)`, `sin(2x)` in the second graph; the graph element indexed for subplot as `subplot(222)`, `sin(3x)` in the third graph; the graph element indexed for subplot as `subplot(223)`, `sin(4x)` in the fourth graph; the graph element indexed for subplot as `subplot(224)`). The following Julia code does this work and a graph figure is obtained as illustrated in Figure 13-7:

```
julia> x = -4pi:pi/100:4pi;

julia> y1 = sin(x);

julia> y2 = sin(2x);
julia> y3 = sin(3x);

julia> y4 = sin(4x);

julia> fig = PyPlot.figure(
         "pyplot_subplot_mixed",
         figsize=(10,10),
         dpi=200);

julia> PyPlot.subplot(221);

julia> PyPlot.plot(x,y1);

julia> PyPlot.subplot(222);

julia> PyPlot.plot(x,y2);

julia> PyPlot.subplot(223);

julia> PyPlot.plot(x,y3);

julia> PyPlot.subplot(224);

julia> PyPlot.plot(x,y4);
```

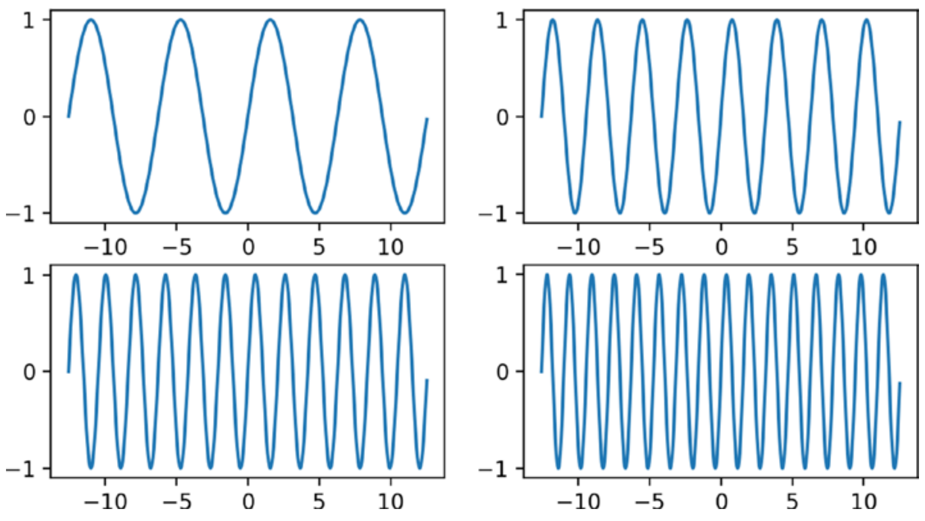


Figure 13-7. *Subplots of sin(x), sin(2x), sin(3x), and sin(4x)*

13.3.6 Histograms

Histograms are graphs where data are sampled into bins and numbers of data points belonging to particular bins are plotted. Histograms are useful in statistics. `Plots` functions has a method named `histogram()` that can be used as follows:

```
julia> x = randn(100000);

julia> Plots.histogram(x)
```

First, 100,000 normalized random numbers are generated using the `randn()` function and then they are plotted using the `histogram()` function within `Plots`. The resulting figure is shown in Figure 13-8. The bell shaped curve verifies that numbers are indeed normally distributed.

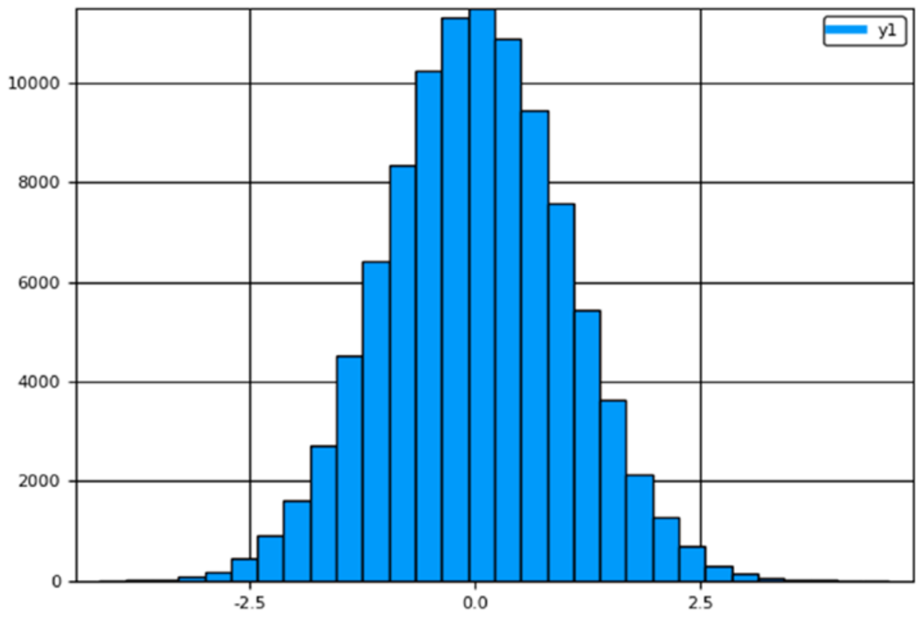


Figure 13-8. *Histogram of random numbers generated using rand() function*

13.3.7 Bar Charts

Bar charts show vertical bars as per data for the y-axis. Let's experiment with 20 random numbers. The following Julia code below does the job and Figure 13-9 is the output:

```
julia> x = rand(20);

julia> Plots.bar(x)
```

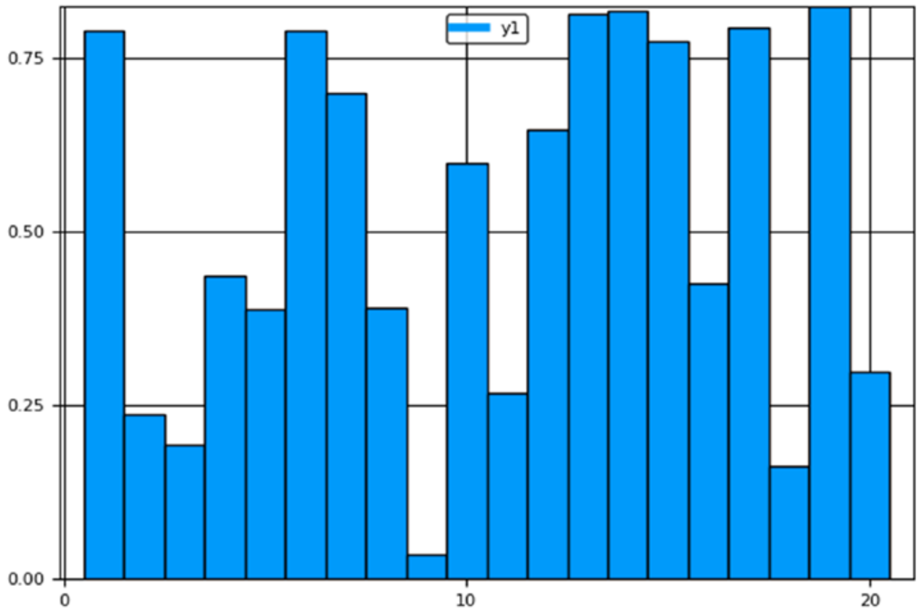


Figure 13-9. *Bar chart of 20 random numbers generated using the rand() function*

13.3.8 Pie Charts

Pie charts depict a "pie" (a circle) whose area is proprtional to the data value. Let's experiment with 10 random numbers. The following Julia code does the job and Figure 13-10 is the output:

```
julia> x = rand(10);

julia> Plots.pie(x)
```

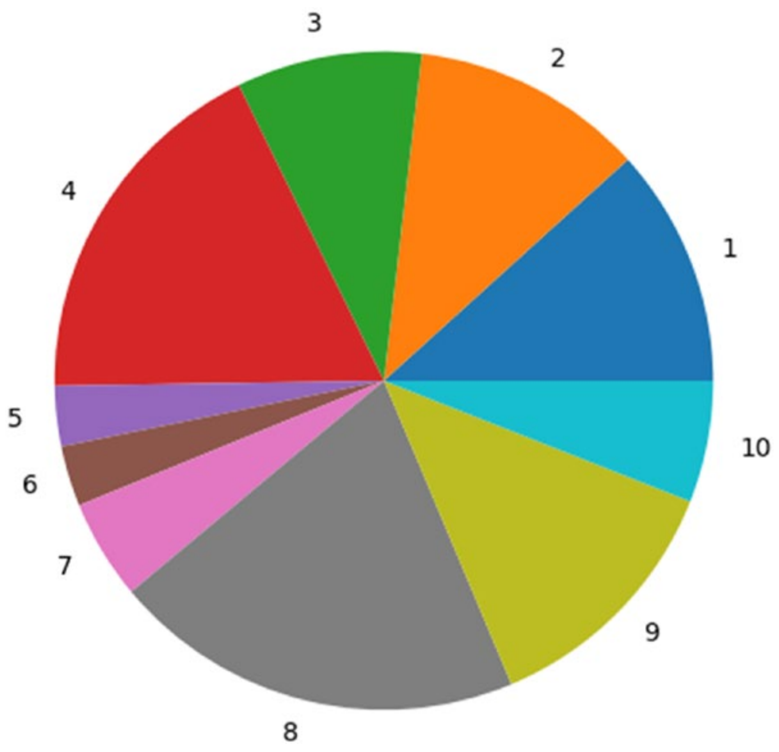


Figure 13-10. *Pie Chart of 10 random numbers generated using rand() function*

13.3.9 Scatter Plots

Scatter plots just put a dot for *x* and *y* data at the coordinate made by *x* and *y*. In our example, x-axis has linearly distributed 100 numbers from 1 to 100 and y-axis has 100 random numbers. The following Julia code performs the job of plotting a scatter plot, as shown in Figure 13-11:

```
julia> x=1:100;

julia> y=randn(100);

julia> Plots.scatter(x,y)
```

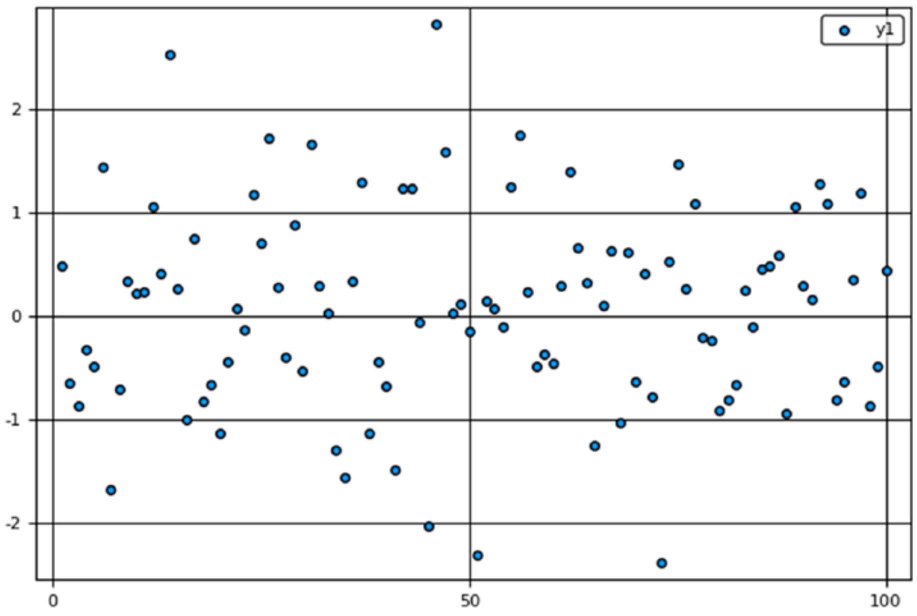


Figure 13-11. *Scatter plot of 100 random numbers generated using the rand() function*

13.4 3D plots

`plot3d()` can be used to plot 3D plots that takes three arguments. Here *x* and *y* variables make the plans on which *z* is defined. In the following Julia code,

$$z = sin(x) + sin(y)$$

and the resultant figure is shown in Figure 13-12.

```
julia> x = 1:0.01:10;

julia> y = 1:0.01:10;

julia> z = sin(x)+sin(y);

julia> Plots.plot3d(x,y,z);
```

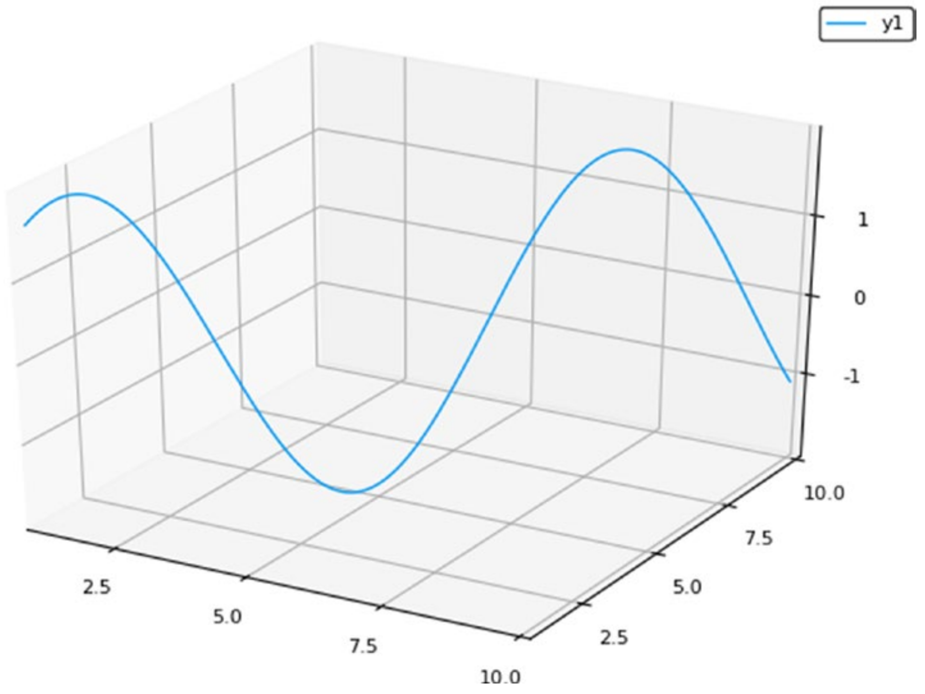


Figure 13-12. *3D plot using the plot3d() function*

13.5 Summary

In this chapter, we have described Julia's ability to process data and effectively visualize the results as plots. The variety of plots and their easy definitions are one of the key features of Julia. This chapter has also dealt with a computational task that is not dealt with by the base package of Julia but by an external package. Hence, this chapter has also demonstrated how to work with external packages and defined their configurations. It is important to note that explaining the plotting capability of Julia could be a book in and of itself. For this reason, this chapter has just given a glimpse of these activities with just one of the options in terms of the `Plots` package. It leaves users the opportunity to explore the rest of the options and make informed decisions.

13.5.1 Bibliography

[1] https://juliaplots.github.io/

CHAPTER 14

Metaprogramming

14.1 Introduction

Metaprogramming is one of the most powerful features of the Julia programming language. Crudely speaking, metaprogramming is about Julia code controlling other parts of source files to an extent that it can modify them and control their execution. To understand this process, you must understand the way Julia code is executed. Broadly speaking, there are two stages of Julia code execution:

- Making an Abstract Syntax Tree
  - Julia code is parsed in the form that is suitable for evaluation.
- Evaluation
  - The parsed code is executed by the compiler.

Metaprogramming is about modifying the code after it has been parsed but before it has been executed. This feature proves to be very useful because you can write short pieces of code that can perform the tedious job of writing bigger pieces of code as per a given rule.

14.2 The : operator

Each Julia task is treated as an expression. REPL evaluates these expressions. To perform metaprogramming, you must be able to stop Julia from evaluating an expression. This is done by the : operator. Let's check out its usage in the following Julia code:

```
julia> a = "Hello"
"Hello"
```

© Sandeep Nagar 2017
S. Nagar, *Beginning Julia Programming*, https://doi.org/10.1007/978-1-4842-3171-5_14

```
julia> :a
:a

julia> b = "1.0"
"1.0"

julia> :b
:b

julia> :(3.5^2)
:(3.5 ^ 2)
```

The output is shown as `:a`, `:b`, and `:(3.5 ^ 2)`. These are termed as *symbols*. These symbols are unevaluated pieces of code ready for modification.

Symbols can be formed alternatively by enclosing the expression within the `quote` and `end` keywords. The following example demonstrates this concept:

```
julia>a= quote
                3.5^2
        end
quote # REPL[437], line 2:
3.5 ^ 2
end

julia> typeof(a)
Expr

julia>  eval(a) 12.25

julia>  3.5^2 12.25

julia> :(3.5^2)
:(3.5 ^ 2)
```

The variable a stores an `Expr` (expression) object. This can be *evaluated* using the `eval()` function. Both the form of defining a symbol `:(3.5 ^ 2)` can be used. The `quote` and `end` keywords are used in multiline usage and the `:` operator is usually used for a single-line usage.

14.3 Expressions

Let's look at an expression by probing it with Julia. For this purpose, let's first construct a simple expression for a piece of code that calculates the hypotenuse of a right-angled triangle when its perpendicular side is give as 2 units and its base side is given as 3 units.

```
julia> A = quote
                p=3
                b=4
                h=sqrt(p^2+b^2)
        end
quote  # REPL[442], line 2:
p = 3 # REPL[442], line 3:
b = 4 # REPL[442], line 4:
h = sqrt(p ^ 2 + b ^ 2)
end
```

```
julia> eval(A) # evaluaing A
5.0
```

This expression will be used henceforth for probing its properties and construction.

14.3.1 fieldnames() and dump()

The `fieldnames()` function can be used to see the structure of this expression stored in `A`. The complete structure, the Abstract Syntax Tree, can be obtained using the `dump()` function.

```
julia> fieldnames(A)
3-element Array{Symbol,1}:
:head
:args
:typ
```

```
julia> dump(A)
Expr
        head: Symbol block
        args: Array{Any}((6,))
```

```
1: Expr
      head: Symbol line
      args: Array{Any}((2,))
      1: Int64 2
      2: Symbol REPL[442]
      typ: Any
2: Expr
      head: Symbol =
      args: Array{Any}((2,))
      1: Symbol p
      2: Int64 3
      typ: Any
3: Expr
      head: Symbol line
      args: Array{Any}((2,))
      1: Int64 3
      2: Symbol REPL[442]
      typ: Any
4: Expr
      head: Symbol =
      args: Array{Any}((2,))
      1: Symbol b
      2: Int64 4
      typ: Any
5: Expr
      head: Symbol line
      args: Array{Any}((2,))
      1: Int64 4
      2: Symbol REPL[442]
      typ: Any
6: Expr
      head: Symbol =
      args: Array{Any}((2,))
      1: Symbol h
      2: Expr
```

```
                                                  head: Symbol call
                                                  args: Array{Any}((2,))
                                                  1: Symbol sqrt
                                                  2: Expr
                                                          head: Symbol call
                                                          args: Array{Any}((3,))
                                                          1: Symbol +
                                                          2: Expr
                                                          3: Expr
                                          typ: Any
                                  typ: Any
                         typ: Any
typ: Any
```

The output is an array of expressions that are subexpressions of the original expression stored in the variable name `A`. Each subexpression has a `head`, `Expr`, and `typ`, which define a block of code, its subexpression, and the types included. This is the Abstract Syntax Tree (AST). Let's try to probe the arguments of the `Symbol block`:

```
julia> A.args[1]
:( # REPL[442], line 2:)

julia> A.args[2]
:(p = 3)

julia> A.args[3]
:( # REPL[442], line 3:)

julia> A.args[4]
:(b = 4)

julia> A.args[5]
:( # REPL[442], line 4:)

julia> A.args[6]
:(h = sqrt(p ^ 2 + b ^ 2))
```

Alternatively, the `for` loop can be used to print the `args` list:

```
julia> for (n, expr) in enumerate(A.args)
println(n,  ":  ",  expr)
end
1:  # REPL[442], line 2:
2: p = 3
3:  # REPL[442], line 3:
4: b = 4
5:  # REPL[442], line 4:
6: h = sqrt(p ^ 2 + b ^ 2)
```

Now let's edit the `A.args`, which stores the expression `:(h=sqrt(p^2+b^2))`, to another expression, `:(multiply = p * b)`:

```
julia> eval(A) # Before editing
5.0

julia> A.args[end]= :(multiply=p*b)
:(multiply = p * b)

julia> eval(A) # after editing
12
```

This is a simple demonstration of metaprogramming where the subexpression ofJulia code is edited to calculate something entirely different without writing the code for the same.

14.4 Expression Interpolation

For writing an expression in a simpler fashion, you can use the $ interpolation operator in expressions, as we learned for strings (see Chapter 9, Section 9.9). For example, if we set `x=-2` and `y=2`, then the expression `expr=:(x^y))` returns `:(x^y)`, whereas `expr1=:($x^y)` will return `:(-2 ^ y)` while `expr2=:(x^$y))` will return `:(x ^ 2)`. All of these evaluate to the same value—4. This is shown in the following Julia code:

```
julia> x=-2
-2
```

```
julia> y=2
2

julia> expr=:(x^y)
:(x ^ y)

julia> expr1=:($x^y)
:(-2 ^ y)

julia> expr2=:(x^$y)
:(x ^ 2)

julia> eval(expr)
4

julia> eval(expr1)
4

julia> eval(expr2)
4
```

One important feature of such an interpolation is that the expression evaluation evaluates at *parse time*, whereas other interpolations evaluate only when the `eval()` function is called after parse time.

14.5 Macros

Using a `macro`, you can generate a new output expression from an unevaluated input expression. These expressions are evaluated at parse time and return an unevaluated expression. They are like functions except for the fact that they map an input expression to an output expression.

Syntax of a macro is as follows:

```
macro name_of_macro
# body of macro
end
```

A macro is invoked by placing a @ before the name without a whitespace and then passing an expression such as the following:

```
julia> @name_of_macro expr1,expr2

# alternative way of calling
julia> @name_of_macro(expr1,expr2)
```

Let's understand this form of definition and usage with the following Julia code. Here a macro named expFeatures is defined that prints the args, head, and typ for an expression. It also evaluates the expressions and returns the evaluated value.

```
julia> macro expFeatures(expression)
        if typeof(expression)==Expr
                println(expression.args)
                println(expression.head)
                println(expression.typ)
        end
answer=eval(expression)
return answer
end
@expFeatures (macro with 1 method)

julia> @expFeatures 3+4-5
Any[:-, :(3 + 4), 5]
Call
Any
2

julia> @expFeatures (3+4)-(5^2)
Any[:-, :(3 + 4), :(5 ^ 2)]
Call
Any
-18
```

```
julia> @expFeatures sin(90),sind(90)
Any[:(sin(90)), :(sind(90))]
Tuple
Any
(0.8939966636005579, 1.0)
```

When the macro `@expFeatures` is fed expressions `3+4-5`, then `args` is found to be `Any[:-, :(3 + 4), 5]`, `head` is found to be `call`, and `typ` is found to be `Any`. Also the expression is evaluated as 2. Similarly other expressions can be fed to this macro to study the features of the same. Two expressions (`sin(90` and `sind(90)`) are fed in the last attempt separarted by a comma (which is why `head` becomes `tuple`) and the evaluation is also done accordingly.

14.6 Built-in Macros

A lot of macros are predefined in the Julia compiler. Some of them will be discussed here. One of the most preferred is to time the execution of a code. Using `@time` for an expression, you can obtain the following:

- A macro to execute an expression
- The time it took to execute
- The number of allocations
- Total number of bytes its execution caused to be allocated
- Returning the value of the expression

```
julia> x=1:10e4;

julia> @time x.^3
1.024123 seconds (63.48 k allocations: 4.167 MiB)
100000-element  Array{Float64,1}:
1.0
8.0
27.0
64.0
125.0
```

```
216.0
343.0
512.0
729.0
1000.0
.
.
.
9.9976e14
9.9979e14
9.9982e14
9.9985e14
9.9988e14
9.9991e14
9.9994e14
9.9997e14
1.0e15
```

This shows that the execution took 1.024123 seconds, 63.48K allocations, and a memory occupation of 4.167 MiB; then the partial display of the cube of each array element is shown. Users can also experiment with `@timev`, `@timed`, `@elapsed`, and `@allocated` and check thier usage using the `help>?` mode.

14.7 Summary

In this chapter, we have presented the concepts of a symbol, expression, interpolation of expressions, and macros to introduce the concept of metaprogramming in Julia. Having easy ways to perform metaprogramming in Julia is one of its most attractive features for developers. A number of macros has been developed and released in a similar fashion. The practices of making useful macros and using them judiciously are considered key skills for a Julia programmer.

Index