

Julia Programming

Module 1: Understanding Julia's Ecosystem

Learning objectives:

1. How Julia is different and looks at some of its features.
2. How to download Julia and set up its environment on your system.
3. Different environments which are popularly available for Julia, such as REPL, Jupyter notebook, and Juno (Atom).
4. Do a few exercises.

What makes Julia unique?

- Scientific computations require the high computing requirements. The requirement was to build a language, that is easy to read and code in, like Python, which is a dynamic language and gives the performance of C, which is a statically typed language.
- In 2012, a new language emerged-Julia. It is a general-purpose programming language which is highly suited for scientific and technical computing. Julia's performance is comparable to C/C++ measured on different benchmarks available on the JuliaLang's homepage and simultaneously provides an environment that can be used effectively for prototyping, like Python.
- Julia is able to achieve such performance because of its design and Low-Level Virtual Machine (LLVM)-based just-in-time (JIT) compiler.

Features and advantages of Julia

Julia is really good at scientific computing but is not restricted to just that, as it can also be used for web and general purpose programming.

Some of Julia's features are mentioned as follows:

1. It is designed for distributed and parallel computation.
2. Julia provides an extensive library of mathematical functions with great numerical accuracy.
3. Julia gives the functionality of multiple dispatches. It will be explained in detail in the coming chapters. Multiple dispatches refer to using many combinations of argument types to define function behaviors. Julia provides efficient, specialized, and automatic generation of code for different argument types.
4. The Pycall package enables Julia to call Python functions in its code and MATLAB packages using the MATLAB.jl package. Functions and libraries written in C can also be called directly without any need for APIs or wrappers.
5. Julia provides powerful shell-like capabilities for managing other processes in the system.
6. Unlike other languages, user-defined types in Julia are compact and quite fast as built-ins.
7. Scientific computations make great use of vectorized code to gain performance benefits. Julia eliminates the need to vectorize code to gain performance. Devectorized code written in Julia can be as fast as the vectorized code.
8. It uses lightweight green threading, also known as tasks or coroutines, cooperative multitasking, or one-shot continuations.
9. Julia has a powerful type system. The conversions provided are elegant and extensible.
10. It has efficient support for Unicode.
11. It has facilities for metaprogramming and Lisp-like macros.
12. It has a built-in package manager (Pkg).

One great feature of Julia is that it solves the 2-language problem. Generally, with Python and R, code that is doing most of the heavy workload is written in C/C++ and it is then called. This is not required with Julia, as it can perform comparably to C/C++. Therefore, complete code—including code that does heavy computations—can be written in Julia itself.

Installing Julia

As mentioned earlier, Julia is open source and is available for free. It can be downloaded from the website at <http://julialang.org/downloads/>.

Julia (command line version)

Windows Self-Extracting Archive (.exe)	32-bit		64-bit	
macOS Package (.dmg)	10.7+ 64-bit			
Generic Linux binaries	32-bit (X86) (GPG)		64-bit (X86) (GPG)	
Linux builds for other architectures	ARMv7 32-bit hard float (GPG)		PowerPC 64 little endian (GPG)	
Source	Tarball (GPG)	Tarball with dependencies (GPG)		GitHub

It is highly recommended to use the generic binaries for Linux provided on the julialang.org website. As Ubuntu and Fedora are widely used Linux distributions, a few developers were kind enough to make the installation on these distributions easier by providing it through the package manager. We will go through them in the following sections.

Julia on Ubuntu (Linux)

Ubuntu and its derived distributions are one of the most famous Linux distributions. Julia's deb packages (self-extracting binaries) are available on the website of JuliaLang, mentioned earlier. These are available for both 32-bit and 64-bit distributions. One can also add the Personal Package Archive (PPA), which is treated as an apt repository to build and publish Ubuntu source packages. In the Terminal, type the following commands:

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

This adds the PPA and updates the package index in the repository. Now install Julia using the following command:

```
$ sudo apt-get install Julia
```

The installation is complete. To check whether the installation is successful in the Terminal, type the following:

```
$ julia --version
```

This gives the installed Julia's version:

```
$ julia version 0.5.0
```

To uninstall Julia, simply use apt to remove it:

```
$ sudo apt-get remove julia
```

Julia on Fedora / CentOS / Red Hat (Linux)

For Fedora/RHEL/CentOS or distributions based on them, enable the EPEL repository for your distribution version. Then, click on the link provided. Enable Julia's repository using the following:

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

Or copy the relevant .repo file available at:

```
/etc/yum.repos.d/
```

Finally, in the Terminal type the following:

```
$ yum install julia
```

Julia on Windows

Go to the Julia download page (<https://julialang.org/downloads/>) and get the .exe file provided according to your system's architecture (32-bit/64-bit). The architecture can be found in the property settings of the computer. If it is amd64 or x86_64, then go for 64-bit binary (.exe), otherwise, go for 32-bit binary. Julia is installed on Windows by running the downloaded .exe file, which will extract Julia into a folder. Inside this folder is a batch file called julia.exe, which can be used to start the Julia console.

Julia on Mac

Users with macOS need to click on the downloaded .dmg file to run the disk image. After that, drag the app icon into the Applications folder. It may prompt you to ask if you want to continue, as the source has been downloaded from the internet and so is not considered secure. Click on Continue if it was downloaded from the official Julia language website.

Julia can also be installed using Homebrew on a Mac, as follows:

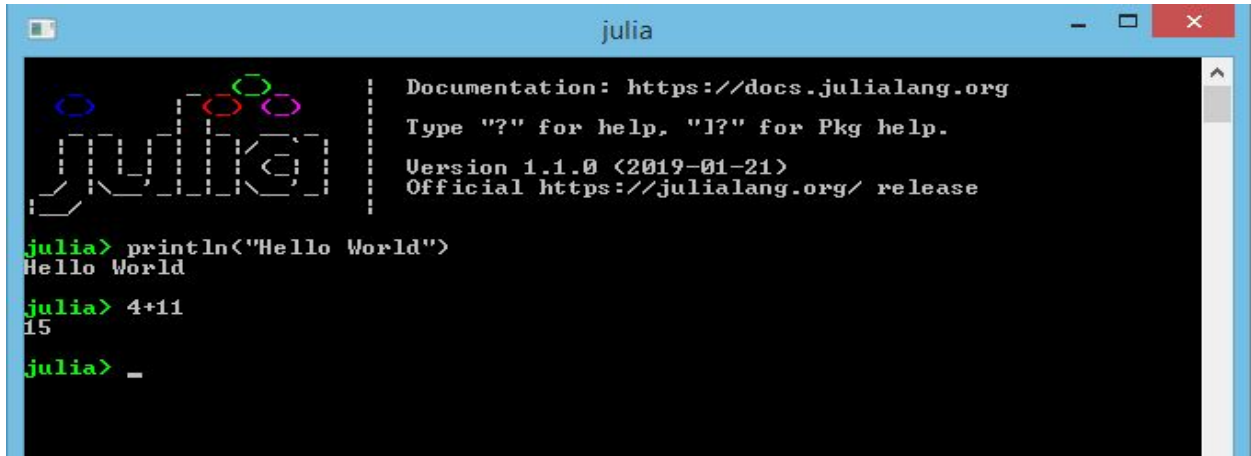
```
$ brew update
$ brew tap staticfloat/julia
$ brew install julia
```

The installation is complete. To check whether the installation is successful in the Terminal, type the following:

```
$ julia --version
```

Using REPL

Read-Eval-Print-Loop (REPL) is an interactive shell or the language shell that provides the functionality to test out pieces of code. Julia provides an interactive shell with a JIT compiler (used by Julia) at the backend. We can give input in a line, it is compiled and evaluated, and the result is given in the next line:



```
julia

Documentation: https://docs.julialang.org
Type "?" for help, "I?" for Pkg help.
Version 1.1.0 (2019-01-21)
Official https://julialang.org/ release

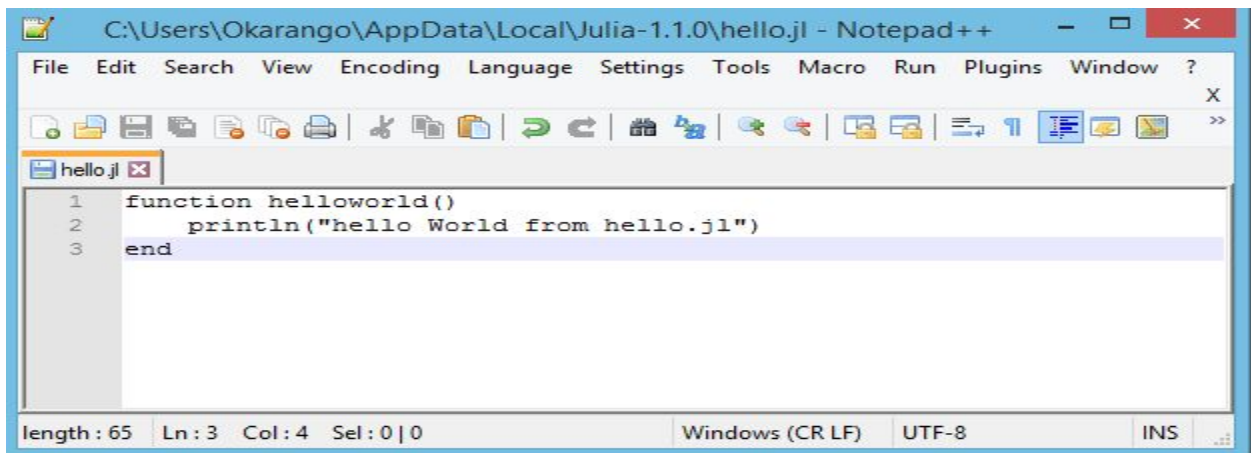
julia> println("Hello World")
Hello World

julia> 4+11
15

julia> _
```

Julia's shell can be started easily, just by writing Julia in the Terminal. This brings up the logo and information about the version. This `julia>` is a Julia prompt and we can write expressions, statements, and functions, just as we could write them in a code file. The benefit of having the REPL is that we can test out our code for possible errors. Also, it is a good environment for beginners. We can type in the expressions and press Enter key to evaluate them. A Julia library, or custom-written Julia program, can be included in REPL using `include`.

For example, let's create a file in notepad++ with the name `hello.jl` and write a function with the name `helloworld()`, which just prints the line `hello World` from `hello.jl`:



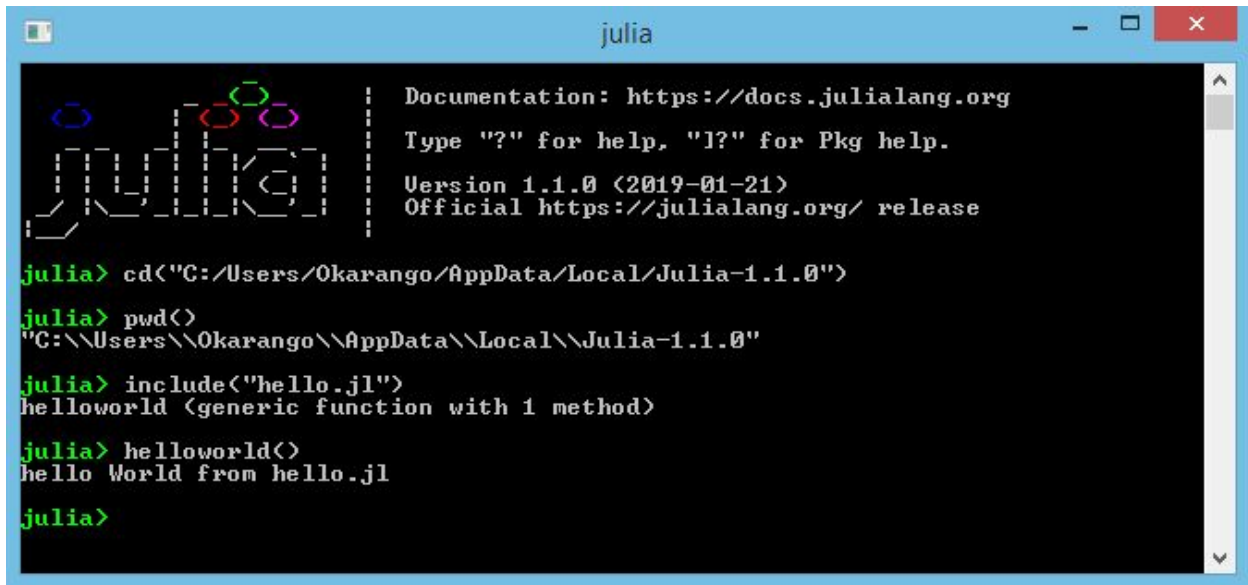
```
C:\Users\Okarango\AppData\Local\Julia-1.1.0\hello.jl - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

1 function helloworld()
2     println("hello World from hello.jl")
3 end

length: 65 Ln: 3 Col: 4 Sel: 0 | 0 Windows (CR LF) UTF-8 INS
```

Now, use this function inside the REPL by writing an `include` statement for the specified file:

A screenshot of a Julia REPL window titled 'julia'. The window has a blue title bar and a black background. On the left, there is a colorful logo of the word 'julia' in a stylized font. To the right of the logo, white text provides documentation links, help instructions, version information (1.1.0), and the official website. The REPL prompt 'julia>' is shown in green. The user enters a series of commands: 'cd' to change the directory to 'C:\Users\Okarango\AppData\Local\Julia-1.1.0', 'pwd' to confirm the current directory, 'include' to load 'hello.jl', and 'helloworld' to call the function. The output shows the directory path, the function signature 'helloworld <generic function with 1 method>', and the result 'hello World from hello.jl'.

When we include the file inside the REPL, we see that it gave the information about the function present inside the file. We used the function and it printed the desired line.

Using Jupyter Notebook

Data science and scientific computing are privileged to have an amazing interactive tool called Jupyter Notebook. With Jupyter Notebook, you can write and run code in an interactive web environment that also has the capability to have visualizations, images, and videos. It makes the testing of equations and prototyping a lot easier, has the support of over 40 programming languages, and is completely open source.

GitHub supports Jupyter Notebooks (static). A Notebook with a record of computation can be shared through the Jupyter Notebook viewer or other cloud storage. Jupyter Notebooks are extensively used for coding machine-learning algorithms, statistical modeling, and numerical simulation, and data munging. Jupyter Notebook is implemented in Python, but you can run the code in any of the 40 languages, provided you have their kernel. You can check whether Python is installed on your system or not by typing the following into the Terminal:

```
$ python --version
Python 2.7.12 :: Anaconda 4.1.1 (64-bit)
```

This will give the version of Python if it is on the system. It is best to have Python 2.7.x, 3.5.x, or a later version. If Python is not installed, then you can install it by downloading it from the official website for Windows.

It is highly recommended to install Anaconda if you are new to Python and data science. Commonly used packages for data science, numerical, and scientific computing—including Jupyter Notebook—come bundled with Anaconda, making it the preferred way to set up an environment. Instructions can be found at <https://www.continuum.io/downloads>.

Otherwise, for Linux (Ubuntu), typing the following should work:

```
$ sudo apt-get install python
```

Jupyter is present in the Anaconda package, but you can check whether the Jupyter package is up to date by typing in the following:

```
$ jupyter --version
```

If for some reason, it is not present, it can be installed using:

```
$ conda install jupyter
```

Another way to install Jupyter is by using the pip command:

```
$ pip install jupyter
```

Now, to use Julia with Jupyter, we need the IJulia package. This can be installed using Julia's package manager. In the REPL, type the following commands:

```
julia> Pkg.update()  
julia> Pkg.add("IJulia")
```

The first step updates the metadata and current packages and the second step installs the desired package. After installing IJulia, we can create a new Notebook by selecting Julia under the Notebooks section in Jupyter.

Juno IDE Atom is an open source text editor developed using Electron, a framework by GitHub. Electron is a framework designed to build cross-platform apps using HTML, JavaScript, CSS, and Node.js. It is completely configurable and can be modified according to your needs. Basic tutorials are available on their website at <https://atom.io/>. There are thousands of open source packages available for Atom. Most of them are there to add or improve functionality, and there are thousands of themes to change the look and feel of the editor.

Features of Atom:

- Split the editor in panes to work side by side
- Open complete projects to navigate easily
- Autocomplete
- Available for Linux, Mac, and Windows

4. What is Juno?

Juno is built on Atom. It is a powerful and free environment for the Julia language. This contains many powerful features, such as:

- Multiple cursors
- Fuzzy file finding
- Vim key bindings

Juno provides an environment that combines the features of the Jupyter Notebook and the productivity of the IDE. It is very easy to use and is a completely live environment. With Atom, you can install new packages through the Settings panel or through the command line using the `apm` command. So, if we need to install a new package and we know its name (let's say `xyz`), we can just write:

```
$ apm install xyz
```

There are many `apm` commands, which can be read using `--help` command:

```
$ apm --help
```

In the last decade, data science has become a buzzword. A data scientist is required to have an in-depth understanding of the domain he/she is working in. Julia was designed for scientific and numerical computation. And with the advent of big data, there is a requirement to have a language that can work on large amounts of data.

Julia Programming

Module 2 - Programming Concepts

Learning Objectives:

Julia is designed in a way that someone who has just started with programming will be able to be up and running in a day with the help of REPL or Jupyter Notebook. Julia provides lots of features that are useful to data scientists, statisticians, and those working in the field of scientific computing.

The reader will go through the syntax, as well as one of the many ways to program in Julia. The following topics will be covered in this chapter:

- Revisiting programming paradigms
- Starting with Julia REPL
- Variables
- Integers, bits, bytes, and bools
- Floating point numbers in Julia
- Logical and arithmetic operations
- Arrays and matrices
- DataFrames and data arrays

Revisiting programming paradigms

The programming paradigm refers to the breaking up of the programming activity into a structure of thoughts. It is an approach towards a problem and the orientation of sub-tasks. Although a problem can be approached using different paradigms, one paradigm may be more suited to solving it than another.

There are many programming paradigms, so we will only be discussing a few of them here:

- Imperative
- Logical
- Functional
- Object-oriented

Imperative programming paradigm

Imperative programming is a procedural way of programming. The primary focus is on the variables and the sequential execution of the tasks that may change the values of these variables. This paradigm is based on the Von Neumann computer, which has reusable memory, allowing us to change the state.

The imperative paradigm makes the assumption that the computer is capable of maintaining the different states of the variables that are generated during the computation process.

The advantages of using the imperative paradigm:

- It is efficient in utilizing system resources.
- It is based on how a computer functions, and therefore closely resembles the machine.
- There are many popular languages that use this programming paradigm.

There are some disadvantages to using this paradigm:

- Many problems cannot be solved by just following the order of statements.
- There is no referential transparency, which means the state of the variables can change. This makes the program difficult to comprehend.
- Debugging is not straightforward.
- A very limited abstraction can be achieved using the imperative programming paradigm.

Logical programming paradigm

The logical programming paradigm, also referred to as the rule-based paradigm, is based on predicate logic. It is a declarative approach to solving a problem and focuses on relations. Prolog is well suited to logical programming.

This paradigm is well suited to being applied to fields where we are dealing with different facts and the relationships among them. The program handles the data and creates a combination of rules, which gives us a valid logical expression.

Such a program can be divided into three sections:

- Definitions and declarations, which define the domain of the problem.
- Facts that are relevant to the domain and the given problem.
- Queries, which are actually the goals we want to achieve from these logical expressions.

There are some disadvantages to logical programming too:

- Functions that can compute either way do not perform well enough.
- Rule-based programming is restricted to domains that can be expressed well using relations.

Functional programming paradigm

The functional programming paradigm originates from a purely mathematical ideology the theory of functions. It treats all subprograms as functions.

Functions (in a mathematical sense) take an argument list and return outputs after computations. The result is dependent on the computations, which are themselves dependent on the inputs we provide to the functions. The focus is on the values and how the expressions are evaluated, rather than statements, as in the imperative paradigm.

Consecutive states are not valid in the functional paradigm. The result from a function would be an input to another expression and would not be saved as a variable.

The functional paradigm is a cleaner and simpler programming paradigm than others because it follows the mathematical functional theory.

Functions are first-class objects in the functional paradigm. This means that functions can be treated as data and we make the assumption that a function will return a value. This allows us to pass a function as an argument to another function or return a function from any other function.

The characteristics and advantages of the functional paradigm:

- Functions provide a high level of abstraction, which reduces the possibility of committing errors.
- Programs are independent of assignment operations and it is possible to write higher-order functions. This makes the functional programming paradigm good for parallel computations.
- It maintains referential transparency, unlike imperative programming. This makes it more suited to mathematical expressions.
- The values in functional programming are non-mutable.

There are also some disadvantages to the functional programming paradigm:

- It becomes complicated in some situations, usually when there is a need to handle lots of sequential activity, which would be handled better with the imperative or object-oriented paradigm.
- The program may be less efficient than a program written in other paradigms.

Object-oriented paradigm

The object-oriented programming (OOP) paradigm is a representation of real-world entities, where everything is an object and we modify the state of an object using a behavior or methods.

The OOP paradigm places the focus on objects. These objects belong to a particular class. Classes have specific methods that objects can use. As objects are real-world entities, they are encapsulated, containing data and the methods that can change the state of this data.

The object-oriented programming paradigm is based on four major principles:

1. ***Encapsulation***, as the name suggests, restricts access from outside of the object's definition. The methods that have access to the objects can only manipulate their state. This prevents outside methods from changing the state of the object to perform an invalid operation.

2. **Abstraction** is a way of defining conceptual boundaries using classes in terms of interfaces and functionalities. This protects the internal properties of the object.
3. **Inheritance** enables code re-usability. Classes are allowed to inherit attributes and behavior from existing classes, thus eliminating the need to rewrite them. This also helps in consistency as, if there is a change, we are only required to make it in a single place. The derived classes can add their own attributes and behavior, and can therefore extend the functionalities provided by the base class.
4. **Polymorphism** refers to having many forms of the same name. This means can have different methods with the same name:
 - **Overriding**: Is runtime polymorphism, where two methods have the same name and signature. The difference is that one of the methods is in the base class and the other is in the derived class. With overriding, the child class can have the specific implementation of the method.
 - **Overloading**: Is compile-time polymorphism, where there are two or more methods in the same class with the same names but different signatures. The decision of which method will be called is based on the values passed as the arguments to the method.

Starting with Julia REPL

Variables in Julia

Just as in other programming languages, we use a variable to store a value that is obtained from a computation or external source.

```
# assign 100 to a variable x
julia> x = 100
100
# multiple the value in the variable by 5
julia> x*5
500
```

We can change the values stored in a variable or mutate the state:

```
# assign a different value to x
# this will replace the existing value in x
julia> x = 24
24
# create another variable y
julia> y = 10
10
```

The names of variables can start with a character or a "_" (underscore). Julia also allows Unicode names (UTF-8), but not all Unicode names are accepted in the variable name:

```
julia> _ab = 40
40
julia> @ab = 10
ERROR: syntax : unexpected "="
julia> 1000
```

```
1000
```

Julia provides some built-in constants and functions and we can assign values to them, although it is not recommended to do so, as it can create confusion:

```
julia> pi
π = 3.1415926535897...
julia> pi = 300
300
```

There are some reserved keywords, which are not allowed to be used as variable names:

```
julia> if = 1000
ERROR: syntax: unexpected "="
julia> for = 100
ERROR: syntax: unexpected "="
```

Naming conventions

Although Julia doesn't have many restrictions on naming and most combinations are allowed, it is good to follow some conventions as good practices:

- Generally, variable names are written in lowercase.
- Underscores are used to separate different words in a variable name, but it is not advisable to use names that would require underscores.
- Function and macros names are in lowercase. Underscores are not used.
- The first character of types and modules is uppercase. The separation between words in names is done using upper camel case.
- The functions that modify or write to any of their arguments end with "!" symbol.

We mentioned earlier that Julia is a strongly-typed language. Therefore, it is necessary for a variable's type to be defined. If it is not defined explicitly, then Julia will try to infer it from the value assigned to the variable.

We can use the `typeof()` function provided by Julia to find the type of the variable.

```
julia> typeof(_ab)
Int64
julia> langname = "Julia"
"Julia"
julia> typeof(langname)
String
```

It should also be noted that, in Julia, we can separate numbers using underscores:

```
julia> 1_0_0
100
```

Integers, bits, bytes, and bools

Integers, bits, bytes, bools, and floating point numbers are used in arithmetic operations. Built-in representations of them are called as numeric primitives, and numeric literals are their representations as values in code.

Type	Number of bits	Smallest value	Largest value
Int8	8	-2^7	$2^7 - 1$
UInt8	8	0	$2^8 - 1$
Int16	16	-2^{15}	$2^{15} - 1$
UInt16	16	0	$2^{16} - 1$
Int32	32	-2^{31}	$2^{31} - 1$
UInt32	32	0	$2^{32} - 1$
Int64	64	-2^{63}	$2^{63} - 1$
UInt64	64	0	$2^{64} - 1$
Int128	128	-2^{127}	$2^{127} - 1$
UInt128	128	0	$2^{128} - 1$
Bool	8	false (0)	true (1)

The UInt type refers to unsigned integers. These are those integers whose values start from 0.

We can also find the smallest and the largest value of a type of integer using the `typemin()` and `typemax()` function:

```
julia> typemax(Int32)
2147483647
julia> typemin(Int32)
-2147483648
```

Floating point numbers in Julia

It is easy to represent floating point numbers in Julia. They are represented in a similar fashion as they are in other languages:

```
# Add a decimal point
julia> 100.0
100.0

julia> 24.
```

```
24.0
```

```
# It is not required to precede a number from the decimal point
```

```
julia> .10
```

```
0.1
```

```
julia> typeof(ans)
```

```
Float64
```

Exponential notation can be very useful and convenient in various scenarios. It can be used in Julia using e:

```
julia> 2.99e8
```

```
2.99e8
```

```
julia> 2.99e8 > 999999
```

```
True
```

We have been using Float64 in the preceding examples. We can also use Float32 on 64-bit computers if required:

```
# Replace e by f to generate Float32
```

```
julia> 2.99f8
```

```
2.99f8
```

```
# Check the type of the preceding variable if it is Float32
```

```
julia> typeof(ans)
```

```
Float32
```

```
# Compare it with the same value
```

```
julia> 2.99f8 == 2.99e8
```

```
True
```

It is easy to convert values from Float64 to Float32 :

```
julia> Float32(2.99e8)
```

```
2.99f8
```

In some use cases, hexadecimal floating point literals are used. In Julia, they are valid only as Float64 values:

```
julia> 0x4.1p1
```

```
8.125
```

```
julia> typeof(ans)
```

```
Float64
```

Operations on floating point numbers

We regularly come across simple operations on floating point numbers in various programming languages not giving the expected value.

```
julia> x = 1.1; y = 0.1; x+y
1.2000000000000002
```

We can use the `setrounding()` function provided by Julia to handle this:

```
julia> setrounding(Float64, RoundDown) do
x + y
end

1.2
```

Writing expressions with coefficients

It becomes extremely convenient to be able to write mathematical expressions without the need to explicitly write the multiplication operator:

```
# example of expression
julia> x = 4; y = 11;
julia> 3x + 4y + 93
149
# example of expression
julia> x = 4; y = 11;
julia> 3x + 4y + 93
149
```

Logical and arithmetic operations in Julia

Logical and arithmetic operations are very similar to in other programming languages. Julia has an exhaustive collection of all the operators required.

Arithmetic operations

Performing arithmetic operations, as discussed in the examples in previous sections, is straightforward. Julia provides a complete set of operators to work on.

Binary operators: `+`, `-`, `*`, `/`, `^`, and `%`. These are just the most used and small subset of the binary operators that are available.

```
julia> a = 10; b = 20; a + b
30
```

These are the unary plus and unary minus. The former performs the identity operation and the latter maps the values to their additive inverses.

```
julia> -4
-4
julia> -(-4)
4
```

There is a special operator `!` that can be used with bool types. It is used to perform negation:

```
julia> !(4>2)
```


False

Performing bitwise operations

These are not frequently used, except for a few. They are used to perform bitwise operations.

Expression	Name
<code>~x</code>	bitwise not
<code>x & y</code>	bitwise and
<code>x y</code>	bitwise or
<code>x ^ y</code>	bitwise xor (exclusive or)
<code>x >>> y</code>	logical shift right
<code>x >> y</code>	arithmetic shift right
<code>x << y</code>	logical/arithmetic shift left

```
julia> 100 | 200
236
julia> ~100
-101
```

Operators for comparison and updating

Julia provides operators that can be used for easy comparison and updating:

```
# updating a variable by adding a value to it
julia> x = 4; x += 10
14

# similarly, division operation can also be done
julia> x = 4; x /= 2
2.0
```

Precedence of operators

There is a specific order of priority for operators being called (also including their element-wise equivalents):

- Syntax (`.` followed by `::`)
- Exponentiation (`^`)
- Fractions (`//`)
- Multiplication (`*` , `/` , `%` , `&` , and `\`)
- Bitshifts (`<<` , `>>` , and `>>>`)
- Addition (`+` , `-` , `|` , and `$`)
- Syntax (`:` , `..` , and `|>`)
- Comparisons (`>` , `<` , `>=` , `<=` , `==` , `===` , `!=` , `!==` , and `<:`)
- Control flow (`&&` followed by `||` followed by `?`)

Type conversions (numerical)

Type conversion refers to changing the type of the variable, but keeping the value of the variable the same:

```
#creating a 8-bit Integer
julia> Int8(100)
100

#we will get an error here as we exceed the size of the Int8
julia> Int8(100*10)
ERROR: InexactError()
in Int8{::Int64} at ./sysimg.jl:53

#this is solved by using Int16
julia> Int16(100*10)
1000

#create a variable x of type Int32
julia> x = Int32(40); typeof(x)
Int32

#using type conversion, convert it to Int8
julia> x = Int8(x); typeof(x)
Int8
```

Understanding arrays, matrices, and multidimensional arrays

An array is an indexable collection of objects such as integers, floats, and Booleans, which are stored in a multidimensional grid. Arrays in Julia can contain values of the Any type. Arrays are implemented internally in Julia itself.

In most of the other languages, the indexing of arrays starts with 0. In Julia, it starts with 1:

```
# creating an array
julia> simple_array = [100,200,300,400,500]
5-element Array{Int64,1}:
100
200
300
400
500

# accessing elements in array
julia> simple_array[2]
200

julia> simple_array[2:4]
3-element Array{Int64,1}:
200
300
400
```

the types of values in an array are homogeneous:

```
# types of values in array are homogeneous
julia> another_simple_array = [100,250.20,500,672.42]
4-element Array{Float64,1}:
100.0
250.2
500.0
672.42
```

we provided two Int type values and 2 Float type values to create an array. Julia converted the Int type values to Float to create the array.

Julia tries to use promote() to promote the values of the same type. If not possible, the array will be of the type Any

```
julia> [1,2,"foobar"]
3-element Array{Any,1}:
1
2
"Foobar"
```

Creating an empty array

Although the size of the array is fixed, Julia provides certain functions that can be utilized to alter the length of the array. Let's create an empty array and add values to it:

```
# this creates an empty array
julia> empty_array = Float64[]
0-element Array{Float64,1}

# the function push!() adds values to the array
julia> push!(empty_array,1.1)
1-element Array{Float64,1}:
1.1
```

We can keep on adding values:

```
julia> push!(empty_array,2.2,3.3)
3-element Array{Float64,1}:
1.1
2.2
3.3
```

Adding values one by one may not suit the purpose. There is another way to add values to the array using the append!() function. Using append!() , we can add all the items of another collection to our collection:

```
julia> append!(empty_array,[101.1,202.2,303.3])
6-element Array{Float64,1}:
1.1
```

```
2.2
3.3
101.1
202.2
303.3
```

Working with matrices

we worked with arrays and operations on them. It is also possible to create two-dimensional arrays or matrices. There is a small difference in the syntax that helps to create these matrices:

```
# this will create an array
julia> X = [1, 1, 2, 3, 5, 8, 13, 21, 34]
9-element Array{Int64,1}:
 1
 1
...
# constructing a matrix
julia> X = [1 1 2; 3 5 8; 13 21 34]
3×3 Array{Int64,2}:

 1  1  2
 3  5  8
13 21 34
```

Different operation on matrices

```
# a 3x2 matrix
julia> A = [2 4; 8 16; 32 64]
3×2 Array{Int64,2}:
 2  4
 8 16
32 64
```

We can change the shape of a matrix using the reshape() function:

```
julia> reshape(A,2,3)
2×3 Array{Int64,2}:
 2 32 16
 8  4 64
```

This converts a 3x2 matrix to a 2x3 matrix:

```
# same could be achieved using the transpose function
julia> transpose(A)
2×3 Array{Int64,2}:
 2 32 16
 8  4 64
```

DataFrame

A DataFrame is a data structure that has labeled columns, which may individually have different data types. Like a SQL table or a spreadsheet, it has two dimensions. It can also be thought of as a list of dictionaries, but fundamentally, it is different.

DataFrames are the recommended data structure for statistical analysis. Julia provides a package called DataFrames.jl, which has all the necessary functions to work with DataFrames.

Julia's package, DataFrames, provides three data types:

- NA : A missing value in Julia is represented by a specific data type, NA .
- DataArray : The array type defined in the standard Julia library, though it has many features, doesn't provide any specific functionalities for data analysis. The DataArray type provided in DataFrames.jl provides such features (for example if we needed to store some missing values in the array).
- DataFrame : This is a two-dimensional data structure, such as spreadsheets. It is much like R or Pandas DataFrames and provides many functionalities to represent and analyze data.

DataArray – a series-like data structure

There are other features similar to Julia's Array type. Type aliases of Vector (a one dimensional Array type) and Matrix (a two-dimensional Array type) are DataVector and DataMatrix, provided by DataArray .

```
julia> using DataArrays
julia> dvector = data([10,20,30,40,50])
5-element DataArrays.DataArray{Int64,1}:
 10
 20
 30
 40
 50

julia> dmatrix = data([10 20 30; 40 50 60])
2x3 DataArrays.DataArray{Int64,2}:
10 20 30
40 50 60
julia> dmatrix[2,3]
60
```

DataFrames – tabular data structures

This is the most important and commonly used data type in statistical computing, whether it is in R (data.frame) or Python (Pandas). This is due to the fact that all the real-world data is mostly in a tabular or spreadsheet-like format. This cannot be represented by a simple DataArray .

To use DataFrames, add the DataFrames package from the registered packages of Julia:

```
julia> Pkg.update()
julia> Pkg.add("DataFrames")
julia> using DataFrame
```

Let's start by creating a simple data frame:

```
julia> df=DataFrame(Name=["Julia", "Python"],Version = [0.5, 3.6])  
2×2 DataFrames.DataFrame
```

Row	Name	Version
1	"Julia"	0.5
2	"Python"	3.6

In this module, we revisited programming paradigms to understand the best approach to the problem. In subsequent sections, we went through the concepts and how primitive data types are used and operated in Julia. After studying integers and floating point operations, we went through important and widely used data structures, arrays, and matrices. After arrays, we explored DataArray and DataFrame packages, which are more suited to representing real-world data and are widely used in statistical computing.

Julia Programming

Module 3: Function in Julia

Learning objectives

1. Create and define functions inside Julia REPL as well as independent Julia scripts.
2. Clearly, understand and differentiate between various argument passing methods and their usage.
3. Create recursive functions, along with having a good understanding of what recursion is and how it is performed in Julia.
4. Use the most commonly used built-in Julia functions in your own code.

Creating functions

Functions in Julia are declared with the function keyword, which is then followed by the body of the function. Another keyword, end, put or marks a logical end to the function in general.

```
function name()
...
body
...
End
```

Let's just have a look at how a function is defined and used inside Julia's REPL:

```
julia> function greet()
println("hello world")
end
greet(generic function with 1 method)
julia> greet()
hello world
```

Functions in Julia can also be defined using a compact form. A simple implementation could be:

```
f(x,y) = x^2 + y^2
```

An important thing to keep in mind is that although Julia's functions closely resemble mathematical functions, they aren't purely mathematical, as they can change with or be affected by the global state of the program.

```
push!(collection of items, the item you want to push to
the end of collection)
julia> push!([1, 2, 3], 4)
4-element Array{Int64,1}:
 1
 2
 3
 4
```

Function arguments

We have been discussing the syntax of functions in Julia and how to create one when required. One very important aspect when we talk about functions are arguments. Undoubtedly, we have been using them freely in almost every other language, knowing that they may be either passed by a value or a reference. But Julia is different. Julia follows a convention known as pass by sharing! Wait, now what does pass by sharing mean? For this, let's first go back to the two most commonly used conventions.

Pass by values versus pass by reference

When we say pass by value, it means that the value of whatever is passed to a function as an argument will be copied into that function, meaning that two copies of the same variable will be passed. On the other hand, when we say pass by reference, the reference or location of whatever is passed to the function is also passed into that function, meaning that only one copy of the variable will be passed.

Pass by sharing

In pass by sharing, we see that variables passed as arguments are not copied. Instead, the function arguments themselves act as new bindings, which refer to the same values as those passed. Before we move on to discuss more functions, let's spend some time talking about the return statement in Julia.

The return keyword

A return statement terminates the execution of a function and returns control to the calling function. A function in Julia may or may not use the return statement explicitly to return a value. This may be a little different for people coming from other languages, such as Python, but will become relatively easier to follow later.

In the absence of the return statement, the last expression is evaluated and returned. For instance, the underlying pieces of code are equal in terms of the value they return:

```
# function returns the value computed in the last statement

julia> function add_without_return(x,y)
x+y
end
add_without_return (generic function with 1 method)
julia> add_without_return(20,30)
50

# function with return statement

julia> function add_using_return(x,y)
return x+y
end
add_using_return (generic function with 1 method)
julia> add_using_return(20,30)
50
julia> add_without_return(20,30) == add_using_return(20,30)
True
```

Arguments

A function argument is a variable passed to a function in the form of an input so that it returns a specific output. A very simple example of a function taking an argument is stated as follows:

```
julia> function say_hello(name)
```

```
println("hello $name")
end
say_hello (generic function with 1 method)
julia> say_hello("rahul")
hello rahul
```

One thing to keep in mind is that, although Julia is dynamically typed, it supports the use of statically-typed variables. Modifying the preceding code, we have:

```
julia> function say_hello(name::String)
println("hello $name")
end
say_hello (generic function with 1 method)
julia> say_hello("rahul")
hello rahul
```

No arguments

Sometimes, we may not want to define a function with any arguments. Some functions are defined like this:

```
julia> function does_nothing
end
does_nothing (generic function with 0 methods)
```

Although this function isn't doing anything here, there will be specific use cases wherein we just want to have the function definition present in the form of an interface.

Varargs

Varargs stands for variable arguments. These come in handy when we are not sure about the total number of arguments to be passed to a function in advance. Hence, we want to have an arbitrary number of arguments.

The way we achieve this in Julia is by using three dots or Let's use an example to explain this further:

```
julia> function letsplay(x,y...)
println(x)
println(y)
end
letsplay (generic function with 1 method)
julia> letsplay("cricket","hockey","tennis")
cricket
("hockey","tennis")
```

On the flip side, the arguments to be passed to the function can be declared in advance in more than one way. For instance, let's have a function that takes in a variable number of arguments in the following fashion:

```
julia> x = (1,2,3,4,5)
(1,2,3,4,5)
julia> function numbers(a...)
println("the arguments are -> ",x)
end
numbers(generic function with 1 method)
julia> numbers(x)
the arguments are -> (1,2,3,4,5)
```

Optional arguments

During the implementation of a specific use case, you may want to have some arguments fixed (that is, having a value) or set to default. For example, you want to convert a number to another number with base 8 or perhaps base 16. The approach best suited to this would be to have a generic function that takes up a parameter base, which is then set to the number as per the requirement.

Hence, instead of having functions like `convert_to_octal()` or `convert_to_hex()`, you just have `convert_to_base(base = 8)` or `convert_to_base(base = 16)`.

example for optional arguments:

```
# function f takes 1 mandatory argument and
# 2 optional arguments

julia> function f(x, y=4, z=10)
x+y+z
end
f(generic function with 1 method)

julia> f(10)
24

julia> f(110)
124
```

Understanding scope with respect to functions

We define functions in Julia, we may also define variables inside the function body. This way, that variable is said to be inside a function's local scope, hence, is called a local variable. On the other hand, any variable that isn't declared inside a function's body is said to be in a global scope, hence, is called a global variable.

Different blocks of code can use the same name without referring to the same entity. This is defined by the scope rules.

Julia has two main types of scopes, global scope, and local scope. The local scope can be nested. Variables at the module or in REPL are generally in global scope unless stated otherwise. Variables in loops, functions, macros, try-catch-finally blocks are of local scope.

The following example is to explain local scope:

```
julia> for i=1:5
    hello = i
end

julia> hello
ERROR: UndefVarError: hello not defined
```

`hello` is only available in the scope of the for loop and not outside of it.

We can modify the previous function, to have the `hello` available outside of the loop:

```
julia> for i=1:5
    global hello
    hello = i
end

julia> hello
5
```

Julia makes use of something called **lexical scoping**, which basically means that a function's scope does not inherit from its caller's scope, but from the scope in which the function was defined!

```
julia> module Utility
    name = "Julia"
    tell_name() = name
end
Utility

julia> name = "Python"
"Python"

julia> Utility.tell_name()
"Julia"
```

Now, when we call `Utility.tell_name()`, we get the "Julia" value. This shows that this function took the value of the `name` variable, which was inside the `Utility` module, where the function `tell_me()` was declared! Hence, the other name that was declared outside the `Utility` module did not impact the result.

Julia also provides further classification of the local scope, calling it either a **soft local scope** or a **hard local scope**.

Suppose that we have a function named `alpha()`, which simply assigns a local variable named `x` to the variable passed and returns it. We have defined a global variable as `x`, which is already set to a value:

```
julia> x = 23
23

julia> function alpha(n::Int64)
           x = n
           return x
       end
alpha (generic function with 1 method)

julia> alpha(25)
25

# global x is unchanged
julia> x
23
```

But what if we wanted to use the same `x` declared globally?

```
julia> x = 23
23

julia> function alpha(n::Int64)
           global x = n
       end
alpha (generic function with 1 method)

julia> alpha(25)
25

# global x is now changed
julia> x
25
```

Nested functions

Functions defined within functions. For those coming from other language backgrounds, such as Python, the concept of closures should be very easily applicable to Julia. A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

Nesting helps in places where you want to mask the real implementation of the function from the end user.

A function with the name `outer` and another function inside it with the name `inner` :

```
julia> function outer(value_a)
           function inner(value_b)
               return value_a * value_b
           end
       end
outer(generic function with 1 method)
```

We are passing two different arguments to both functions and then utilizing them to return a value that concatenates a string (`*` is used for string concatenation in Julia) OR multiplies two integers, depending upon the data type of the arguments being passed. Also, for the time being, let's also assume that we either pass two strings or two integers and don't intermix these values because with the current function defined, it will throw an error.

```
julia> result = outer(10)
(::inner) (generic function with 1 method)

julia> typeof(result)
Function

julia> result(10)
100
```

Similarly, passing both the arguments inside the functions of the data type `String`, we have:

```
julia> result = outer("learning ")
(::inner) (generic function with 1 method)

julia> typeof(result)
Function

julia> result("Julia")
"learning Julia"
```

In both cases, we first assign the value of the function to a variable named `result` and then pass the second argument to the `result` callable. Once done, we get the desired results as expected.

Multiple dispatch

Dispatch means to send a message to a listener or a call to a function. Basically, to send a piece of data (or packet of information) to code that is ready to handle it.

Dispatch can be of many different types. Starting off with a few of them we have:

- **Static dispatch:** The dispatch order can be defined at compile time. Essentially, in static dispatch, all types are already known before the execution of the program. The compiler is able to generate specific code for every possible combination of data types and know in advance when and where they will be used. This is one of the most common in most languages. To break it down, if we have a place in the code where the function or a method is called using `funct()` or perhaps `x.funct()`, then that very same function will be invoked each and every time; there will not be any changes to it.
- **Dynamic dispatch:** The dispatch order can be defined at runtime. This simply means that the compiler has to make up a lookup table of all the functions defined and then determine which ones to actually call and which not to at runtime. In terms of code, let's say we have a few classes such as `classA` and `classB`, and both have the implementation of a function called `foo()`, then at runtime, both the classes will be examined, and finally, either of them (`classA.foo()` or `classB.foo()`) may be called.
- **Multiple dispatch:** The dispatch order is dependent on the function name as well as the argument types being passed, that is, the signature of the function and also the actual implementation that gets called is determined directly at runtime. In terms of code, let's suppose we have `classA`, which implements a method called `foo(int)` for integers and `foo(char)` for character types. Then, we have a call in the program that calls this function with `classA.foo(x)`. At runtime, we have both `classA` and `x` examined for the actual function to be called.

Recursion

Recursion is a technique in computer science that allows for a bigger problem to be broken down into smaller similar sub-problems that makes it easier to solve and debug.

we call a function recursive if it makes repeated calls to itself. This would typically involve the function having a base condition and being small enough to magnify itself easily enough to solve the overall problem.

Julia functions can also make recursive calls, just like any other language. Let's take the case of the Fibonacci series, wherein every number in the sequence is the sum of the previous two:

```
# fibonacci series
1, 1, 2, 3, 5, 8, 13, ...
```

So, if you look closely, the number 2 is the sum of its previous two digits 1+1, and then the next number, 3, is the sum of its previous two, 1+2, and so on. This problem makes a genuine case of recursion.

```
julia> function generate_fibonacci(n::Int64)
    if n < 2
        return 1
    else
        return
            generate_fibonacci(n-1)+generate_fibonacci(n-2)
    end
end
generate_fibonacci(generic function with 1 method)
```

Built-in functions

Julia provides a number of built-in functions, which are very helpful once you fully understand the richness of the Julia base library. Like every other language, Julia has functions for most common tasks performed by users, as well as some surprises as we go through this topic.

We will now walk through some of the most common built-in functions one by one, along with detailed examples:

- **workspace()** : This is a function specifically for Julia REPL and isn't available outside of it. The work of this function is actually to clear out the current workspace in the Julia REPL, deleting all the functions, variables, constants, or types defined by the user without needing to exit the REPL and restart it once again.
- **typeof()** : This function is used mainly to know the data type of an argument passed to it. This is similar to the `type()` function for those familiar with Python:
- **methods()** : This function is very useful and is frequently used when the user wants to know the methods available corresponding to a user-defined function. In other words, when you use multiple dispatches and want to inquire about the implementations we have for that particular function, then we use this method:

```
julia> methods(+)
# 163 methods for generic function "+":
+(x::Bool, z::Complex{Bool}) at complex.jl:136
+(x::Bool, y::Bool) at bool.jl:48
+(x::Bool) at bool.jl:45
+{T<:AbstractFloat}(x::Bool, y::T) at bool.jl:55
+(x::Bool, z::Complex) at complex.jl:143
+(x::Bool, A::AbstractArray{Bool, N<:Any}) at
arraymath.jl:126
+(x::Float32, y::Float32) at float.jl:239
+(x::Float64, y::Float64) at float.jl:240
+(z::Complex{Bool}, x::Bool) at complex.jl:137
...
```


- **readline()** and **readlines()**: This function is used to take in input from the user. There are many ways to use this function. Like, for example, if we want to ask the user to enter his/her name in the Julia REPL, then we may use this as follows:

```
julia> name = readline()
"Julia"
"\nJulia\n"

julia> println(name)
"Julia"
```

- **enumerate()**: This is one of the most commonly known functions for people coming from a different language background. The `enumerate()` function must be used when we need to iterate over a collection of items and at the same time keep track of the index position of that particular item. One simple example is shown as follows:

```
julia> fruits
4-element Array{String,1}:
"apples"
"oranges"
"bananas"
"Watermelon"

julia> for (index,fruit) in enumerate(fruits)
    println("$index -> $fruit")
end
1 -> apples
2 -> oranges
3 -> bananas
4 -> watermelon
```

- **parse()**: This function is basically used to parse down a given string and return an expression. It can basically infer the data type of the argument passed as a string to itself. To understand what is meant by this, just have a look at the following example:

```
julia> parse("2")
2

julia> parse("2.22")
2.22

julia> parse("Julia") :Julia
```

Julia Programming

Module 4: Understanding Types and Dispatch

Learning Objectives

1. Understand the various types in Julia and be able to supply them.
2. Create new data types using the existing ones.
3. Clearly, understand the difference between a module and an interface.
4. Have a solid understanding of what multiple dispatch is, and how it is used in Julia.

Types, often referred to as **data types**, are simply a classification of data that lets the computer know the differences between the kinds of input being provided by the user. Julia also uses a type system that uniquely identifies integers, strings, floats, Booleans, and other data types.

Julia's type system

What are types?

To answer this question, consider the following four lines:

```
1
1.10
'j'
"Julia"
```

Starting from the first line: we have 1, which is an integer; 1.10, which is a float (or decimal); 'j', which represents a single character; and lastly, "Julia", which is a simple string made from a collection of characters used together.

But, even though we have prior knowledge about the data types in use, how do we let the machine know the same? How will the computer know that 1 is an integer, and not a float or a string? Well, the answer to this question is types!

Statically-typed versus dynamically-typed languages

Statically-typed languages, such as C, C++, or Java, wherein we need to explicitly define the type of the data beforehand. This lets the compiler know about the incoming data before the program is actually executed. **Dynamically-typed languages** such as Perl, Python, and Ruby, in which the user need not declare the type of data beforehand and the interpreter will automatically infer the type of data at runtime.

So, is Julia a dynamically-typed or statically-typed language?

The answer to this question is both (yes!). However, it inclines towards being a dynamically-typed language, because of its ability to infer the type of data at runtime. Having said that, it doesn't mean that Julia does not have a rich type system. You will be surprised to know that a Julia code that has types already declared is a lot faster in terms of execution speed!

Type annotations

In the previous module, we read about functions in Julia and how to statically declare the data type of the argument in a function definition. In this section, we will be focusing on type **declarations** and **conversions**.

Let's have a look at the following example, where we declare a simple mathematical function to find the cube of a number:

```
# declare the function
julia> function cube(number::Int64)
```

```

        return number ^ 3
    end
cube (generic function with 1 method)

# function call
julia> cube(10)
1000

```

If you follow along closely, you will notice the use of an operator, `::`, along with `Int64` being used while declaring this function `cube`. The `::` is nothing but a simple operator available in Julia that lets you attach type annotations to an expression or a variable in a program. The `Int64` is a type that is used to denote that the argument number is of the Integer data type.

More on types

Abstract data types are the ones that cannot be instantiated. They serve as the basic pillars of the type system in Julia. Or, in other words, other types inside Julia can inherit any one of these base types. Examples of abstract data types are `Number`, `Integer`, and `Signed`.

Julia supports all the basic data types, along with the ability to add in new user-defined data types as well as composite types.

Before we begin talking about the different data types in detail, we want to share three simple functions that we will be used to know more about types:

- `typeof()`: This is used to tell the type of data being supplied to it
- `typemax()`: This is used to know the maximum value supported by the specific type
- `typemin()`: This is used to know the minimum value supported by the specific type

The Integer type

The `Integer` type is used to tell the Julia LLVM compiler about an incoming integer-type object. It's called `Int8`, `Int16`, `Int32`, `Int64`, or `Int128`, depending upon the machine you are using. For instance, if you are working on a 64-bit machine, the `Integer` type would be `Int64`:

```

# Knowing the type of data type passed
julia> typeof(16)
Int64

# Highest value represented by the Int64 type
julia> typemax(Int64)
9223372036854775807

# Lowest value represented by the Int64 type
julia> typemin(Int64)
-9223372036854775808

```

We can also have unsigned integers, and the way Julia represents them is by using the types `UInt8`, `UInt16`, `UInt32`, `UInt64`, and `UInt128`.

The Float type

This one is used to denote the `Float` type; in other words, integers with decimals. We use the `Float64` type here.

```
julia> typeof(1.10)
Float64

julia> typemax(Float64)
Inf

julia> typemin(Float64)
-Inf
```

Here, `Inf` denotes infinity, which is Julia's unique way to denote the value infinity.

The Char type

The `Char` type is used to denote a single character:

```
julia> typeof('c')
Char
```

The String type

The `String` type is used to denote string data, which is a collection of characters. Notice that we have double quotes around a `String` type, unlike the single quotes around character type data:

```
julia> typeof("Julia")
String
```

The Bool type

The `Bool` type is used to denote the values of the Boolean type `true` and `false`:

```
julia> typeof(true)
Bool
```

A quick way to find out the data type we are working with is the `isa()` function. On checking the original documentation, we see the following:

```
isa(x, type) -> Bool

Determine whether x is of the given type.
```

This is almost the same functionality provided by the `isinstance()` function in Python, which also checks for data type and returns a Boolean value. Practically, it's as simple as the following command:

```
julia> isa(5, Int64)
```

```
true

julia> isa(5.5, Float64)
true

julia> isa(5.5, UInt64)
false
```

Type conversions

So far in this module, we have been discussing various data types in Julia, mostly which specific type they are and how we declare them. Now, what if we wanted to have a value of a specific data type converted to another data type?

For instance, we have a function with the name `distance_covered(speed::Int64, time_taken::Int64)`, which returns the distance covered by a moving vehicle's speed and time is taken. Now, as we know, distance is calculated as the product of speed with the time taken. Or, in other words, we have the function definition like this:

```
julia> function
distance_covered(speed::Int64,time_taken::Int64)
return speed * time_taken
end
distance_covered(generic function with 1 method)
```

Now, whatever the result of this product, we only want to have an integer output. But there is a catch! The product of an integer and an integer is always an integer. This means that the result of the `distance_covered()` function is always going to be an integer. Hence, to have a float value, we need to use type conversion here, so that we can get a float solution:

```
# Distance covered by vehicle having a speed of 64 kmph and traveling
for 2 hours.
julia> distance_covered(64, 2)
128

# testifying that the result was a integer!
julia> typeof(ans)
Int64
```

To help us solve this issue, Julia has provided us with a function by the name of `convert()`. Before we learn more about the `convert()` function, let's first use it to get our desired result from the `distance_covered()` function in the float.

The function `convert()` takes two arguments. The first argument is the data type, which you want to have in the final result, or to which you want to convert the existing value; while the second argument is the value you want to convert.

```
julia> convert(Float64, 128)
```

```
128.0
```

```
julia> typeof(128.0)
Float64
```

So now, we get the result of the function `distance` in `Float64`, which is what we wanted.

In the preceding function, you will see we converted `128` to `128.0` (that is, from `Int64` to `Float64`). But what if we wanted to convert a float value to an integer value? Take a look at the following code:

```
# easily done
julia> convert{Int64}(128.0)
128

# breaks! but why???
julia> convert{Int64}(128.5)
ERROR: InexactError{Int64}
  in convert{Int64}(::Type{Int64}, ::Float64) at ./int.jl:239
  in convert{Int64}(::Type{Int64}, ::Float64) at
/Applications/Julia-0.5.app/Contents/Resources/julia/lib/julia/
sys.dylib:?
```

The first time, it's able to convert `128.0` to `128` easily, but what happens the second time? The function fails to convert a float with a non-zero value after the decimal! The error called `InexactError{Int64}` was raised due to Julia's inability to convert the value into a float.

As a technical language, Julia works hard to maintain accuracy in numerical computations. Since floating point numbers cannot be accurately represented as an integer, Julia raises an error.

How can we *force* Julia to do the conversion? We can give Julia specific instructions for converting from `Float` to `Int` via rounding.

```
julia> ceil{Int}(3.4)
4

julia> floor{Int}(3.4)
3

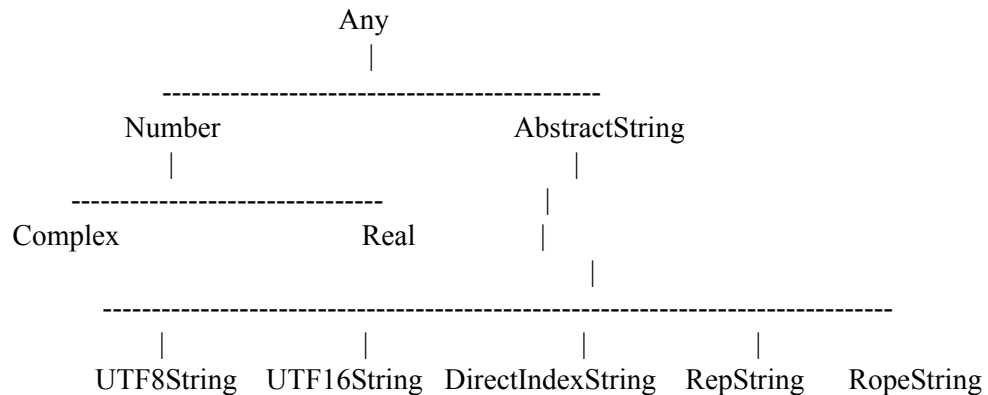
julia> round{Int}(3.4)
3
```

The subtypes and supertypes

Julia's type system is organized into a clean hierarchy of data types. Some of the data types sit above other data types, and vice versa. This, however, should not be confused with the precedence of types, as we are

not talking about that here. Rather, our focus is mainly to understand how Julia's type system organized itself into a tree structure.

To start with, the starting point of all data types in Julia is the `Any` datatype. The `Any` type is like the parent node of the tree, and all the other possible data types are directly or indirectly its child nodes. Following is Julia's type hierarchy (sample):



Looking at the tree structure clearly, in the simplest of terms you may get to understand that the `Number` type is a subtype of the `Any` type, which is acting as a child of the parent type `Any`. So what do we call the parent type with respect to the `Number` type? It is called a **Supertype**. If that was a bit difficult to follow, let's introduce two functions that are meant to demonstrate and showcase the exact explanation.

The `supertype()` function

This function is used to return the `supertype` function of a type passed as its argument. Opening the Julia REPL, we can now check for the supertype of the `Number` type:

```
julia> supertype(Number)
Any
```

What about the supertype of the type `Any`? Let's try that out too!

```
julia> supertype(Any)
Any

julia> typeof(Any)
DataType
```

`Any` is the starting point of all the other data types.

The `subtypes()` function

This function is used to return the `subtypes` function of a type passed as its argument. Let's check the subtypes of the `Number` type:

```
julia> subtypes(Number)
2-element Array{Any,1}:

```



```
Complex{T<:Real}  
Real
```

This is exactly what we were discussing in Julia's type hierarchy. An important relation to keep in mind is that now, for the types `Complex` and `Real`, the `Number` type will be seen as a supertype function. So, it depends upon the type hierarchy whether a specific data type becomes a subtype function and/or a supertype function.

User-defined and composite data types

Until now, we have been dealing with the data types that were available to a user and provided by Julia. Now we will be exploring ways to use types that are not made available to us by Julia, and we need to create them in order to address the problems we are dealing or intend to deal with.

The most important keyword that we need to create a user-defined data type is called a `type`. The following is an example of how to create a simple data type in Julia:

```
julia> type Person  
        name::String  
        age::Int64  
    end  
  
julia> rahul = Person("rahul",27)  
Person("rahul",27)  
  
julia> typeof(rahul)  
Person  
  
julia> rahul.name  
"Rahul"  
  
julia> rahul.age  
27
```

This example is one of the simplest examples of what we can consider a user-defined type in Julia. Observing closely, the `Person` type has two fields, namely `name`, and `age`. The fields can easily be accessed by using a period (.) notation.

Interestingly, if you try to get the supertype or subtype function of this `Person` type, this is what you get:

```
julia> supertype(Person)  
Any  
  
julia> subtypes(Person)  
0-element Array{Any,1}
```

This shows that the `Person` type is placed directly below the `Any` type in the tree hierarchy of Julia's type system. At this point in time, it will also be interesting to introduce a new function called `fieldnames()`, which is used to list down all the field names for any defined type:

```
julia> fieldnames(Person)
2-element Array{Symbol,1}:
 :name
 :age
```

The function `fieldnames(T)` takes in the datatype `T` and outputs all the field names declared in that data type. Fieldnames are of the `Symbol` type, which is yet again a different data type of the `DataType` type:

```
julia> typeof(:name)
Symbol

julia> typeof(Symbol)
DataType
```

Unlike traditional object-oriented languages like Java, Python, or C++, Julia doesn't have classes! Yes, that's correct. Julia separates the types along with their methods and prefers to use multiple dispatch instead.

Composite types

A **composite type** is a collection of named fields, which can be treated as a single value. We can quickly define a composite type as:

```
julia> type Points
    x::Int64
    y::Int64
    z::Int64
end
```

We have already covered the usage in the previous section. By mutable, we mean that the `fieldnames` type, once assigned, can be changed and reassigned to some new value, other than the one used while creating the object. This can be understood by the example given as follows:

```
julia> struct Point
        x::Int
        y::Int
        z::Int
    end

julia> p = Point(1,2,3)
Point{1, 2, 3}

julia> p.x = 10
ERROR: type Point is immutable
```

We see that we were not able to change the values of `x`, `y`, and `z` once they were assigned at the time of object creation. The `ERROR: type Point is immutable` error sums up the issue in simpler words.

We need to use the keyword `mutable` to make the type mutable.

```
julia> mutable struct MutPoint
    x::Int
    y::Int
    z::Int
end

julia> p = MutPoint(1,2,3)
MutPoint{Int64}(1, 2, 3)

julia> p.x = 10
10

julia> p
MutPoint{Int64}(10, 2, 3)
```

Hence, we are able to assign the values of `x`, `y`, and `z`, which are the fields to different values than those declared or assigned at the time of creation of the object sample.

Inner constructors

Till now, we have seen how to use Julia's predefined types as well as user-defined composite types. Now, we are going to delve a little deeper into learning what happens when a user-defined type gets created, and the applications of making this feature more useful to the end user.

As we have already discussed, when we create a new type, we may or may not include field names in the body. Functions, on the other hand, remain distant, and unlike other object-oriented languages, the methods aren't present.

So when we create a new type object, the default constructor comes into action:

```
julia> type Family
    num_members :: Int64
    members :: Array{String, 1}
end

julia> f1 = Family(2, ["husband", "wife"])
Family{Int64, Array{String,1}}(2, String["husband", "wife"])
```

We have declared a composite type, `Family`, and it has two fields, `num_members` and `members`. The first field name is for declaring the number of family members, and the second field name explicitly calls out all family members in an array.

But what if we wanted to validate the fields of the Family type? One way is to create another function that can validate the same, and apply it over the object being created for the same type:

```
julia> f2 = Family(1, ["husband", "wife"])
Family{1,String["husband","wife"]}

julia> function validate(obj :: Family)
    if obj.num_members != length(obj.members)
        println("ERROR! Not all members listed!! ")
    end
end

validate (generic function with 1 method)

julia> validate(f2)
ERROR! Not all members listed!!
```

So, as you can see, we created a function called validate, which will take an object of the Family type and then validate if the number of members mentioned is exactly equal to the members listed in the array!

Alternatively, we can do the same thing with a much simpler method, by declaring inner constructors. They can be used to validate the fields even before the object gets created, and hence, can provide better control over the object creation process, along with saving us useful time.

We will create the object using new, which like in Java and other languages, is for the purpose of creating an object for a given specific type.

Here is what we could have done instead:

```
julia> type Family
    num_members :: Int64
    members :: Array{String, 1}
    Family(num_members, members) = num_members != length(members)
    ?
    error("Not equal") : new(num_members, members)
end

julia> f1 = Family(1, ["husband", "wife"])
```

On running the preceding line of code, we get the result in which the first line of the error prints out what we wanted:

```
ERROR: LoadError: Not equal
in Family{::Int64, ::Array{String,1}} at ./sample.jl:4
in include_from_node1{::String} at ./loading.jl:488
in
include_from_node1{::String} at ./julia/lib/julia/sys.dylib:?
```

```
in process_options(::Base.JLOptions) at ./client.jl:265
in _start() at ./client.jl:321
in _start() at ../julia/lib/julia/sys.dylib:?
while loading ../sample.jl, in expression, starting on line 7
```

Declaring internal constructors can be time-saving, and also make the code more efficient by stopping the creation of unwanted objects if they don't fulfill a certain criterion.

Consider the following example:

```
julia> type A
    x
    y
end

julia> type B
    x
    y
    B(x, y) = new(x, y)
end

julia> a = A(1,1)
A(1,1)

julia> b = B(1,1)
B(1,1)
```

Interestingly, both the forms are equal, although we have declared the inner constructor form in `type B`. Julia automatically generates constructors for types. Users can declare additional constructors or override the default one using inner constructors.

Note: Use of inner constructors is discouraged. From the manual: "It is considered good form to provide as few inner constructor methods as possible: only those taking all arguments explicitly and enforcing essential error checking and transformation."

Modules and interfaces

Like many other languages, Julia also has a way to group similar logical sets of code together into workspaces or also called as namespaces. They help us in creating top-level definitions - that is, global variables, without taking on the risk of code conflicts, as the names and variables used inside a module remain unique.

In Julia, we create a module as follows:

```
module SampleModule
```

```
..
..
end
```

Modules help us to indicate code that can be imported into other parts of the program, as well as the set of code meant to be used (or visible) to the world outside.

Here is one small example of a functional module:

```
# marks the start of the module named Utilities
julia> module Utilities
    # marks the type, variable or functions to be made
    available
    export Stype, volume_of_cube
    type Stype
        name::Int64
    end
    function area_of_square(number)
        return number ^ 2
    end
    function volume_of_cube(number)
        return area_of_square(number) * number
    end
    # marks the end of the module
end
```

Here, we have a module named Utilities, which has two functions named volume_of_cube and area_of_square, along with a user-defined type called Stype.

Out of these three, we have volume_of_cube and Stype exported, which are made available to the world outside. The area_of_square function is kept private.

Once a module is made, it is ready to be used by other parts of the program. To do so, Julia provides a number of reserved keywords that facilitate usage. The two most used of them are using and import.

We will be focusing on the usage of these two, using a very simple example:

```
julia> module MyModule
    foo = 10
    bar = 20
    function baz()
        return "baz"
    end
    function qux()
        return "qux"
    end
end
```

```
        end export foo, baz
    end
MyModule
```

We will attempt to use foo by importing MyModule.

```
# No public variables are brought into scope
julia> import MyModule

julia> foo
ERROR: UndefVarError: foo not defined

julia> baz()
ERROR: UndefVarError: baz not defined
```

We will attempt to do it now with using.

```
julia> using MyModule

julia> foo
10

julia> bar
ERROR: UndefVarError: bar not defined

julia> MyModule.bar
20

julia> baz()
"baz"
```

When we run the command:

- **using** MyModule, then all the exported functions, as well as other functions, are brought into the scope. We could also have chosen to individually call the functions declared in the MyModule module using `MyModule.foo`. A second form of the same can be to use `MyModule.foo`. In both these cases, we only get what we call - that is, only those functions enter the current scope that has been called explicitly.
- **import** MyModule, all functions from the MyModule module will populate the current scope. There is another keyword called `importall`, which is slightly different from `import` in a way that it just gets the functions or variables exported by the module.

However, the biggest difference between using and import is that, with using, none of the functions from the module are available for method extension, while with import, there is flexibility for method extension.

Including files in modules

We have seen how we can create a module using the `module` keyword. However, what if our package, or module, is spread up into different files? The way Julia addresses this issue is by introducing `include`, which can be used to include any Julia file (with the `.jl` extension) in a module.

Here is an example that will make things clear. Suppose that we have a module with the `PointTypes` name, and our code for it is written in the `transformations.jl`. It contains:

```
function move(p::Point, x, y)
    slidex(p, x)
    slidey(p, y)
end

function slidex(p::Point, dist)
    p.x += dist
end

function slidey(p::Point, dist)
    p.y += dist
end
```

Once we are done with module creation, the next step would be to test the same. This can be done as follows:

```
julia> module PointTypes
    mutable struct Point
        x::Int
        y::Int
    end

    # defines point transformations
    include("transformations.jl")

    export Point, move

end
PointTypes

julia> using PointTypes
julia> p = Point(0,0)
PointTypes.Point{Int64}(0,0)

julia> move(p, 1, -2)
-2

julia> p
```



```
PointTypes.Point(1, -2)

julia> slindex(p, 1)
ERROR: UndefVarError: slindex not defined
```

Multiple dispatch explained

To have a quick look at what multiple dispatch means in functions, here is the same code for a function that prints out the cube of a number:

```
julia> function calculate_cube(num::Int64)
    return num ^ 3
end
calculate_cube (generic function with 1 method)

julia> function calculate_cube(num::Float64)
    return num ^ 3
end
calculate_cube (generic function with 2 methods)

# 2 methods for generic function "calculate_cube":
calculate_cube(num::Float64) at REPL[1]:2
calculate_cube(num::Int64) at REPL[1]:2

julia> calculate_cube(10)
1000

julia> calculate_cube(10.10)
1030.301
```

Here, the function supported both kinds of concrete types (which are Int64 and Float64) values. One gave an integer output and the other gave Float type output.

But how can the same technique be applied to user types? For that, we first have to equip ourselves with what we call parametric types. Let us first see an example:

```
julia> type Coordinate{T}
    x::T
    y::T
    z::T
end
```

Here, we have a user-defined type named Coordinate, which represents a point in a 3D space that has x, y, and z axes. On closely watching the syntax, we see {T}, which is enclosed within curly braces. Here, T acts as an arbitrary type, which can be passed on this Coordinate user-defined type as a Parameter.

The body of the `Coordinate` type has the values `x`, `y`, and `z`, which are all annotated using `::` to the same type `T`. Moving ahead, we now create an object of this new type - let's see how it works:

```
# when T is of Int64 type
julia> point = Coordinate{Int64}(1,2,3)
Coordinate{Int64}(1,2,3)

julia> point.x
1

julia> point.y
2

julia> point.z
3

# when T is of Float64 type
julia> point = Coordinate{Float64}(1.0,2.0,3.0)
Coordinate{Float64}(1.0,2.0,3.0)

julia> point.x
1.0

julia> point.y
2.0

julia> point.z
3.0
```

Here, the variable `point` holds values for an `Int64` type, and then holds the `Float64` type values.

Let's get back to multiple dispatch. After clearly understanding how parametric composite types are made or instantiated, let's implement multiple dispatch using parametric types:

```
# Creating a parametric type
julia> type Coordinate{T}
    x::T
    y::T
    z::T
end

# Method that works on Int64
julia> function calc_sum(value::Coordinate{Int64})
    value.x + value.y + value.z
end
calc_sum (generic function with 1 method)
```

```
# Method that works on Float64
julia> function calc_sum(value::Coordinate{Float64})
    value.x + value.y +value.z
end
calc_sum (generic function with 1 method)

# Multiple Dispatch
julia> methods(calc_sum)
# 2 methods for generic function "calc_sum":
calc_sum(value::Coordinate{Int64}) at REPL[61]:2
calc_sum(value::Coordinate{Float64}) at REPL[60]:2

# Calling the method on Int64
julia> calc_sum(Coordinate(1,2,3))
6
julia> typeof(ans)
Int64

# Calling the method on Float64
julia> calc_sum(Coordinate(1.0,2.0,3.0))
6.0
julia> typeof(ans)
Float64
```

Julia Programming

Module 5 - Working with Control Flow

Learning Objectives

- How to structure Julia programs using various control flow techniques to provide efficient execution of the code.
- Conditional and repeated evaluation.
- Exception handling.

Conditional and repeated evaluation

Conditional evaluation helps break the code into smaller chunks, with each chunk being evaluated based on a certain condition. There are many ways in which such conditions can be imposed and the code can be handled accordingly.

Compound expressions are ways in which we can make sure that a sequence of code gets evaluated in an orderly fashion. Let's have a look at some code:

```
julia> volume = begin
    len = 10
    breadth = 20
    height = 30
    len * breadth * height
end
6000

julia> volume
6000
```

The `begin` keyword, however, is not necessary to include, given that you use an alternate method to declare compound expressions using `;"` key.

Correct method of using `;"` key

```
julia> volume = (length = 10; breadth = 20; height = 30; length *
*breadth *height)
```

```
julia> volume
6000
```

Wrong method of using `;"` key

```
julia> volume = length = 10; breadth = 20; height = 30; length *
breadth *height
```

```
julia> volume
10
```

While writing any code, we are often confronted with situations where we need to make some decisions based on a set of rules or predefined business logic. In such cases, we would generally expect the code to be intelligent enough that it can take the right path accordingly. Here is when conditional evaluation comes into the picture.

Here, we will be implementing the famous FizzBuzz program which states that, for any given range of numbers (let's say 1 to 30), print Fuzz instead of the number for multiples of 3 and Buzz for multiples of 5. For numbers which are both divisible by 3 and 5, the program should print FizzBuzz.

```
julia> for i in 1:30
    if i % 3 == 0 && i % 5 == 0
        println("FizzBuzz")
    elseif i % 3 == 0
        println("Fizz")
    elseif i % 5 == 0
        println("Buzz")
    else
        println(i)
    end
end
```

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
```

So, as expected, the `FizzBuzz` string got printed twice in positions 15 and 30, the `Fizz` string got printed eight times, and the `Buzz` string got printed four times.

The code also contains some unknown keywords such as `for` and `in`. However, the keywords `if`, `elseif`, and `else` makeup what we collectively call **conditionals**.

This structure is present in almost every programming language of the modern era, with just the slight difference of using the `end` as the logical ending of any functional conditional block of code.

The keywords `if`, `else`, and `elseif` basically evaluates the `elseif` expressions in Boolean terms. So, given any condition, `if` will try to check whether that expression evaluates to `true` or `false`. If the condition comes out to be Boolean `true`, then the code in `if` block gets evaluated; otherwise, we have an `else` or an `elseif` block.

Short-circuit evaluation

In Julia, apart from the regular Boolean operators, such as `true` and `false`, we also have `&&` and `||`.

- Looking at the `var1 && var2` expression, `var2` only gets executed if `var1` evaluates to `true`
- Looking at the `var1 || var2` expression, `var2` only gets evaluated if `var1` evaluates to `false`

To get a better understanding of the two, let's have a look at some code:

```
julia> isdigit("10") && println("Reached me!")
Reached me!

julia> isdigit("Ten") && println("Reached me!")
false

julia> isdigit("Ten") || println("Reached me!")
Reached me!

julia> isdigit("Ten") && println("Reached me!")
False
```

In the first line of code, the `isdigit("10")` function takes in a string value and checks whether it's a digit or not. In this case, as the value is `true`, the second expression, `println("Reached me!")`, got successfully executed.

On the other hand, in the third line of code, the `isdigit("Ten")` expression evaluates to `false`, and hence the `println("Reached me!")` expression gets executed.

Repeated evaluation

In Julia, we are provided with two constructs, namely `while` and `for`. These two are used for evaluating the code in loops. The following is an example, for both `while` and `for` loops:

```

julia> collection = [1,2,3,4,5,6,7,8,9]
9-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
 7
 8
 9

julia> while length(collection) > 1
        pop!(collection)
        println(collection)
    end
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7]
[1,2,3,4,5,6]
[1,2,3,4,5]
[1,2,3,4]
[1,2,3]
[1,2]
[1]

```

In this example, we have an array of integers from 1 to 9. We named it a `collection`, and then we used a `while` construct to loop over the array of items until the length of the `collection` was greater than 1. However, the code is written using `while` loops can easily become an infinite loop if our `end` condition is not correctly mentioned. For instance, in the preceding code, we have used a function provided by Julia called `pop!`, which takes in a collection of items and removes (or pops) the last item from it, this way modifying the original collection to be shorter and shorter.

Had we not controlled the end point of the `while` loop, this is what it would have resulted in:

```

julia> collection = [1,2,3,4,5,6,7,8,9]
9-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
 7
 8

```


9

```
julia> while length(collection) > 1
           println(collection)
       end
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
[1,2,3,4,5,6,7,8]
.....
.
.
```

Another construct called the **for** construct. It works in the following fashion:

```
julia> statement = "This is a great example!"
"This is a great example!"

julia> for words in split(statement)
           println(words)
       end
This
is
a
great
example!
```

While coding, we are often confronted with code that has something to do with the range of items to loop upon. For instance, how does someone know whether any number between 1 and 10 is either even or odd?

```
julia> numbers = [1,2,3,4,5,6,7,8,9,10]
10-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
 7
 8
```

```

9
10

julia> for n in numbers
if rem(n,2) == 0
println("$n is even")
end
end
2 is even
4 is even
6 is even
8 is even
10 is even

```

The break and continue

The break block is used in places when we want the control to be shifted to an outer block of code. This means that we do not want to go further into the evaluation, and want to immediately break and come out from that loop cycle.

```

julia> word = "julia"
"julia"
julia> for letter in word
println(letter)
if letter == 'l'
break
end
end
j
u
l

```

The continue statement is used in places where we want the evaluation to go on if the given condition is met. Here is the code, modified to showcase the operation of a continuous block:

```

julia> for letter in "julia"
if letter == 'l'
continue
end
println(letter)
end
j
u
i
a

```

Exception handling

It is important to establish proper exception or error handling. This can be ensured by using Julia's built-in exception handling methods, which we will be discussing in this portion.

Julia provides many types of errors:

- `ArgumentError`: The argument passed to a function does not resemble the argument the function was expecting originally.
- `AssertionError`: This is the result when there is a wrong assertion statement written and it evaluates to false.
- `BoundsError`: Trying to access an out-of-bounds element while indexing an array.
- `DivideError`: This is the result when a number is divided by 0.
- `DomainError` : Arguments outside a valid domain.
- `EOFError`: This means that you have reached the end of the file, and hence there is nothing more to be read from it.
- `InexactError`: Failure in trying to do exact type conversion.
- `KeyError`: An attempt was made to access an element that isn't a key element - that is, non-existent.
- `LoadError`: An error occurred while loading a file.
- `MethodError`: There was an error while trying to use a signature of a function that isn't supported. In other words, the user tried to use a method that wasn't listed in the list of `methods(function)`.
- `OutOfMemoryError`: The computation requires more memory than allocated.
- `ReadOnlyMemoryError`: Trying to write in read-only memory.
- `OverflowError`: This happens when the result of an operation is too large.
- `ParseError`: Problem in parsing the given expression.
- `StackOverflowError`: This happens when a function call goes beyond the call stack.
- `SystemError`: An error due to the failure of a system call.
- `TypeError`: An assertion failed for type checking. This happens most commonly when the wrong type of argument is passed to a function.
- `UndefRefError`: Unknown reference made.
- `UndefVarError`: Unknown reference made to a variable that doesn't exist.
- `InitError`: This error occurs when there is a problem calling the module's `__init__` method.

There are many types of exception as well:

- `ErrorException`: An error occurred
- `InterruptException`: An external interruption occurred in the computation
- `NullException`: Trying to access a Null value
- `ProcessExitedException`: The process ended, hence further attempts to access this process will result in errors

The throw() function

The `throw()` function does not throw an exception as soon as it encounters one. It immediately throws an error when called. Looking over the official documentation, this is what it says:

```
throw(e)
    Throw an object as an exception.
```

Here, `e` is an object. `throw` will convert it into an exception.

```
julia> throw("")
ERROR: ""

julia> throw(3+3)
ERROR: 6

julia> throw("some error message")
ERROR: "some error message"

julia> throw(ErrorException("an Exception has occurred"))
ERROR: an Exception has occurred
```

To correctly throw a `TypeError`, you must actually throw an instance of the object. An instance can be created using type `TypeError` constructor :

```
julia> function say_hi(name)
    if typeof(name) != String
        throw(TypeError)
    else
        println("hi $name")
    end
end
say_hi (generic function with 1 method)

julia> say_hi('n')
ERROR: TypeError
Stacktrace:
 [1] say_hi(::Char) at ./REPL[155]:3

julia> try
    say_hi('n')
catch e
    println(typeof(e))
end
DataType
```

Notice that the `typeof(e)` call returns `DataType`. To correct this, create an instance of a `TypeError`.

```
julia> function say_hi(name)
    if typeof(name) != String
        throw(TypeError(:say_hi, "printing name", String, name))
    else
        println("hi $name")
    end
end
say_hi (generic function with 1 method)

julia> say_hi('n')
ERROR: TypeError: say_hi: in printing name, expected String,
got Char
Stacktrace:
 [1] say_hi(::Char) at ./REPL[158]:3

julia> try
    say_hi('n')
catch e
    println(typeof(e))
end
TypeError
```

Here, we have a function named `say_hi`, which takes in a name and prints out a message, `hi $name`, where the name is to be provided externally. When we provide `"jack"` as the name, the function works; but then, if the name provided is not of the `String` type, then it's coded to throw a `TypeError`, which it does when `1` is passed as the argument.

Although the example is just meant to provide an understanding of how `throw` works, the preceding situation could have easily been avoided without using `throw` at all, if we were to use multiple dispatch as shown in the following code:

```
julia> workspace()

julia> function say_hi(name :: String)
    if typeof(name) != String
        throw(TypeError)
    else
        println("hi $name")
    end
end
say_hi (generic function with 1 method)
# MethodError will be thrown instead of TypeError
```

```
julia> say_hi(1)
ERROR: MethodError: no method matching say_hi(::Int64)
Closest candidates are:
say_hi(::String) at REPL[15]:2
```

So a more useful example would be to check whether the length of the given name matches a specific number. Here is the modified code:

```
julia> workspace()

julia> function say_hi(name :: String)
    if length(name) < 5
        throw(ArgumentError)
    else
        println("hi $name")
    end
end

say_hi (generic function with 2 methods)

julia> say_hi("joe")
ERROR: ArgumentError
in say_hi(::String) at ./REPL[21]:3

julia> say_hi("david")
hi david
```

The error() function

This function can be used to raise an `ErrorException` of a custom type, which the developer wants to show the end user.

It should be noted that the use of this function is generally discouraged outside of debugging and logging. It is almost always better to use a more specific or custom exception.

```
julia> function say_hi(name :: String)
    if length(name) < 5
        error("Name less than 5!!!")
    else
        println("hi $name")
    end
end

say_hi (generic function with 1 method)

julia> say_hi("joe")
ERROR: Name less than 5!!!
```

```
in say_hi(::String) at ./REPL[33]:3
```

Here, we have used a custom message of our own, `Name less than 5!!!`, which gets printed on the screen if the length is less than 5.

We also have two more slightly less-used functions compared to `throw()` and `error()`, namely `info()` and `warn()`. These two basically resemble log levels, when we want to tell the end user about the level of problems in the code:

```
julia> info("I am information to the end user")
INFO: I am information to the end user

julia> warn(" I am a warning issue to the end user")
WARNING: I am a warning issue to the end user
```

The try/catch/finally blocks

The most important part of exception handling is to take care of using these statements. They not only provide effective safety from unknown errors but also increase control over the code from the developer's perspective. In other words, all well-written code, if possible, must include `try/catch` blocks:

```
julia> try
    exponent("alpha")
catch(e)
    println(e)
end
MethodError(exponent, ("alpha",), 0x000000000000055aa)
```

We already know that `exponent` doesn't work over `String`-type values; however, in this example, we tried to evaluate it firsthand inside the `try` block, and failing that, `catch` will catch and hold the exception as `e`.

We can see that there was a `MethodError` exception raised by the code, which was then later on reported by the `catch` block.

The `finally` block is used when we need to do a cleanup once the code execution is done. You can think of this in terms of closing a database connection once the entry in the database table has been made. The syntax for that would be:

```
try
...
...
finally
    db_connection.close()
End
```

Julia Programming

Module 6: Interoperability and Metaprogramming

Learning Objectives

1. How Julia interacts with the outside world, by using different ways of making system calls to the operating system (OS).
2. Using the code from other languages such as C and Python.
3. The aspect of Julia in the form of metaprogramming.
4. Types of macros provided by default in Julia, and how to create one.

Interacting with operating systems

One of the best features of Julia is its great REPL, which provides users with a lot of flexibility while calling OS-specific commands.

To call any operating system command from inside the Julia REPL, we just have to press a ";" key and the prompt immediately changes:

```
julia >; # As soon as you press this ";"  
  
shell >
```

The change in the prompt is immediate, and on the same line. To start with, Julia provides a command named `pwd()`, which is synonymous with the one we used in a Linux operating system to discover a user's current directory:

```
shell> pwd  
/home/myuser  
  
julia> pwd()  
"/home/myuser"
```

Notice closely, for the first command, we have used a ";" and then called the `pwd` command in shell mode. While for the next one, we have used the `pwd()` function inside the Julia prompt. We will now be discussing some of the major functions used for performing filesystem operations as well as I/O operations.

Filesystem operations

Accessing files and doing operations over them is one of the most basic operations, and one can achieve the desired results by having the right knowledge about functions that interact with the filesystem. Given here are some of the most useful functions, along with some examples.

- `homedir()` and `joinpath()` functions: Julia is platform independent. This means that it runs on any platform without any hiccups and you can easily make a script which is cross-platform and can be used by everyone:

That being said, how to handle the different conventions while using directories and paths across different operating systems? This question is easily addressed by the two functions `homedir` and `joinpath`. Have a look at the following example:

```
# check the home directory  
julia> homedir()  
"/Users/myuser"  
  
# can be used to create paths which can be used further  
julia> joinpath(homedir(), "Documents")
```

```
"/Users/myuser/Documents"
```

We will now be moving on to other commands, which can make use of these 2 functions and work seamlessly across any platform.

- `cd()`: Lets the user change the current directory to the desired one. The usage is simple. To see the effects of the directory change, we will be using `pwd()` function, which we studied earlier. Here is an example:

```
julia> pwd()
"/Users/myuser"

julia> cd("../")

julia> pwd()
"/Users/"
```

- `readdir` function: This function is used for reading the contents of any specific directory. One example is as follows:

```
# homedir() is cross platform
julia> d = homedir()

julia> readdir(d)
45-element Array{String,1}:
".CFUserTextEncoding"
".DS_Store"
".Trash"
".anaconda"
".atom" ...
"dosbox"
"dottest.dot"
"dottest.png"
"test"
"transformations.jl"
```

Here, we have passed the "d" directory (which we initialized with home directory) as an argument to the function, which then shows all the contents of this directory as an array.

- `mkdir` function: This is used to create an empty directory, along with the permissions to create it. Here is an example of the function used:

```
# cross-platform
julia> d = joinpath(homedir(), "LearningJulia")

julia> mkdir(d)
```

```
julia> cd("LearningJulia")

julia> pwd()

"/Users/julians/LearningJulia"
```

The function `mkdir()` can also take a second argument that has to be an unsigned integer, wherein the permissions can be passed. The permissions are calculated as (r,w,x), and more on file permissions can be read at https://en.wikipedia.org/wiki/File_system_permissions.

- `stat` function: This function is very useful, if you want to have a lot of information about a specific file. To show what this function is capable of, let's create an empty file with the name `sample.jl`, and run this function over the file:

```
julia> stat("sample.jl")
StatStruct(mode=100644, size=0)

julia> stat("sample.jl").ctime
1.502545241e9

julia> stat("sample.jl").mtime
1.502545241e9

julia> stat("sample.jl").size
0

julia> stat("sample.jl").uid
0x000000006d16c610

julia> stat("sample.jl").gid
0x0000000064cbf778

julia> stat("sample.jl").rdev
0x0000000000000000

julia> stat("sample.jl").blksize
4096

julia> stat("sample.jl").blocks
0

julia> stat("sample.jl").device
0x0000000001000004

julia> stat("sample.jl").mode
```

```
0x000000000000081a4
```

```
julia> stat("sample.jl").inode
```

```
0x0000000000002f538d
```

- `cp` and `mv` functions: These two functions mimic the two basic functionalities of copying and moving a file from a source to a destination. The following simple example shows their usage:

```
julia> cp(".viminfo", "new_viminfo")
```

```
julia> isfile("new_viminfo")  
true
```

```
julia> mv("new_viminfo", "viminfo.bkp")
```

```
julia> isfile("viminfo.bkp")  
true
```

In the first command, the `cp` function copies `.viminfo` to a new location, which is `new_viminfo`; however, on the other hand, `mv` moves this new `new_viminfo` file to another file called `viminfo.bkp` (or basically renames it).

- `isdir`, `homedir`, `basename`, `dirname`, and `splitdir`: These are very useful functions while getting to know more about the directories. Given here is a set of examples that clearly help in understanding the differences between these commands, and where they can be used:

```
# to check whether the given path is a directory or not.
```

```
julia> isdir(joinpath(homedir(), "Documents"))  
true
```

```
julia> isdir("sample.txt")  
false
```

```
# To know your home directory.
```

```
julia> homedir()  
"/Users/myuser"
```

```
# to know the exact name of the directory you are in,  
without the full path.
```

```
julia> basename(homedir())  
"myuser"
```

```
# To know about the directory which has the directory you  
are looking for.
```

```
julia> dirname(homedir())
```

```
"/Users"
```

```
# To split the directory path and split them into a tuple
julia> splitdir(homedir())
("/Users", "myuser")
```

- `mkpath`, `ispath`, `abspath`, and `joinpath`: Sometimes, while working with files and directories, you may want to know if you are in the right path, or whether the file that you are trying to access is in the right path or not. To resolve such issues and to make sure they don't occur, keeping a check on the current path is important. Given here are the four most important path-specific commands, along with their examples:

```
# To check whether the path exists or not
julia> ispath(homedir())
true

julia> ispath(joinpath(homedir(), "random"))
false

# To know the absolute path of a directory
julia> abspath("Documents")
"/Users/rahullakhanpal/Documents"

# To make a path and subsequent directories in it
julia> mkpath(joinpath("Users", "adm"))

#To make sure the path was created in real!
julia> for (root, dirs, files) in walkdir("Users")
    println("Directories in $root")
end
Directories in Users
Directories in Users/adm
=#
```

I/O operations

After having read about some filesystem operations, let's move on to I/O and network related tasks. Here, we will be going through a list of some of the most used functions, while performing the operations mentioned earlier.

The `STDOUT`, `STDERR`, and `STDIN` are the three global variables inside Julia for denoting a standard output, error, and input stream respectively.

- `open()` function: This function is used to open up a file for the purpose of reading or writing a file. A very simple use case would be to create a file named `sample.txt` in your current directory, and open Julia REPL:

```
julia> file = open("sample.txt")
IOStream(<file sample.txt>)

julia> file
IOStream(<file sample.txt>)
```

Now, since the file is opened, what do we do with it? We probably will go ahead and read the contents of it. Let's suppose we had `hello world!!` written inside the `sample.txt` file; what happens when we try to read its contents now? Take a look at the following code:

```
julia> lines = readlines(file)
1-element Array{String,1}:
"hello world!!\n"
```

The function `readlines()` will specifically be used to read all the contents of this file. The result comes out as an `Array of String`. However, there is more than one way of opening a file in Julia, and that depends upon the mode in which we open the file. The common ones are `r`, `w`, and `r+`, where `r+` stands for read as well as write.

- **write and read function:** This function, as the name suggests, is used to write and read contents to and from a file, respectively. A quick example shown here will help you gain a basic understanding of both these functions:

```
# open up a file named "sample.txt" and write the message
julia> write("sample.txt", "hi how are you doing?")
21

julia> read("sample.txt")
21-element Array{UInt8,1}:
0x68
0x69
0x20
0x68
0x6f
0x77
0x20
0x61
0x72
0x65
0x20
0x79
0x6f
0x75
0x20
0x64
```

```

0x6f
0x69
0x6e
0x67
0x3f

# to actually read the contents
julia> readline("sample.txt")
"hi how are you doing?"

```

Here, first, we have called the write function to open up a file `sample.txt`, and write the contents "hi how are you doing?". Next up, we now use the read function to open up the same file and try to read the contents. But the result is not what we expected!

Instead, we get an `Array{UInt8,1}` type of array, which shows that we are trying to access a stream of unsigned integers. A close look at the methods for this function further clears the confusion:

```

julia> methods(read)
# 37 methods for generic function "read":
read(::Base.DevNullStream, ::Type{UInt8}) at coreio.jl:13
read(s::IOStream) at iostream.jl:236
read(s::IOStream, ::Type{UInt8}) at iostream.jl:151
read(s::IOStream, T::Union{Type{Int16},Type{Int32},Type{Int64},Type{UInt16},Type{Int32},Type{UInt64}}) at iostream.jl:160
read(s::IOStream, ::Type{Char}) at iostream.jl:180
read(s::IOStream, nb::Integer; all) at iostream.jl:260
...
..
.

```

To close the file after reading, we have a function named `close()` that takes in the name of the file as its parameter:

```
julia> close("sample.txt")
```

Example

Now that we have read about filesystem and I/O operations, we can now give a fully fledged example of how a simple Julia script can be created to read and write data into a file.

Here, we have a file named `sample.jl`, which we have created in the following manner:

```

# Arguments
in_file = ARGS[1]

```

```

out_file = ARGS[2]

# Keeping track using a counter
counter = 0
for line in eachline(in_file)
    for word in split(line)
        if word == "Julia"
            counter += 1
        end
    end
end

# write the contents to the output file
write(out_file, "the count for julia is $counter")

# read the contents from the o/p file for user's help
for line in readlines(out_file)
    println(line)
end

```

This script basically takes in two inputs:

- `in_file`, which is a TXT file from which we will be reading the data
- `out_file`, to which we will be writing the result

The `in_file` that we are going to supply has the name `readme.txt`, and contains the following information (the following text has been taken from the official Julia website):

```
mylinux-machine:home myuser$ cat readme.txt
```

Julia is a high-level, high-performance dynamic programming language for numerical computing. It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library. Julia's Base library, largely written in Julia itself, also integrates mature, best-of-breed open source C and Fortran libraries for linear algebra, random number generation, signal processing, and string processing. In addition, the Julia developer community is contributing a number of external packages through Julia's built-in package manager at a rapid pace. Julia, a collaboration between the Jupyter and Julia communities, provides a powerful browser based graphical notebook interface to Julia.

While the `out_file` will be the file we write to, in this case, we have `result.out`. Now we go ahead and call the script in the following manner from the Command Prompt:

```

mylinux-machine:home myuser$ julia sample.jl
readme.txtresult.out
the count for julia is 4

```


If you examine the script `sample.jl` closely, we are just reading in a file that we have passed as an argument (in this example, `readme.txt`), and computing the number of times the exact word Julia appears in the text. Finally, the total number of words are written into the file `result.out`:

```
mylinux-machine:home myuser$ cat result.out
the count for julia is 4
```

Calling C and Python!

Julia, as we know it from our very first introduction, takes the best from both worlds. It matches Python in terms of ease of code and maintenance, while it targets to achieve the speeds of C.

But what if we really need to make outside calls to code or functions written in these two languages? We then require the ability to import the code directly into Julia and be able to make use of it. Let's see, one by one, how Julia manages to make external calls to these two programming languages.

Calling C from Julia

C can be called as the mother of modern-day programming languages, and most of the languages today use C somewhere in their source codes to either make their code run quicker or just to add a wrapper over an existing C function or library.

Julia also makes use of C in some of its libraries, although most of the core libs are written in Julia itself. There are some things that make Julia stand apart from the crowd when it comes to calling C. They are as follows:

- Make calls to C without any hassle
- Absolutely no overhead
- No further processing or compilation needed before calling the C function, hence it can be used directly

Now before we move on to actually start calling C from Julia, let's take a look at what a compiler (like LLVM) needs to know before it makes a call to any C function.

- Name of the library
- Name of the function itself
- Number and types of the arguments (also called Arity)
- The return type of the function
- Values of the arguments passed

The function which does this all in Julia is called `ccall()`. Its syntax can be written as the following:

```
ccall(:name, "lib", return_type, (arg1_type, arg2_type...), arg1, arg2)
```

The following is a simple example of how to make a call to a C function using `ccall`. Here we are calling the `clock` function from the standard C library itself, and it will return a value which will be an `Int64` type value:

```
julia> ccall(:clock, :libc, Int64, ())
1437953
```

Here we can also do something like this:

```
julia> ccall(:clock, :libc), Cint, ())  
1467922
```

Notice that we were able to replace `Int64` with `Cint`. But what is `Cint`? `Cint` here is actually the C equivalent of signed `int` c-type, if you open up the REPL:

```
julia> Int32 == Cint  
true
```

Is that surprising? No! The reason is that Julia makes use of these aliases for C-based types. There are more of them defined such as `Cint`, `Cuint`, `Clong`, `Culong`, and `Cchar`. Let's try another example, but a complex one:

```
julia> syspath = ccall(:getenv, :libc, Ptr{Cchar}, (Ptr{Cchar},), "SHELL")  
Ptr{Int8} @0x00007fff5ca5bbe0  
  
julia> unsafe_string(syspath)  
"/bin/bash"
```

Here we are making `ccall` to the standard C library to make use of the `getenv` function and getting the `SHELL` value from it, which in itself is a Julia string.

However, here we are passing an argument of type `Ptr{Cchar}` and the expected argument type of the result is also `Ptr{Cchar}`, both of which represent a Julia type (pointer to a character). So the overall result we are trying to get here is the value of the `SHELL` parameter from the environment variables.

But how does this execution happen? Before even `ccall` is called, there is a call to convert function internally, which converts `SHELL` (which is a Julia string) to a `Ptr{Cchar}` type.

After this the actual call happens that returns a `Ptr{Int8} @0x00007fff5ca5bbe0` value, wherein there is a Pointer to an `Int8` along with the buffer address. To make this more readable, we make use of a function by the name `unsafe_string` that converts it to a Julia String type.

The reason why this function is prefixed with `unsafe` is that it will crash if the value of the pointer is not a valid memory address. This is popularly called a segmentation fault. However, here it works as usual as `Ptr{Int8} @0x00007fff5ca5bbe0` is a valid memory address.

The reason why we are able to get the value as a Julia type is that there are lots of conversions defined as standard while doing calls to C-based Strings and Integers. Even C structs can be replicated to Julia using composite types.

Calling Python from Julia

Using another language inside Julia may not be the best of ideas as far as speed and efficiency is concerned. But, there may be situations when you would be required to do so. Here, we talk about a package named PyCall, which is used to import Python code in Julia.

Here is how PyCall can be downloaded from inside a Julia shell:

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

Once it is installed, we can start using Python commands right away. Given here are some code snippets that let you quickly understand how easy it is to invoke and use in Julia right away:

```
# Invoke PyCall
julia> using PyCall

# Starting with a simple hello world
julia> py"""
print 'hello world'
"""
hello world

# Some random operations
julia> py"len('julia') "
5
julia> py"str(5) "
"5"
```

As you can see, we have been using the syntax; `py"""` to invoke and run common Python functions. But what if we need to invoke third-party libraries such as Numpy? We then make use of the `@pyimport` macro, which lets us use them. Here is one such quick example:

```
julia> @pyimport numpy as np

julia> np.sin(120)
0.5806111842123143
```

It should be noted that the syntax of Python code we are using should be according to the Python path set in the environment variable. This can easily be rechecked as:

```
julia> ENV["PATH"] "/Applications/Julia-0.6.app/Contents/Resources/julia/bin:/Library/Frameworks/Python.framework/Versions/3.6/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Applications/Julia-0.6.app/Contents/Resources/julia/bin:/usr/local/go/bin:/usr/local/mysql/bin"
```

Now, what if we want to use Python's built-in data types, such as `dict`? The PyCall has a solution for that too, in the form of `pybuiltin`. Let's see a detailed example:

```

# Using pybuiltin, directly create a python dict object
julia> pybuiltin(:dict)(a=1,b=2)
Dict{Any,Any} with 2 entries:
"b" => 2
"a" => 1

julia> pycall(pybuiltin("dict"), Any, a=1, b=2)
PyObject {'a': 1, 'b': 2}

julia> d = pycall(pybuiltin("dict"), Any, a=1, b=2)
PyObject {'a': 1, 'b': 2}

# making sure of the type, still a python object
julia> typeof(d)
PyCall.PyObject

# using pyDict to convert python's object to a julia dict object
julia> julia_dictionary=PyDict{Symbol,Int64}(pycall(pybuiltin("dict"), Any, a=1, b=2))
PyCall.PyDict{Symbol,Int64,true} with 2 entries:
:a => 1
:b => 2

# as we see, it's now a julia dict object
julia> typeof(julia_dictionary)
PyCall.PyDict{Symbol,Int64,true}

# accessing dictionary elements
julia> julia_dictionary[:a]
1

# accessing dictionary elements
julia> julia_dictionary[:b]
2

# a function that takes kw arguments
julia> f(; a=0, b=0) = [10a, b]
f (generic function with 1 method)

julia> f(;julia_dictionary...)
2-element Array{Int64,1}:
10
2

```

Expressions and macros

Metaprogramming is fun, and Julia definitely has one of the best metaprogramming features compared with some of its rivals. The initial cues and inspiration have been taken from Lisp, and similarly to Lisp, Julia is written in Julia itself—or in other words, Julia is homoiconic.

To explain how Julia interprets a code, here is a small code snippet:

```
julia> code = "println(\"hello world \") "  
"println(\"hello world \") "  
  
julia> expression = parse(code)  
:(println("hello world "))  
  
julia> typeof(expression)  
Expr
```

Here, we have simply passed a normal Julia code `println("hello world")` as a string to the function `parse`. This function, in turn, takes this piece of string and converts it into a data type called `Expr`, which is evident from the preceding code.

To see what's inside this `Expr` type of data closely, we can dig deeper:

```
julia> fieldnames(expression)  
3-element Array{Symbol,1}:  
:head  
:args  
:typ  
  
julia> expression.args  
2-element Array{Any,1}:  
:println  
"hello world "  
  
julia> expression.head  
:call  
  
julia> expression.typ  
Any
```

Any `Expr`-type object has three parts:

- Symbols, which, in this case, are `:call` and `:println`
- A series of arguments, which, in this case, are `:println` and `"hello world "`
- Finally, a result type, which, here, is `Any`

Julia also provides another function called `dump`, which helps to provide detailed information about the expression-type objects:

```
julia> dump(expression)
Expr
  head: Symbol call
  args: Array{Any}((2,))
    1: Symbol println
    2: String "hello world "
  type: Any
```

But, we know what arguments and return types are. The one thing we don't have a good understanding of is a symbol!

Symbols in Julia are the same as in Lisp, Scheme or Ruby. When a language can represent its own code, it needs a way to represent things like assignments, function calls, things that can be written as literal values. It also needs a way to represent its own variables. That is, you need a way to represent it as data.

```
Let's take the example:
julia> foo == "foo"
```

The difference between a symbol and a string is the difference between `foo` on the left hand side of that comparison and `"foo"` on the right hand side. On the left, `foo` is an identifier and it evaluates to the value bound to the variable `foo` in the current scope. On the right, `"foo"` is a string literal and it evaluates to the string value `"foo"`. A symbol in both Lisp and Julia is how you represent a variable as data. A string just represents itself.

There is one more way to create an expression, which is by using an `Expr` constructor. Given here is a most basic type of expression that one can create:

```
julia> sample_expr = Expr(:call, +, 10, 20)
:((+) (10,20))

julia> eval(sample_expr)
30
```

So, what is happening here? We have created a custom expression wherein we are passing `:call` as the first argument, which symbolized the head of the expression. Next, we supply `+`, `10`, and `20` as the arguments to this expression. To evaluate this expression finally at runtime, we use a function named `eval`:

```
julia> sample_expr.args
3-element Array{Any,1}:
+
10
```

```
20
```

```
julia> eval(sample_expr)
30
```

Please take note that `eval` takes an `Expr` -type value as input, which is why the data type of `sample_expr` is `Expr`.

Now, suppose we wanted to evaluate this expression instead:

```
julia> sample_expr = Expr(:call, +, x, y)
ERROR: UndefVarError: x not defined
```

This throws an error, saying `x` is not defined, which indeed is the case too. But how do we go about it? How do we replace these values with the one we want to pass? Or, in other words, how do we interpolate these variables?

One way is to inject the values of `x` and `y` directly in the expression we are trying to create and later evaluate:

```
julia> x = 10
10

julia> y = 10
10

julia> sample_expr = Expr(:call, +, :x, :y)
:((+) (10,10))

# or even this works
julia> sample_expr = Expr(:call, +, x, y)
:((+) (10,10))

julia> eval(sample_expr)
20
```

Another way to achieve the same results is to use `$`, but, in this case, we won't be interpolating the values at runtime; rather, we will be doing it at the time of parsing the expression. See the following example:

```
julia> x = 10
10

julia> y = 10
10

julia> e = :($x + $y)
:(10 + 10)
```

```
julia> eval(e)
20
```

Hence, overall there are two ways by which interpolation can be done with `Expr` objects:

- Using quotes (`:`) at runtime
- Using dollar (`$`) at parse time

Another way of expressing the expression is by using the `quote` keyword. For the most part, using a quoted expression is synonymous with the one we have been using so far—for example, expression preceded by a `:"` key:

```
julia> quote
    30 * 100
end
quote # REPL[12], line 2:
    30 * 100
end

julia> eval(ans)
3000

julia> :(30 * 100)
:(30 * 100)

julia> eval(ans)
3000
```

As you may have noticed, there is no difference between the two! But then, why is `quote` available separately to the end user? The main reason, as the official Julia documentation states, is because when using `quote`, this form introduces `QuoteNodes` elements to the expression tree, which must be considered while manipulating the tree.

In other words, using `quote`, you can splice what is implemented better.

Macros

Macros in Julia are a very powerful tool for code evaluation, and in some of the previous chapters, we have been using them regularly (for instance, using `@time` to know the overall compute time of a program).

Macros are similar to functions, but where functions take in normal variables as arguments, macros, on the other hand, take expressions and return modified expressions.

Note: Functions are evaluated at runtime. Macros are evaluated at parse time. This means that macros can manipulate functions (and other code) before they are ever executed.

The syntax of a macro could be defined as follows:

```
macro NAME
    # some custom code
    # return modified expression
end
```

At the time of calling a macro, the `@` symbol is used, which is used to denote a macro in Julia. This symbol is, however, similar to the ones used in other languages, such as decorators in Python:

```
julia> macro HELLO(name)
    :( println("Hello! ", $name))
end
@HELLO (macro with 1 method)
julia> @HELLO("Raaaul")
Hello! Raaaul
```

To see what's going on inside a macro and to help debug them better, we can use the Julia function with the name `macroexpand`:

```
julia> macroexpand(:(@HELLO("rahul")))
:(println("Hello!", "rahul"))
```

But why metaprogramming?

To understand why metaprogramming is used, have a look at this scenario. You have a function similar to the one given here, which takes a number and prints it the given number of times:

```
julia> function foo(n::Int64)
    for i=1:n
        println("foo")
    end
end
foo (generic function with 1 method)
julia> foo(2)
foo
foo
```

Simple? Yes. But now suppose, in some different module in your code, you are doing the same kind of operation, but with some other name:

```
julia> function bar(n::Int64)
    for i=1:n
        println("bar")
    end
end
```

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

```
julia> bar(2)
```

```
bar
```

```
bar
```

```
julia> function baz(n::Int64)
```

```
for i=1:n
```

```
println("baz")
```

```
end
```

```
end
```

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

```
julia> baz(2)
```

```
baz
```

```
baz
```

A lot of code repetition, right? You certainly want to avoid this problem, and here is where metaprogramming comes to the rescue. With metaprogramming, you can essentially generate code on the fly, which can cut down your time as a developer.

So, to make sure we don't repeat ourselves, let's see what we can do instead:

```
julia> for sym in [:foo, :bar, :baz]
```

```
    @eval function $(Symbol(string(sym))) (n::Int64)
```

```
        for i in 1:n
```

```
            println("$sym")
```

```
        end
```

```
    end
```

```
end
```

```
julia> foo(1)
```

```
foo
```

```
julia> bar(1)
```

```
bar
```

```
julia> baz(1)
```

```
baz
```

So as you can see, we were able to do the same with the help of metaprograms! In the next section, we will be talking about some very important built-in macros available in Julia!

Built-in macros

We will be focusing on discussing some of the most used ones out of these. Let's start exploring them:

- `@time`: This one is a useful macro to find out the total time taken by a program to complete. In other words, we can use it to keep track of our program's execution speed. Its usage, along with a small example, is given here:

```
# simple function to find recursive sum
julia> function recursive_sum(n)
    if n == 0
        return 0
    else
        return n + recursive_sum(n-1)
    end
end
recursive_sum (generic function with 1 method)

# A bit slow to run for the 1st Time, as the function gets
compiled.
julia> @time recursive_sum(10000)
0.003905 seconds (450 allocations: 25.816 KiB)
50005000

# Much much faster in the second run!
julia> @time recursive_sum(10000)
0.000071 seconds (5 allocations: 176 bytes)
50005000
```

Julia is good at scientific calculations, and `@time` comes in handy when understanding the time taken by the program to run.

- `@elapsed`: Closely resembling the `@time` macro is the `@elapsed` macro, which discards the result and just displays the time taken by the program to run. Reapplying `@elapsed` to the previous average function:

```
julia> @elapsed average(10000000, 1000000000)
2.144e-6

julia> typeof(@elapsed average(10000000, 1000000000))
Float64
```

The result of `@elapsed` is always represented in floating point numbers.

- `@show`: This macro, when used along with any piece of code, will return an expression as well as compute the result of it. The sample given here will be helpful in using `@show`:

```

julia> @show(println("hello world"))
hello world
println("hello world") = nothing

julia> @show(: (println("hello world")))
$(Expr(:quote, : (println("hello world")))) = : (println("hello
world"))
: (println("hello world"))

julia> @show(: (3*2))
$(Expr(:quote, : (3 * 2))) = : (3 * 2)
: (3 * 2)

julia> @show(3*2)
3 * 2 = 6
6

julia> @show(Int64)
Int64 = Int64
Int64

```

- **@which:** This macro is quite useful when you have multiple methods for a single function and you want to inspect or know about the method that would be called when a specific set of arguments will be supplied. Because Julia relies heavily on multiple dispatch, the @which macro comes in handy at a lot of places. The following is a example which showcases its usage.

```

# create a function that tripples an Integer
julia> function tripple(n::Int64)
    3n
end
tripple (generic function with 1 method)

# redefine the same function to accept Float
julia> function tripple(n::Float64)
    3n
end
tripple (generic function with 2 methods)

# check the methods available for this function
julia> methods(tripple)
# 2 methods for generic function "tripple":
tripple(n::Float64) in Main at REPL[22]:2
tripple(n::Int64) in Main at REPL[21]:2

# Get the correct method , when 'n' is an Int64

```

```
julia> @which tripple(10)
tripple(n::Int64) in Main at REPL[21]:2

# Get the correct method , when 'n' is Float64
julia> @which tripple(10.0)
tripple(n::Float64) in Main at REPL[22]:2
```

So, as you can see, for given a set of arguments, @which was able to tell the correct method for a function.

- **@task:** A task in Julia is similar to a coroutine. This macro can be used to return a task without running it, and hence it can be run later on. We will now try to create a very simple task and show how @task can be used to run it at a later time:

```
julia> say_hello() = println("hello world")
say_hello (generic function with 1 method)

julia> say_hello_task = @task say_hello()
Task (runnable) @0x000000010dcdfa90

julia> istaskstarted(say_hello_task)
false

julia> schedule(say_hello_task)
hello world
Task (queued) @0x000000010dcdfa90

julia> yield()

julia> istaskdone(say_hello_task)
true
```

- **@code_llvm, @code_lowered, @code_typed, @code_native, and code_warntype:** These macros are all related to the way the code gets represented in Julia, as well as how it interacts with the layer beneath it. It basically digs down an extra layer and helps you to debug and know what's going on behind the scenes. Let's take up a simple example of a fibonacci sequence:

```
julia> function fibonacci(n::Int64)
    if n < 2
        n
    else
        fibonacci(n-1) + fibonacci(n-2)
    end
end
```

```

fibonacci (generic function with 1 method)

# OR, can also define it this way
julia> fibonacci(n::Int64) = n < 2 ? n : fibonacci(n-1) +
fibonacci(n-2)

```

```

fibonacci (generic function with 1 method)

julia> fibonacci(10)
55

```

Now let's try each of these macros on this piece of code one by one:

The `@code_lowered` displays code in a format that is intended for further execution by the compiler. This format is largely internal and isn't intended for human usage. The code is transformed into a single static assignment, in which each variable is assigned only once, and every variable is defined before it is used:

```

julia> @code_lowered fibonacci(10)
CodeInfo(: (begin
    nothing
    unless n < 2 goto 4
    return n
  4:
    return          e(Main.fibonacci) (n          -
    1)+(Main.fibonacci) (n - 2)
end))

```

The `@code_typed` represents a method implementation for a particular set of argument types after type inference and inlining:

```

julia> @code_typed fibonacci(10)
CodeInfo(: (begin
    unless (Base.slt_int) (n, 2)::Bool goto 3
    return n
  3:
    SSAValue(1) = $(Expr(:invoke, MethodInstance for
fibonacci(::Int64), :(Main.fibonacci), :((Base.sub_int) (n,
1)::Int64)))
    SSAValue(0) = $(Expr(:invoke, MethodInstance for
fibonacci(::Int64), :(Main.fibonacci), :((Base.sub_int) (n,
2)::Int64)))
    return          (Base.add_int) (SSAValue(1),
SSAValue(0))::Int64 end))=>Int64

```

```
julia> @code_warntype fibonacci(10)
Variables:
  #self#::#fibonacci
  n::Int64

Body:
  begin
    unless (Base.slt_int)(n::Int64, 2)::Bool goto 3
    return n::Int64
  3:
    SSAValue(1) = $(Expr(:invoke, MethodInstance for
fibonacci(::Int64), :(Main.fibonacci), :((Base.sub_int)(n,
1)::Int64)))
    SSAValue(0) = $(Expr(:invoke, MethodInstance for
fibonacci(::Int64), :(Main.fibonacci), :((Base.sub_int)(n,
2)::Int64)))
    return (Base.add_int)(SSAValue(1),
SSAValue(0))::Int64 end::Int64
```

Julia uses the LLVM compiler framework to generate machine code. It uses the LLVM's C++ API to construct this LLVM intermediate representation. So when we do `@code_llvm`, the code that it generates is just the intermediate representation along with some high-level optimizations:

```
julia> @code_llvm fibonacci(10)

define i64 @julia_fibonacci_61143.2(i64) #0 !dbg !5 {
top:
  %1 = icmp sgt i64 %0, 1
  br i1 %1, label %L3, label %if

if: ; preds = %top
  ret i64 %0

L3: ; preds = %top
  %2 = add i64 %0, -1
  %3 = call i64 @julia_fibonacci_61143(i64 %2)
  %4 = add i64 %0, -2
  %5 = call i64 @julia_fibonacci_61143(i64 %4)
  %6 = add i64 %5, %3
  ret i64 %6
}
```

Julia uses and executes native code. The `@code_native` represents exactly that, and it's just a binary code in memory. This one resembles the assembly language closely enough, which represents instructions:

```
julia> @code_native fibonacci(10)
.section __TEXT,__text,regular,pure_instructions
Filename: REPL[50]
    pushq %rbp
    movq %rsp, %rbp
    pushq %r15
    pushq %r14
    pushq %rbx
    pushq %rax
    movq %rdi, %rbx
Source line: 1
    cmpq $1, %rbx
    jle L63
    leaq -1(%rbx), %rdi
    movabsq $fibonacci, %r15
    callq *%r15
    movq %rax, %r14
    addq $-2, %rbx
    movq %rbx, %rdi
    callq *%r15
    addq %r14, %rax
    addq $8, %rsp
    popq %rbx
    popq %r14
    popq %r15
    popq %rbp
    retq

L63:
    movq %rbx, %rax
    addq $8, %rsp
    popq %rbx
    popq %r14
    popq %r15
    popq %rbp
    retq
    nopl (%rax)
```


Type introspection and reflection capabilities

Type introspection and reflection capabilities form a very useful part of any modern-day programming language. Their need arises due to the fact that many a time while coding, we come across a situation where we need to understand the types of objects or data that we are dealing with. Sometimes, we may need to find the type of an object, and other times we may end up coding some logic based on that object's types and properties.

Type introspection

As we know from the starting chapters that Julia supports multiple dispatch and we can create a new data type from an abstract data type, let's define a new type, named `Student`, and then create two sample objects for this type:

```
julia> type Student
        name::String
        age::Int64
    end

julia> alpha = Student("alpha",24)
Student("alpha", 24)

julia> beta = Student("beta",25)
Student("beta", 25)
```

Now that we have two of these students, with the names `alpha` and `beta`, how can I be sure of which type they are? You should definitely be thinking of a function that we studied earlier, remember? If not, then let's look at the following example for the answer:

```
julia> typeof(alpha)
Student

julia> typeof(beta)
Student
```

The `typeof` function was the one. But what if we wanted to check whether an object is of a given type using a single function? Let's take a look at the following code:

```
# similar to isinstance in python
julia> isa(alpha, Student)
true

# even this is possible!
julia> alpha isa Student
true
```

If you look at the second implementation, that is, `alpha` is a `Student`, you can see an inline syntax being used. How easy is that for the end user to read and understand? It is for this and many reasons that Julia is such a nice language to read.

Reflection capabilities

Reflection actually provides you with the ability to manipulate the attributes of an object at runtime. So whenever we create a function in Julia, we can ask some basic things about it, such as how many arguments does that function have, or more likely, what are the methods of that function available in the current scope, and so on.

To understand better, have a look at this code snippet, which creates a function and then tries to ask some questions about its properties:

```
# the first method tries to take in all integer values
julia> function
calculate_quad(a::Int64,b::Int64,c::Int64,x::Int64)
return a*x^2 + b*x + c
end
calculate_quad (generic function with 2 methods)

julia> calculate_quad(1,2,3,4)
27

# the second method takes all but x as integer values
julia> function
calculate_quad(a::Int64,b::Int64,c::Int64,x::Float64)
return a*x^2 + b*x + c
end
calculate_quad (generic function with 3 methods)

julia> calculate_quad(1,2,3,4.75)
35.0625

# to know what all methods does the function supports
# which as we can see that there are 2 currently
julia> methods(calculate_quad)
# 3 methods for generic function "calculate_quad":
calculate_quad(a::Int64, b::Int64, c::Int64, x::Float64) in
Main at REPL[31]:2
calculate_quad(a::Int64, b::Int64, c::Int64, x::Int64) in Main
at REPL[29]:2
calculate_quad(a, b, c, x) in Main at REPL[27]:2
```

This was a demonstration of how method signatures can be known for a function. Next is how to know what all the fields are inside a type. For that, we use a function called `fieldnames`, which gives all the names of the fields declared inside of the type:

```
# from the already declared Student class
julia> fieldnames(Student)
2-element Array{Symbol,1}:
 :name
 :age
```

On the other hand, if you want to know the individual field types of all the fields inside a type, you may be prompted to use the `types` property of a type, which results in a `SimpleVector` holding all the data types:

```
julia> Student.types
svec(String, Int64)

julia> typeof(Student.types)
SimpleVector
```

Reflection is widely used in metaprogramming, and Julia, being homoiconic, uses this property to its advantage. In the previous topic, when we covered built-in macros, we talked a great deal about how Julia reads its own code, and how some macros (`@code_llvm`, `@code_lowered`, `@code_native`, `@code_typed`, and `@code_warntype`) break down the code into different readable formats. We strongly urge you to go back to the previous topic and try them out for yourselves.

Julia Programming

Module 7: Numerical and Scientific Computation with Julia

Learning Objectives

- Working with data.
- Linear algebra and differential calculus.
- Statistics [mean(), std(), median()].
- Optimization in Julia.

Working with data

There are various ways and sources from which we can get data. We can get data directly from the user through the Terminal or using a script or from a source file (which can either be a binary file or structured files like CSV or XML files). To see how Julia accepts data from the user from any of these resources, let's dive right into these ways one by one.

When starting to play with data for the very first time, it's obvious to use the Julia REPL or a notebook environment such as the IJulia notebook. Now, Julia understands every incoming bit of data in a byte stream. So if we try to use the regular built-in functions such as `read()` and `write()`, they will basically be oriented toward a binary I/O. Let's see a simple example of a `read()` operation in action:

```
# usage of read with Char
julia> read(STDIN, Char)
j
'j': ASCII/Unicode U+006a (category Ll: Letter, lowercase)
```

```
# usage of read with Bool
julia> read(STDIN, Bool)
true
true
```

```
# usage of read with Int8
julia> read(STDIN, Int8)
23
50
```

```
# method signature for Bool Type
julia> @which read(STDIN, Bool)
read(s::IO, ::Type{Bool}) in Base at io.jl:370
```

```
# method signature for Char Type
julia> @which read(STDIN, Char)
read(s::IO, ::Type{Char}) in Base at io.jl:403
```

```
# method signature for Int8 Type
julia> @which read(STDIN, Int8)
read(s::IO, ::Type{Int8}) in Base at io.jl:365
```

```
# Or Alternatively
julia> for val in [:Bool, :Char, :Int8]
@eval println(@which read(STDIN, $val))
end
```

```
read(s::IO, ::Type{Bool}) in Base at io.jl:370
read(s::IO, ::Type{Char}) in Base at io.jl:403
read(s::IO, ::Type{Int8}) in Base at io.jl:365
```

Read basically expects an I/O stream along with the type of data to be expected. However, we do not use `read()` often; rather we use a more generalized function by the name `readline()`, which helps us take the user input in the string format (along with a newline character in the end). Take a look at the following command:

```
julia> statement = readline()
julia is fast!
"julia is fast!"
```

```
julia> readline()
julia
"julia"
```

```
julia> methods(readline)
# 4 methods for generic function "readline":
readline(s::IOStream; chomp) in Base at iostream.jl:234
readline() in Base at io.jl:190
readline(filename::AbstractString; chomp) in Base at io.jl:184
readline(s::IO; chomp) in Base at io.jl:190
```

the `readline()` function (similar to the `input()` function of Python 3) returns the value in a `String` type. To be able to take in integer data from the user, we then have to use the `parse()` function:

```
julia> number = parse{Int64}(readline())
23

julia> println(number)
23
```

Working with text files

We now know how to read data from the user. But how about if we had a text file that was supposed to be read? We may then have to take the help of the `open` function, which helps us read files from the system.

```
julia> methods(open)
# 8 methods for generic function "open":
open(fname::AbstractString) in Base at iostream.jl:113
open(fname::AbstractString, rd::Bool, wr::Bool, cr::Bool, tr::Bool,
ff::Bool) in Base at iostream.jl:103
open(fname::AbstractString, mode::AbstractString) in Base at
iostream.jl:132
```

```
open(f::Function, cmds::Base.AbstractCmd, args...) in Base at
process.jl:599
open(f::Function, args...) in Base at iostream.jl:150
open(cmds::Base.AbstractCmd) in Base at process.jl:575
open(cmds::Base.AbstractCmd, mode::AbstractString) in Base at
process.jl:575
open(cmds::Base.AbstractCmd, mode::AbstractString,
other::Union{Base.FileRedirect, IO, RawFD}) in Base at process.jl:575
```

Let's take an example of a file stored at a location on my desktop, and try reading it in Julia. For the sake of simplicity, I have created a file by the name of sample.txt , which has the underlying five lines about Julia:

```
shell> cat sample.txt
Julia is a great and powerful language
It is homoiconic.
It supports static as well as dynamic typing.
Julia walks like python runs like C.
Julia uses multiple dispatch.
```

Let's open this file inside Julia:

```
# read file from the device folder in the current path
julia> file = open("sample.txt")
IOStream(<file sample.txt>)

# read the contents of file
julia> file_data = readlines(file)
5-element Array{String,1}:
"Julia is a great and powerful language"
"It is homoiconic."
"It supports static as well as dynamic typing."
"Julia walks like python runs like C."
"Julia uses multiple dispatch."
```

So far, we have successfully opened up the file and read its content. Also, we have been able to convert the contents of the file in to the array format, which will later help us do some quick operations. This is where the fun part starts!

```
# running an enumerate over the file provides us with a counter too!
julia> enumerate(file_data)
Enumerate{Array{String,1}}(String["Julia is a great and powerful
language",
"It is homoiconic.", "It supports static as well as dynamic typing.",
```

```
"Julia walks like python runs like C.", "Julia uses multiple  
dispatch."])
```

```
# using enumerate to our advantage to get the length of each line
```

```
julia> for lines in enumerate(file_data)
```

```
println(lines[1], "-> ", lines[2])
```

```
end
```

```
1-> Julia is a great and powerful language
```

```
2-> It is homoiconic.
```

```
3-> It supports static as well as dynamic typing.
```

```
4-> Julia walks like python runs like C.
```

```
5-> Julia uses multiple dispatch.
```

```
# convert everything to uppercase
```

```
julia> for line in file_data
```

```
println(uppercase(line))
```

```
end
```

```
JULIA IS A GREAT AND POWERFUL LANGUAGE
```

```
IT IS HOMOICONIC.
```

```
IT SUPPORTS STATIC AS WELL AS DYNAMIC TYPING.
```

```
JULIA WALKS LIKE PYTHON RUNS LIKE C.
```

```
JULIA USES MULTIPLE DISPATCH.
```

```
# reverse each line!!
```

```
julia> for line in file_data
```

```
println(reverse(line))
```

```
end
```

```
egaugnal lufrewop dna taerg a si ailuJ
```

```
.cinociomoh si tI
```

```
.gnipyT cimanyd sa llew sa citats stroppus tI
```

```
.C ekil snur nohtyp ekil sklaw ailuJ
```

```
.hctapsid elpitlum sesu ailuJ
```

```
# to simply count the number of lines in a file
```

```
julia> countlines("sample.txt")
```

```
5
```

```
# print the first line of the file
```

```
julia> first(file_data)
```

```
"Julia is a great and powerful language"
```

```
# print last line of the file
```

```
julia> last(file_data)
```


"Julia uses multiple dispatch."

Working with CSV and delimited file formats

Let's say we have a CSV file by the name `sample.csv` and we want to read its content. This is how we do it in Julia:

```
julia> csvfile = readcsv("sample.csv")
5×3 Array{Any,2}:
1 "James"          "UK"
2 "Lora"           "UK"
3 "Raj"            "India"
4 "Rangnatham"     "Sri lanka"
5 "Azhar"          "Bangladesh"
```

So using `readcsv`, we have converted the CSV file into an array of 5x3 matrix which can now be manipulated using the various array operations.

```
# getting only the names of the people
julia> csvfile[:,2]
5-element Array{Any,1}:
"James"
"Lora"
"Raj"
"Rangnatham"
"Azhar"
```

```
# getting only the top 3 data
julia> csvfile[1:3,:]
3×3 Array{Any,2}:
1 "James"  "UK"
2 "Lora"   "UK"
3 "Raj"    "India"
```

```
# reverse sorting the rows
julia> sortrows(csvfile, rev=true)
5×3 Array{Any,2}:
5 "Azhar"          "Bangladesh"
4 "Rangnatham"     "Sri lanka"
3 "Raj"            "India"
2 "Lora"           "UK"
1 "James"          "UK"
```

Working with DataFrames

They are basically a type of data structure that can hold the metadata about their data and hence they can be easily accessed as well as queried just like a database table does using SQL.

Julia provides a package named `DataFrames.jl`, which provides the necessary data structures for doing the job. It's the recommended source for doing DataFrame operations and as it's listed in the `METADATA.jl`.

```
# install the package
julia> Pkg.add("DataFrames")

# first time load takes times due to precompilation
julia> using DataFrames

# check the functions provided by DataFrames
julia> names(DataFrames)
254-element Array{Symbol,1}:
 :&
 Symbol("@csv2_str")
 Symbol("@csv_str")
 Symbol("@data")
 Symbol("@formula")
 Symbol("@pdata")
 Symbol("@tsv_str")
 Symbol("@wsv_str")
 Symbol("@~")
 :AIC
 :AICc
 :AbstractContrasts
 :AbstractDataArray
 :AbstractDataFrame
 :AbstractDataMatrix
 :AbstractDataVector
 ...
 :winsor
 :winsor!
 :wmean
 :wmean!
 :wmedian
 :wquantile
 :writetable
 :wsample
 :wsample!
```

```
:wsum
:wsum!
:xor
:zscore
:zscore!
:|
```

However, some of the most important data structures provided by the DataFrames packages are:

- `NA`: This corresponds to a missing value.
- `DataArray`: This provides extra functionalities in comparison with the default `Array` type in Julia.
- `DataFrame`: This is a 2D data structure and it's much like R `DataFrames` and Pandas `DataFrames`. It provides a much richer set of functions to manipulate data.

There are two more data structures by the name of `DataMatrix` and `DataVector`, but they are just type aliases of `DataArrays` of different dimensions.

```
julia> DataMatrix{Any} == DataArray{Any, 2}
true
```

```
julia> DataVector{Any} == DataArray{Any, 1}
true
```

NA

In the real world, every data that we get or apply our manipulations and transformations to can always have missing data. So the immediate question that follows is how do we handle that missing value?

In Julia, when we create an `Array`, suppose of length 5 with some random values, if we try having any array position as `nothing`, we can get something like:

```
julia> a = [1,2,3,nothing,5,6]
6-element Array{Any,1}:
 1
 2
 3
nothing
 5
 6

# try to access the element
julia> a[4]
```

```
# check the data type
julia> typeof(a[4])
Void
```

But we might not want to have this kind of empty data in our result set. After all, we might need to classify the data as Not Available or NA to make more sense in the result set. Hence, the NA data of the DataFrames structure can be used instead:

```
julia> using DataArrays
julia> a = DataArray([1.1, 2.2, 3.3, 4.4, 5.5, 6.6])
6-element DataArrays.DataArray{Float64,1}:
1.1
2.2
3.3
4.4
5.5
6.6

julia> a[1] = NA
NA

julia> a
6-element DataArrays.DataArray{Float64,1}:
NA
2.2
3.3
4.4
5.5
6.6

julia> typeof(a[1])
DataArrays.NAtype
```

One of the biggest features that are associated with using NA is that it may or may not prevent a function from modifying the values of a given data. In other words, if the data which we might work on does not contain NA, then the result won't be different from what is expected. Take a look at the following block of code for an illustration as to how the DataArray behaves in the presence or absence of NA :

```
julia> true || a
true

julia> true && a
6-element DataArrays.DataArray{Float64,1}:
NA
```

2.2
3.3
4.4
5.5
6.6

```
julia> mean(a)
NA
```

```
julia> mean(a[2:length(a)])
4.4
```

DataArrays

Let's take a look at what DataArrays in Julia look like:

```
# creating a 2D data matrix
julia> data_matrix = DataArray([1 2 3;4 5 6])
2×3 DataArrays.DataArray{Int64,2}:
1 2 3
4 5 6

# Creating a 1D vector
julia> data_vector = DataArray([1 2 3 4 5 6 7 8])
1×8 DataArrays.DataArray{Int64,2}:
1 2 3 4 5 6 7 8

# Creating a column major 1D array
julia> data_vector_column_major = DataArray([1,2,3,4])
4-element DataArrays.DataArray{Int64,1}:
1
2
3
4

# assigning the first value as NA
julia> data_vector_column_major[1] = NA
NA

# dropping NA values
julia> data_vector_column_major
4-element DataArrays.DataArray{Int64,1}:
NA
2
```

3
4

DataFrames

After discussing `NA` and `DataArrays`, now let's come to the most important data structure provided by the `DataFrames` module-- `DataFrame`. As previously discussed, they contain extra information about the data they hold, or, in other words, the metadata.

```
julia> dframe = DataFrame(Names = ["John","Ajay"], Age = [27,28])
2×2 DataFrames.DataFrame
┌ Row │ Names │ Age │
├───┼───┼───┼
│ 1   │ John  │ 27  │
│ 2   │ Ajay  │ 28  │
```

Now this dataset cannot be represented in the form of a `DataArray`. There are many reasons as to why `DataFrames` are unique in their purpose, including the following :

- They can hold data of different data types. A Matrix, which is just a 2D Array, cannot hold data of different data type, hence it's of no use when it comes to data from the real world. A `DataFrame` is thus heterogeneous in nature.
- They hold metadata, that is, labels in forms similar to a database table's headings for each column; thus the data inside them can be easily examined.
- Their individual rows are of the type `DataArray` and not vectors. Given here is the code that shows that individual rows and columns of `DataFrame` are `DataArrays` :

```
julia> dframe.columns[1]
2-element DataArrays.DataArray{String,1}:
"John"
"Ajay"

julia> typeof(dframe.columns[1])
DataArrays.DataArray{String,1}
```

Linear algebra and differential calculus

Julia is targeted toward the scientific community. Starting with this topic, we will see how Julia greatly eases mathematical calculations in the real world. We will start off by explaining how Julia proves to be a great resource for solving problems in linear algebra.

Linear algebra

The syntax used in Julia closely resembles that of **MATLAB**, but there are some important differences. To begin with, look at the matrix of some randomly generated numbers:

```
julia> A = rand(3,3)
3×3 Array{Float64,2}:
 0.770806  0.561571  0.787753
 0.605451  0.424917  0.646309
 0.551773  0.376415  0.834139
```

The `rand` function takes in parameters asking for the dimensions of the array, and as we have passed here a `(3,3)`, we get an array of size 3x3 of type `Float64`, containing random Gaussian numbers.

Similarly, we have another function named `ones`, which takes in a single parameter and reproduces an array containing `1.0`:

```
julia> ones(5)
5-element Array{Float64,1}:
 1.0
 1.0
 1.0
 1.0
 1.0
```

We need to know the difference between a vector and a matrix in Julia, as this could be confusing for those coming from a MATLAB background. So, in Julia, we have the following:

- `Vector{T}` is nothing but an alias for `Array{T, 1}`. This could be clearly seen in the Julia REPL:

```
julia> Vector{Float64} == Array{Float64,1}
true
```

- `Matrix{T}` is nothing but an alias for `Array{T, 2}`, which again could be easily proved in the Julia REPL:

```
julia> Matrix{Float64} == Array{Float64,2}
true
```

Let's now move on to some generic operations that can be done in linear algebra and try to get you at ease with most of these:

Multiplication: Unlike any other language, Julia gives you the freedom to write expressions in the form that you would usually do in your mathematical assignments, that is, without using any `*` keywords in between! However, you can still use it:

```
# create a random matrix of 3x3 dimension
julia> A
3×3 Array{Float64,2}:
0.673465 0.880229 0.100458
0.752117 0.545464 0.0180286
0.531316 0.221628 0.179626

# Easily, we could do
julia> b = 2 * A
3×3 Array{Float64,2}:
1.34693 1.76046 0.200917
1.50423 1.09093 0.0360573
1.06263 0.443255 0.359252
```

Matrix transpose: Again, very easy to follow!

```
julia> transpose_of_A = A'
3×3 Array{Float64,2}:
0.770806 0.605451 0.551773
0.561571 0.424917 0.376415
0.787753 0.646309 0.834139
```

Linear equations: Code in a way that you would write and solve on a paper! Julia makes it very easy to solve equations:

```
# define the value of x
julia> x = 5
5

# solve the equation
julia> equation = 3x^2 + 4x + 3
98
```

Differential calculus

Differential calculus is a branch of mathematics concerned with the determination, properties, and application of derivatives and differentials. In broader terms, these are concerned with the rate at which quantities change.

We will not go into much detail explaining this topic as it's vast and there are various internet resources for those unfamiliar with it. However, people with a strong math background will find themselves at home while discussing some of the problems.

For working with differential calculus, we will be using the Julia package `Calculus.jl`. Let's install the package and then start exploring its functions:


```

julia> Pkg.add("Calculus")
julia> using Calculus

# list of functions supported by Calculus
julia> names(Calculus)
22-element Array{Symbol,1}:
Symbol("@sexpr")
:AbstractVariable
:BasicVariable
:Calculus
:SymbolParameter
:Symbolic
:SymbolicVariable
:check_derivative
:check_gradient
:check_hessian
:check_second_derivative
:deparse
:derivative
:differentiate
:hessian
:integrate
:jacobian
:processExpr
:second_derivative
:simplify
:symbolic_derivative_bessel_list
:symbolic_derivatives_larg

```

Most of the differential calculus revolves around differentiation and integration. In the given example, we are using the ' operator to denote the differentiation levels of a function. Hence ''' would mean a third order differentiation:

```

julia> f(x) = sin(x)
f (generic function with 1 method)

julia> f'(1.0) - cos(1.0)
-5.036193684304635e-12

julia> f''(1.0) - (-sin(1.0))
-6.647716624952338e-7

julia> f'''(1.0) - (-cos(1.0))

```

```
0.11809095011119602
```

You can also perform symbolic differentiation, as depicted here:

```
julia> differentiate("cos(x) + sin(x) + exp(-x) * cos(x)", :x)
:(-(sin(x)) + cos(x) + (-(exp(-x)) * cos(x) + exp(-x) * -(sin(x))))

julia> differentiate("cos(x) + sin(y) + exp(-x) * cos(y)", [:x, :y])
2-element Array{Any,1}:
:(-(sin(x)) + -(exp(-x)) * cos(y))
:(cos(y) + exp(-x) * -(sin(y)))
```

Statistics

One of the strongest points of Julia has been its powerful support for statistical and mathematical functions. The Julia community has been very active in identifying the areas that can be covered using packages made purely in Julia, and hence we have now a group that deals purely in packages of Julia for stats. It's called JuliaStats, and can be found on GitHub at <https://github.com/JuliaStats>.

Some of the notable packages listed on the Julia stats page are as follows:

- PDMats.jl : Uniform interface for positive definite matrices of various structures
- Klara.jl : MCMC inference in Julia
- StatsBase.jl : Basic statistics for Julia
- HypothesisTests.jl : Hypothesis tests for Julia
- ConjugatePriors.jl : A Julia package to support conjugate prior distributions
- PGM.jl : A Julia framework for probabilistic graphical models.
- TimeSeries.jl : Time series toolkit for Julia
- StatsModels.jl : Specifying, fitting, and evaluating statistical models in Julia
- Distributions.jl : A Julia package for probability distributions and associated functions

Simple statistics

We will now be exploring the various simple statistical operations that can be performed in Julia, using some default built-in functions:

```
julia> x = [10,20,30,40,50]
5-element Array{Int64,1}:
 10
 20
 30
 40
 50

# computing the mean
julia> mean(x)
```

```
30.0
```

```
# computing the median
```

```
julia> median(x)
```

```
30.0
```

```
# computing the sum
```

```
julia> sum(x)
```

```
150
```

```
# computing the standard deviation
```

```
julia> std(x)
```

```
15.811388300841896
```

```
# computing the variance
```

```
julia> var(x)
```

```
250.0
```

As we can see, given an array of elements, we can easily draw out basic statistical inferences from the data. However, Julia does have functions that support some cumulative operations on the data as well! These are as follows:

- `cummax()` : This function is used to find the cumulative maximum
- `cummin()` : This function is used to find the cumulative minimum
- `cumsum()` : This function is used to find the cumulative maximum
- `cumprod()` : This function is used to find the cumulative product

However, all these are now deprecated and the newer implementation is by using `accumulate()` . Let's see an example showing the usage of both the previous and newer implementations:

```
# old implementations
```

```
julia> cummax(x)
```

```
5-element Array{Int64,1}:
```

```
10
```

```
20
```

```
30
```

```
40
```

```
50
```

```
julia> cummin(x)
```

```
5-element Array{Int64,1}:
```

```
10
```

```
10
```

```
10
10
10
```

```
julia> cumsum(x)
5-element Array{Int64,1}:
 10
 30
 60
100
150
```

```
julia> cumprod(x)
5-element Array{Int64,1}:
 10
 200
 6000
240000
12000000
```

Checking out the new accumulate function, we have the following:

```
# using accumulate function
julia> accumulate(+,x)
5-element Array{Int64,1}:
 10
 30
 60
100
150
```

```
julia> accumulate(*,x)
5-element Array{Int64,1}:
 10
 200
 6000
240000
12000000
```

```
julia> accumulate(max,x)
5-element Array{Int64,1}:
 10
 20
```

```
30
40
50
```

```
julia> accumulate(min,x)
5-element Array{Int64,1}:
10
10
10
10
10
```

Basic statistics using DataFrames

We have studied what DataFrames are and how they can be created in the previous topics. DataFrames can hold extra information about data and hence they prove to be very efficient while describing a data set:

```
julia> dframe = DataFrame(Subjects = ["Maths", "Physics", "Chemistry"], Marks = [90, 85, 95])
3×2 DataFrame
```

Row	subject	Marks
1	Maths	90
2	Physics	85
3	Chemistry	95

```
julia> describe(dframe)
Subjects
Summary Stats:
Length:      3
Type:        String
Number Unique: 3
Number Missing: 0
% Missing:    0.000000
```

```
Marks
Summary Stats:
Mean:      90.000000
Minimum:    85.000000
1st Quartile: 87.500000
Median:     90.000000
3rd Quartile: 92.500000
Maximum:    95.000000
```

```

Length:          3
Type:            Int64
Number Missing:  0
% Missing:       0.000

```

As you see here, we have declared a DataFrame `dframe` , which holds the name of the subjects as well the marks scored by a student out of 100. The `describe` function when runs upon this DataFrame returns all the possible statistical inferences that can be made from the given data. As rightly noted in the official documentation of Julia for this function:

```
describe(df)
```

Pretty-print the summary statistics provided by `summarystats`: the mean, minimum, 25th percentile, median, 75th percentile, and maximum.

Using Pandas

The package `Pandas` has always played a major role in manipulating data. Similar to `DataFrames.jl` , it provides with its own set of data structures. Because of its widely known abilities and robust performance, the `Pandas` package is available for Julia as well.

To install `Pandas` , you need to have Python running on your system and `Pandas` installed as a package. That can be achieved by running the `sudo pip install pandas` command.

Run the following command inside Julia REPL:

```

julia> Pkg.add("Pandas")
julia> using Pandas

```

Once done, we can revisit the same DataFrame we made earlier, but with a little twist. We need to supply the arguments to `pandasDataFrame` in the form of a hashmap:

```
julia> pandasDataFramePandas.DataFrame(Dict{ :Subjects => ["Maths", "Physics", "Chemistry"], :Marks => [90, 85, 95]})
```

```

      Marks  Subjects
0      90      Math
1      85     physics
2      95     Chemistry
julia> Pandas.describe(pandasDataFrame)
      Marks
Count  3.0
Mean   90.0
Std    5.0
Min    85.0
25%    87.5
50%    90.0
75%    92.5
Max    95.0

```

```
julia> query(pandasDataframe, : (Marks>90) )
      Marks  Subject
2      95    Chemistry
```

Advanced statistics topics

So far we have been seeing very simple statistics operations. Now we will discuss some of the advanced statistical operations in brief, including the Julia packages that help in achieving the results:

- Distributions.jl
- Timeseries.jl
- HypothesisTests.jl

We won't be going into much detail about each and every topic from a mathematical point of view, as there are tonnes of resources already available; rather, we will be focusing on the functions provided by these packages.

Distributions

As per the official documentation of distributions.jl , here are the features implemented by this package:

- Moments (for example, mean, variance, skewness, and kurtosis), entropy, and other properties
- Probability density/mass functions (pdf) and their logarithms (logpdf)
- Moment generating functions and characteristic functions
- Maximum likelihood estimation
- Posterior w.r.t. conjugate prior and Maximum-A-Posteriori (MAP) estimation

```
julia> Pkg.add("Distributions")
julia> using Distributions
```

```
# create the random array of 10 element following Normal distribution
julia> x = rand(distribution, 10)
10-element Array{Float64,1}:
 1.19027
 2.04818
 1.14265
 0.459416
-0.396679
-0.664713
 0.980968
-0.0754831
 0.273815
-0.194229
```

Other than the normal distribution, we also have other distributions, which are as follows:

Binomial distribution:

```
julia> Binomial()  
Distributions.Binomial{Float64}(n=1, p=0.5)
```

Cauchy distribution:

```
julia> Cauchy()  
Distributions.Cauchy{Float64}(μ=0.0, σ=1.0)
```

Poisson distribution:

```
julia> Poisson()  
Distributions.Poisson{Float64}(λ=1.0)
```

TimeSeries

TimeSeries are often used in places where there is financial data and every second holds its own importance. In Julia, we have the package by the name TimeSeries.jl, which supports all the timeseries operations.

The most significant data structure provided by TimeSeries is the TimeArray data structures, which help in holding the dates as timestamps and not strings.

For understanding it in detail, let's just also install the TimeSeries package as well as the MarketData package, which will help us study some TimeSeries data:

```
julia> Pkg.add("TimeSeries")  
julia> Pkg.add("MarketData")  
  
julia> using TimeSeries  
julia> dates = collect(Date(2017,8,1):Date(2017,8,5))  
5-element Array{Date,1}:  
2017-08-01  
2017-08-02  
2017-08-03  
2017-08-04  
2017-08-05  
julia> sample_time = TimeArray(dates, rand(length(dates)))  
5x1 TimeSeries.TimeArray{Float64,1,Date,Array{Float64,1}} 2017-08-01  
to 2017-08-05  
  
2017-08-01 | 0.7059  
2017-08-02 | 0.292  
2017-08-03 | 0.2811  
2017-08-04 | 0.7929  
2017-08-05 | 0.2092
```


If we closely look at this TimeArray object `sample_time`, we can see that it has four fields. Out of these four, the `timestamp` field holds the dates array while `values` holds data from the time series, and its row count must match the length of the timestamp array.

The meta defaults to hold nothing while `colnames` contains the names of the columns:

```
julia> fieldnames(sample_time)
4-element Array{Symbol,1}:
:timestamp
:values
:colnames
:meta
```

```
julia> sample_time.timestamp
5-element Array{Date,1}:
2017-08-01
2017-08-02
2017-08-03
2017-08-04
2017-08-05
```

```
julia> sample_time.values
5-element Array{Float64,1}:
0.70586
0.291978
0.281066
0.792931
0.20923
```

```
julia> sample_time.colnames
1-element Array{String,1}:
""
```

```
julia> sample_time.meta
```

```
# retrieve the first element
```

```
julia> head(sample_time)
1x1 TimeSeries.TimeArray{Float64,2,Date,Array{Float64,2}} 2017-08-01
to 2017-08-01
2017-08-01 | 0.7179
```

```
# retrieve the last element
```

```
julia> tail(sample_time)
```

```
1x1 TimeSeries.TimeArray{Float64,2,Date,Array{Float64,2}} 2017-08-05
to 2017-08-05
2017-08-05 | 0.4142
```

Hypothesis testing

A hypothesis test is a testing mechanism that is used to determine whether the sample data had enough evidence that a given condition is true for the entire population of data.

There are two kinds of opposing hypothesis examined for a population (that is, the entire data set,) namely:

- Null hypothesis: This is the statement being tested
- Alternative hypothesis: This is the statement that you will be able to conclude as true

In Julia, the hypothesis testing can be achieved using the HypothesisTests.jl package. The package can be installed and used just like the packages discussed so far in this chapter (the reason being that METADATA.jl has the entry for these packages):

```
julia> Pkg.add("HypothesisTests")
julia> using HypothesisTests
```

Once the package is installed, we can start testing. For the sake of simplicity, I will be discussing a simple example to demonstrate how the package can be used:

```
julia> using Distributions

julia> sampleOne = rand(Normal(), 10)
10-element Array{Float64,1}:
 1.19027
 2.04818
 1.14265
 0.459416
-0.396679
-0.664713
 0.980968
-0.0754831
 0.273815
-0.194229

julia> testOne = OneSampleTTest(sampleOne)
One sample t-test
-----
Population details:
parameter of interest:      Mean
```

```
value under h_0:          0
point estimate:          0.47641935520300993
95% confidence
interval:      (-0.13332094295432084,      1.0861596533603408)
```

Test summary:

```
outcome with 95% confidence:   fail to reject h_0
two-sided p-value:            0.11093746407653728
```

Details:

```
number of observations:      10
T-statistic:                1.7675319478229796
degrees of freedom:         9
empirical standard error:    0.2695393176852065
```

Optimization

In mathematics and computer science, an optimization problem is a problem of finding the best solution from all the feasible solutions. They can broadly be divided into two categories depending upon the variables:

- Continuous (continuous optimization problem)
- Discrete (combinatorial optimization problem)

Some of the problems that can be categorized as optimization problems are given here:

- Shortest path
- Maximum flow through a network
- Vehicle routing

Julia, in particular, provides a number of optimization packages, the group of which is collectively called as JuliaOpt. The two most notable packages used are:

- JuMP (Julia for Mathematical Programming)
- Convex.jl

JuMP

JuMP is an AML implemented in Julia. Readers coming from a Python background may be familiar with PuLP. It currently supports several open source solvers for a wide variety of problem cases. Some of its features include:

- User friendliness
- Speed
- Solver independence
- Access to advanced algorithmic techniques
- Ease of embedding

Installing and getting started with JuMP is easy. However, we also need to install a solver alongside, which in this case will be Clp :

```
julia> Pkg.add("JuMP")
julia> Pkg.add("Clp")

julia> using JuMP
julia> using Clp

# creating a model without a solver
julia> m = JuMP.Model()
Feasibility problem with:
* 0 linear constraints
* 0 variables
Solver is default solver

# but we want to use Clp as solver, we have
julia> m = JuMP.Model(solver = Clp.ClpSolver())
Feasibility problem with:
* 0 linear constraints
* 0 variables
Solver is ClpMathProg
```

Moving on, let's now have a look at a full-fledged example. Let's save the underlying code snippet in a file named `optimiser.jl` :

```
using JuMP
using Clp
m = Model(solver = ClpSolver())
@variable(m, 0 <= a <= 2 )
@variable(m, 0 <= b <= 10 )

@objective(m, Max, 5a + 3*b )
@constraint(m, 1a + 5b <= 3.0 )

print(m)

status = solve(m)

println("Objective value: ", getobjectivevalue(m))
println("a = ", getvalue(a))
println("b = ", getvalue(b))
```

On trying to run this file from the shell, we have the following output:

```
mymachine:user$ julia optimiser.jl
```

```
Max 5 a + 3 b
```

```
Subject to
```

```
a + 5 b ≤ 3
```

```
0 ≤ a ≤ 2
```

```
0 ≤ b ≤ 10
```

```
Objective value: 10.6
```

```
a = 2.0
```

```
b = 0.2
```

Julia Programming

Module 8 - Data Visualization and Graphics

Learning Objectives:

1. How Julia provides a set of very advanced and feature-rich libraries to accomplish the task of effectively doing data visualization.
2. How PyPlots work and how we can make simple graphical plotting using the package.
3. Detail about the two most popular libraries in Julia, namely `Vega.jl` and `Gadfly.jl`.

Basic plots

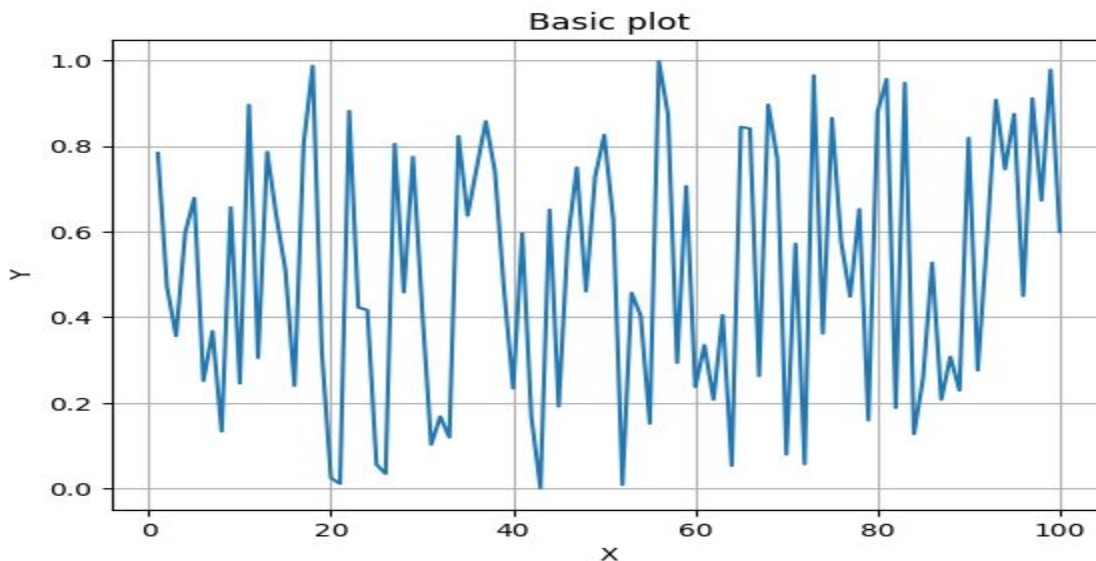
We will be focusing on creating some of the very simple and easily used graphs and plots. The library of choice which we will be using to create such plots will be PyPlot.

The installation of the library is easy if you have matplotlib already installed on your system. If not, you need to install it manually by running the following command:

```
python -m pip install matplotlib
```

Once this is done, we will open the Julia REPL and run `Pkg.add("PyPlot")`. PyPlot is a large and rich library.

```
x = 1:100
y = rand(100)
p = plot(x,y)
xlabel("X")
ylabel("Y")
title("Basic plot")
grid("on")
```

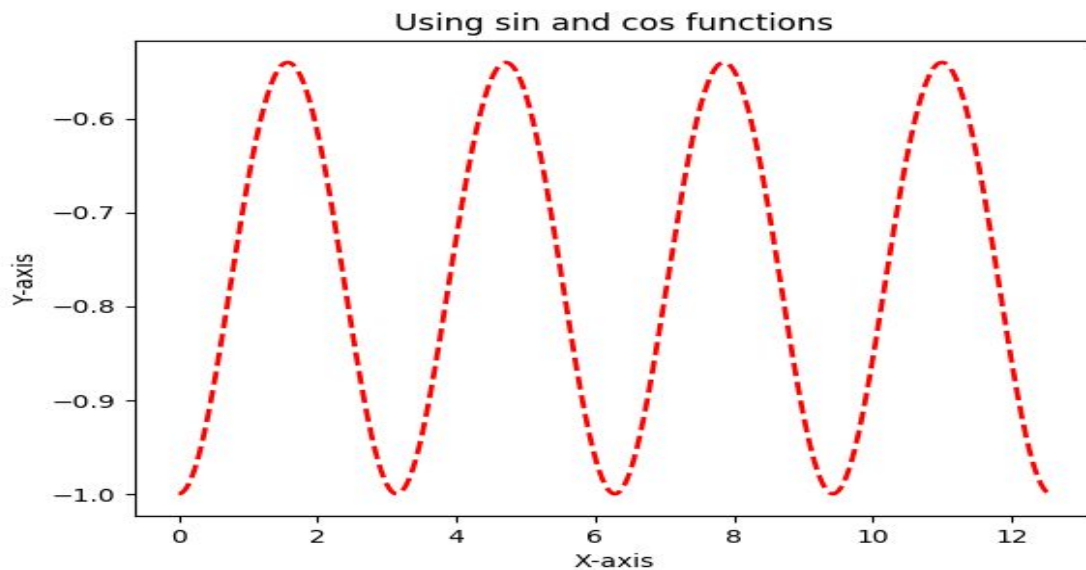


Let's explain how this works quickly:

- We have an x-coordinate, which is simply a range of numbers from 1 to 100 as given in the code. The label for this axis has been set to X using the function `xlabel`.
- Similarly, we have the y-coordinate, which is a randomly generated range of 100 numbers. The label for y-axis is set to Y by using a `ylabel` function.
- Our graph has been labeled, but we still don't have any title for it. This can be achieved using the function `title`, which takes in a string value.
- Finally, the `plot` function is the one that actually generates the graph.

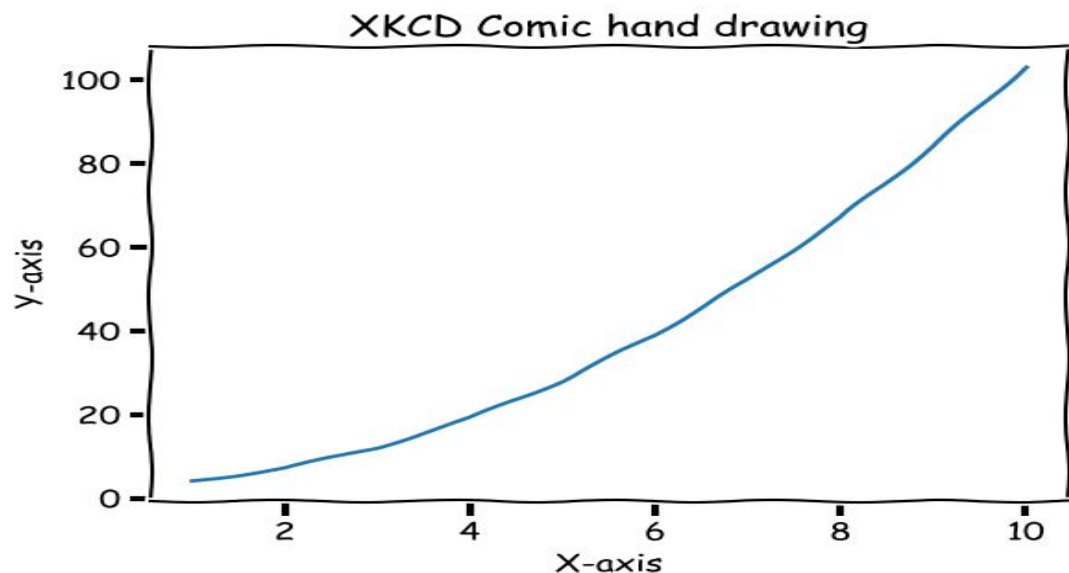
Here is yet another line graph, which makes use of the sin and cos functions:

```
using PyPlot
x = linspace(0, 4pi, 1000); xlabel("X-axis")
y = cos(pi + sin.(x)); ylabel("Y-axis")
plot(x, y, color="red", linewidth=2.0, linestyle="--");
title("Using sin and cos functions");
```



One final example is an XKCD graph, which is just a casual-style, handwritten graph mode:

```
x = [1:1:10;]; y = ones(10)
for i=1:1:10 y[i] = pi + i*i end
xkcd(); xlabel("X-axis"); ylabel("Y-axis")
title("XKCD Comic hand drawing"); plot(x,y)
```

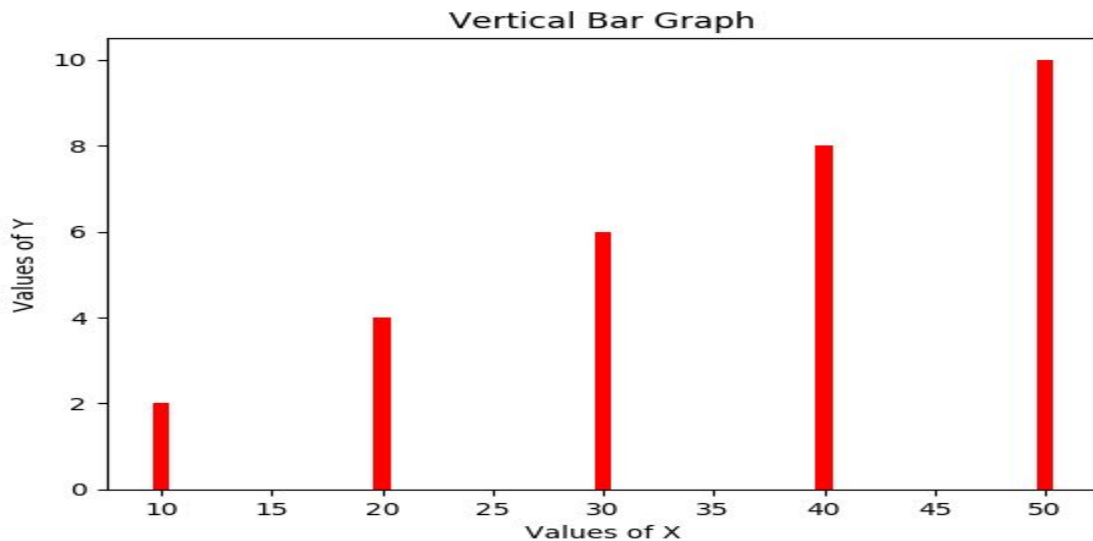


We will now see how various different types of plots can be drawn in Julia using PyPlot.jl.

Bar graphs

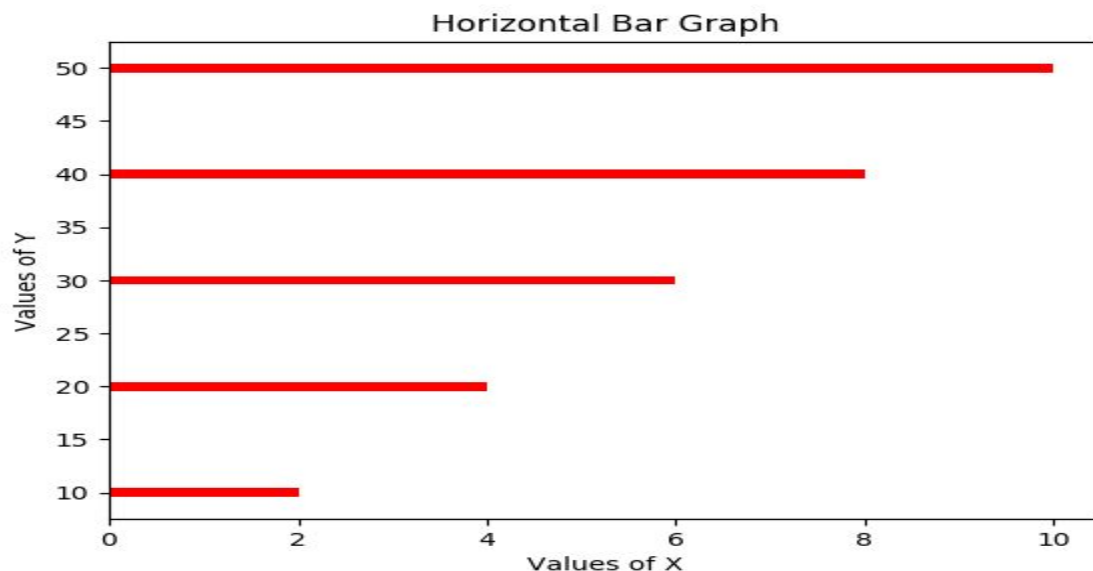
A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars, with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. We will now see how to create a bar graph in Julia using the very simple `bar` function:

```
x = [10,20,30,40,50];y = [2,4,6,8,10]
xlabel("Values of X");ylabel("Values of Y")
title("Vertical Bar Graph")
bar(x,y,color = "red")
```



This is a simple vertical bar graph. What if we wanted to have a horizontal bar graph in its place? For this we have the `barh` function, which does exactly that:

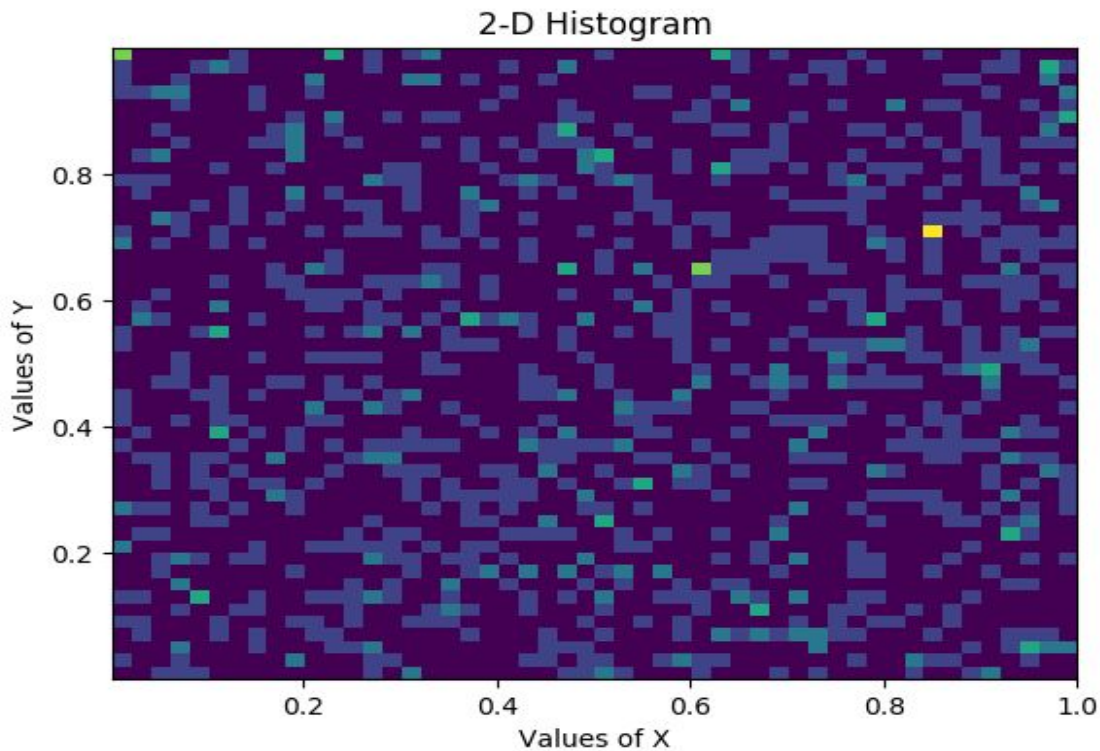
```
x = [10,20,30,40,50];y = [2,4,6,8,10]
xlabel("Values of X");ylabel("Values of Y")
title("Horizontal Bar Graph")
barh(x,y,color = "red")
```



Histograms

PyPlot provides a function named `hist2D`, which helps to create a histogram. It takes in three parameters, which are generally x-axis parameters, y-axis parameters, and bins. The following figure shows how `hist2D` creates a graphical representation of the data:

```
x = rand(1000);xlabel("Values of X");  
y = rand(1000);ylabel("Values of Y");  
title("2-D Histogram")  
hist2D(x,y,bins = 50)
```



Pie charts

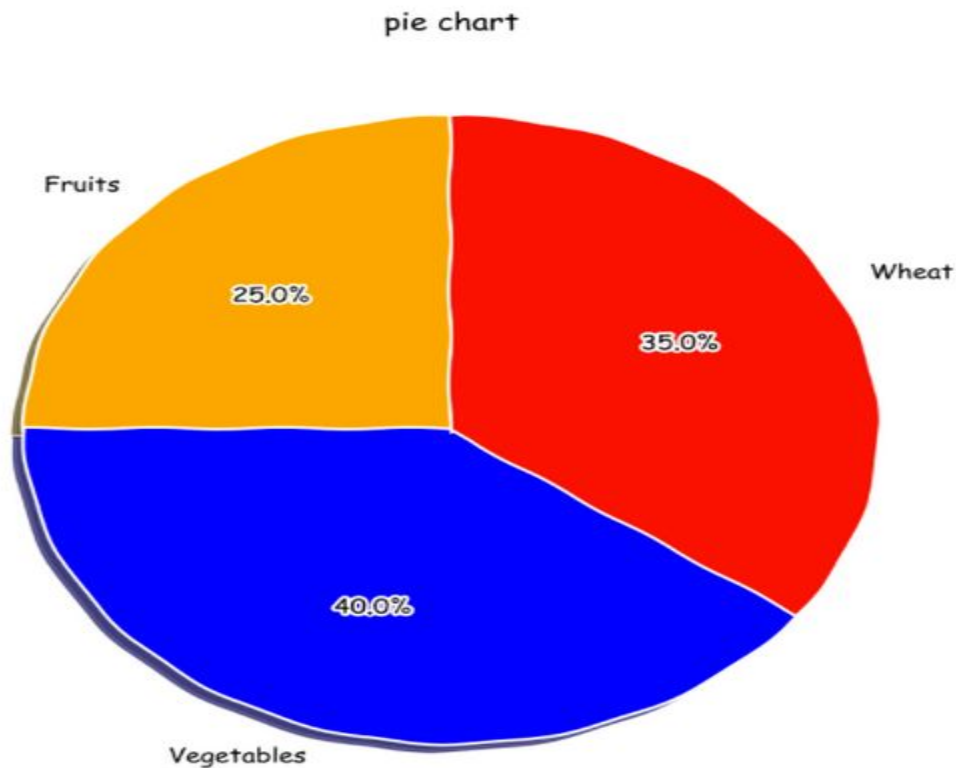
A pie chart is a graphical representation of data that displays all possible numerical proportions. The length of the arc in pie charts is directly proportional to the quantity it represents.

PyPlot, has a function named `pie`, which helps to do exactly the same. Given here is a simple diet layout (fictional) for a person who eats just three things:

```

labels = ["Fruits"; "Vegetables"; "Wheat"]
colors = ["orange"; "blue"; "red"]
sizes = [25; 40; 35]
explode = zeros(length(sizes))
fig = figure("pyplot_piechart", figsize=(10,10))
p = pie(sizes, labels=labels, shadow=true,
        startangle=90, colors=colors,
        autopct="%1.1f%%")
title("pie chart")

```



Scatter plots

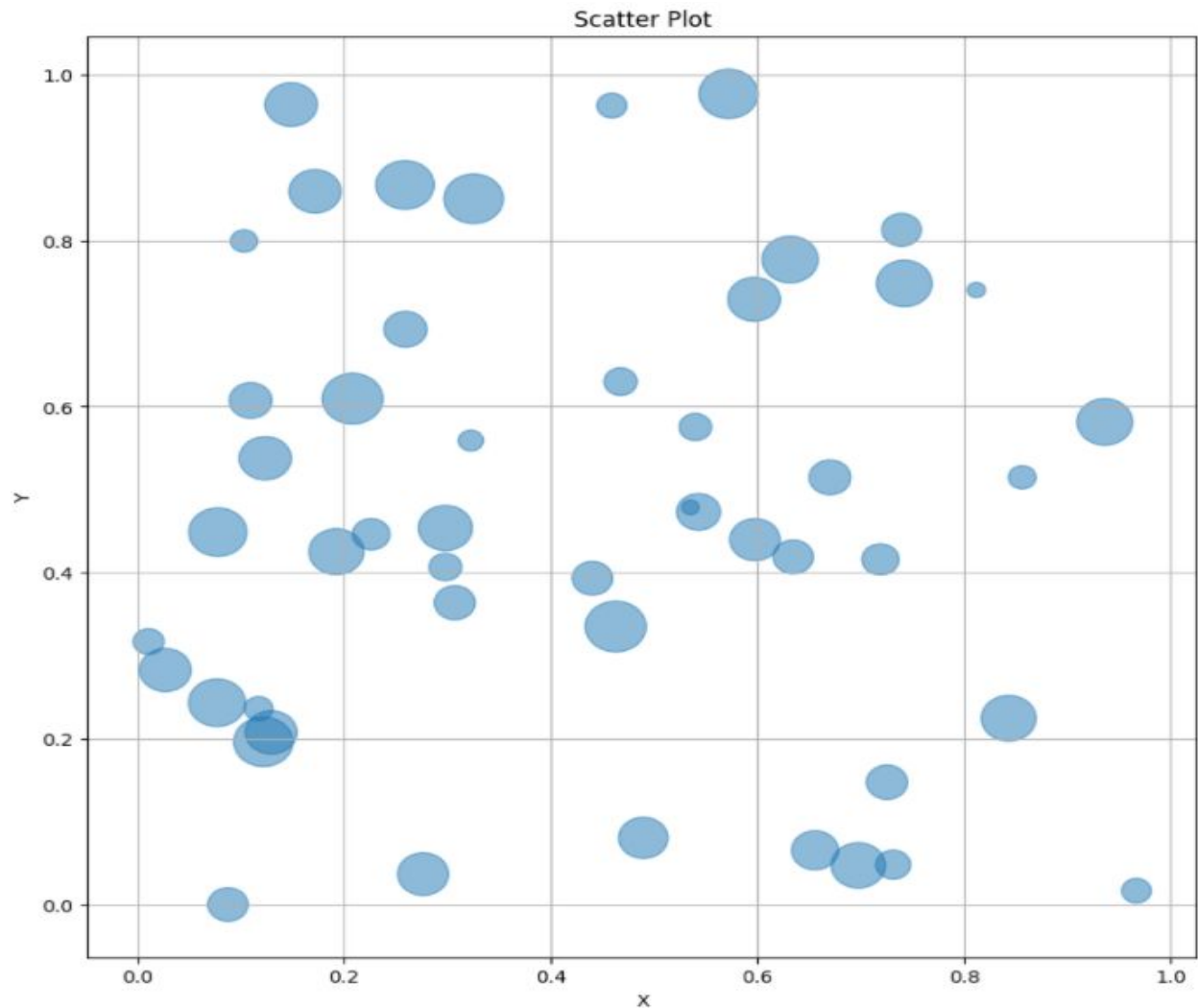
Scatter plots help us to identify the correlation between the two variables. Or, in other words, they help us understand the relationships between data.

Here is an example of a scatter plot implemented using the PyPlot library:

```

using PyPlot
fig = figure("scatterplot", figsize=(10,10))
x = rand(50)
y = rand(50)
areas = 1000*rand(50);
scatter(x,y,s=areas,alpha=0.5)
title("Scatter Plot")
xlabel("X")
ylabel("Y")
grid("on")

```



Vega

Vega is a data visualization library that provides a Julia wrapper around the Vega Visualization Grammar from Trifacta. It was created and developed by **Randy Zwitch** as an open source project.

Installing Vega is pretty simple to do, as listed in the `METADATA.jl`:

```
julia> Pkg.add("Vega")
julia> using Vega
```

However, Vega requires an internet connection to render all graphs because it does not store local copies of the JavaScript libraries. However, efforts are made to actually include all the JavaScript libraries as part of the package so that it does not require any internet connection in the future.

Vega provides a series of composite types, which are literally just the translation of Vega grammar in Julia. They help the end user to build visualizations in native Julia syntax. The following are all the primitives defined in Julia:

- VegaVisualization
- VegaData
- VegaScale
- VegaAxis
- VegaLegend
- VegaMark

Apart from these visualizations, Vega also provides interactivity to the graphs by providing some very interesting functions collectively called, **Visualization Mutating Functions**. They all are listed as follows:

- `colorscheme!`: This is used to change or alter the color scheme of the graph. Here is an example:

```
colorscheme!(v::Vega.VegaVisualization; palette,reversePalette)
```

- General visualization properties:
 - `height` `height::Int`
 - `width` `width::Int`
 - `background` `background::AbstractString` , as of Julia 0.5
 - `padding` `padding::Union{VegaPadding, Number, String}`
 - `viewport` `viewport::Vector{Int}`
- `hline! / vline!` : These functions help in drawing horizontal or vertical lines on the graph. Their usage syntaxes are `hline!(v::Vega.VegaVisualization; value, strokeDash, strokeWidth, stroke)` and `vline!(v::Vega.VegaVisualization; value, strokeDash, strokeWidth, stroke)`.
- `hover!` : Those familiar with JavaScript or jQuery must know the use for hover. It basically alters the property of the element inside a graph if hovered over using a mouse:

```
hover!(v::Vega.VegaVisualization; opacity, color)
```

Here, opacity can range from 0 to 1, while colors can be set to any hex code or as names of the original RGB colors, which the graph will change to once hovered over.

- `jitter!` : This is used as `jitter!(v::Vega.VegaVisualization; pctXrange, pctYrange)`. Jitter adds a random amount of noise to data as a means of remedying overplotting. The `pctXrange` / `pctYrange` arguments are the percentages of the data series range allowable (+/-) to add to the data.
- `legend!` : This is used as `legend!(v::Vega.VegaVisualization; title, show)`. It helps in changing or altering the visualization's title.

- `stroke!:` This is used as `stroke!(v::Vega.VegaVisualization; color, width, opacity, visible)`. This function helps by adding borders around VegaMarks and changes their visualization.
- `text!:` This is used as `text!(v::Vega.VegaVisualization; title, y, fill, fontSize, align, baseline, fontWeight, font, x)`. This function changes the visualization by adding text annotations. It's just an alias of the `title!` with some default values.
- `title!:` This is used as `title!(v::Vega.VegaVisualization; title, y, fill, fontSize, align, baseline, fontWeight, font, x)`. As the name suggests, it modifies the title of the graph.
- `xlab!/ylab! :` This is used as `xlab!(v::Vega.Vega Visualization; title,grid, ticks, format, formatType, layer, properties, tickSize, tickSizeMajor, tickSizeMinor, tickSizeEnd)` or `ylab!(v::Vega.VegaVisualization; title, grid, ticks, format, formatType, layer, properties, tickSize, tickSizeMajor, tickSizeMinor, tickSizeEnd)`. It just helps in modifying the x-axis/y-axis.
- `xlim!/ylim!:` This is used as `xlim!(v::Vega.VegaVisualization; min, max, reverse, round, _type, exponent)` or `ylim!(v::Vega.VegaVisualization; min, max, reverse, round, _type, exponent)`.

Area plots

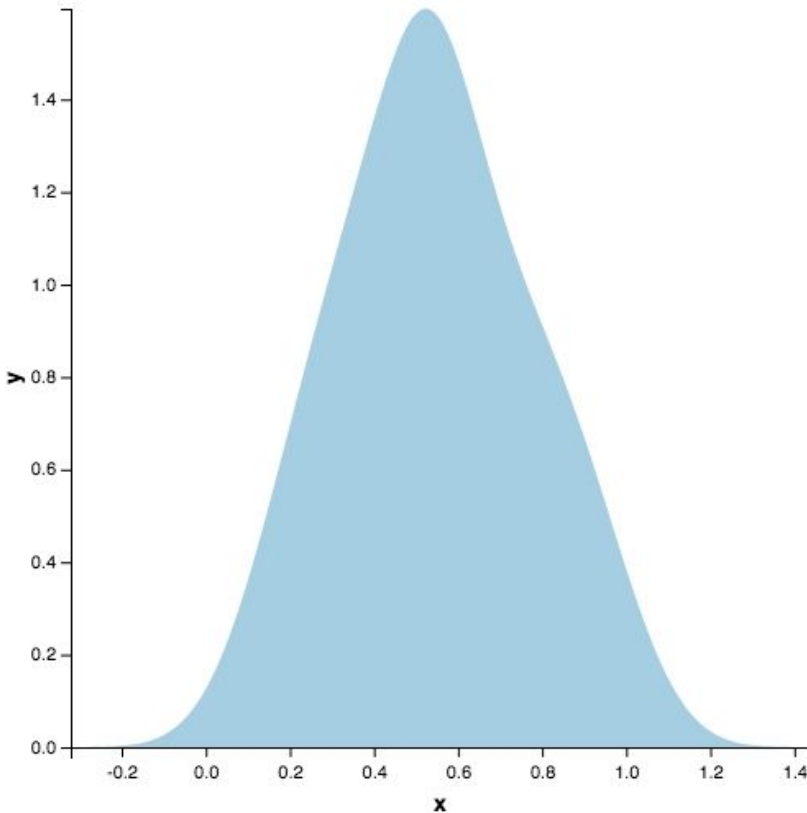
An area chart or area graph displays graphically quantitative data. It is based on a line chart. The area between an axis and a line are commonly emphasized with colors, textures, and hatchings. Commonly, we compare two or more quantities with an area chart.

Vega provides a function named `areaplot`. The following is a sample code depicting the usage of `areaplot`:

```
using Vega, KernelDensity, Distributions

x = rand(Beta(3.0, 2.0), 10)
k = kde(x)

areaplot(x = k.x, y = k.density)
```



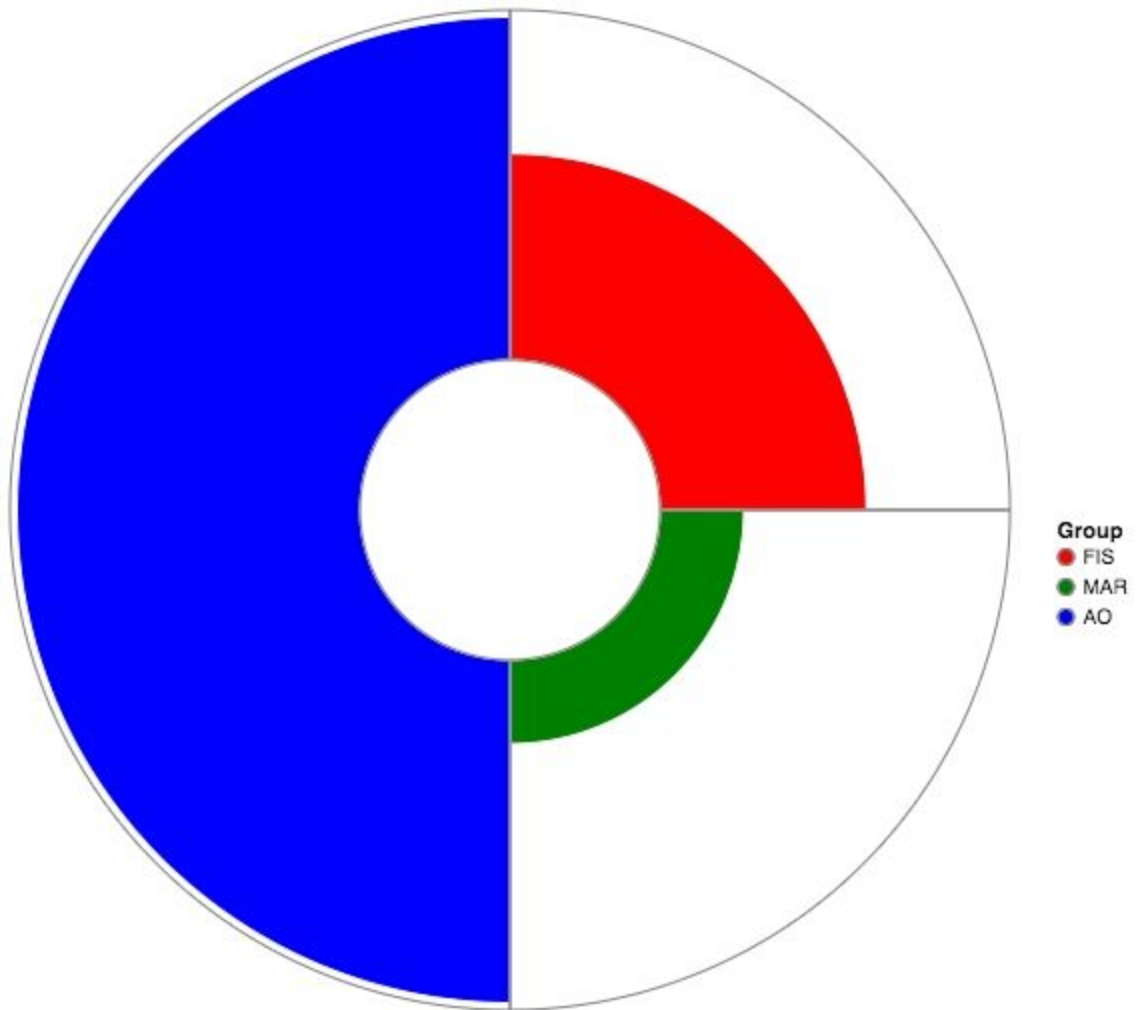
Aster plots

An aster plot displays pie slices as lengths extending outward to the edge (0 at the inner edge to 100 at the outer). The widths of the pie slices represent the weight of each pie, which is used to arrive at a weighted mean of the length scores in the center.

Aster plots in `Vega` are recreations of `d3.js` aster plots. Apart from producing simple plots, you can actually add a bit of interactivity, too, using functions like `hover!` :

```
score = [59, 24, 98]
id = ["FIS", "MAR", "AO"]
weight = [0.5, 0.5, 1]

v = asterplot(x = id, y = score, weight = weight, holesize =
75)
colorscheme!(v, palette = ["red", "green", "blue"])
hover!(v, opacity = 0.5)
```

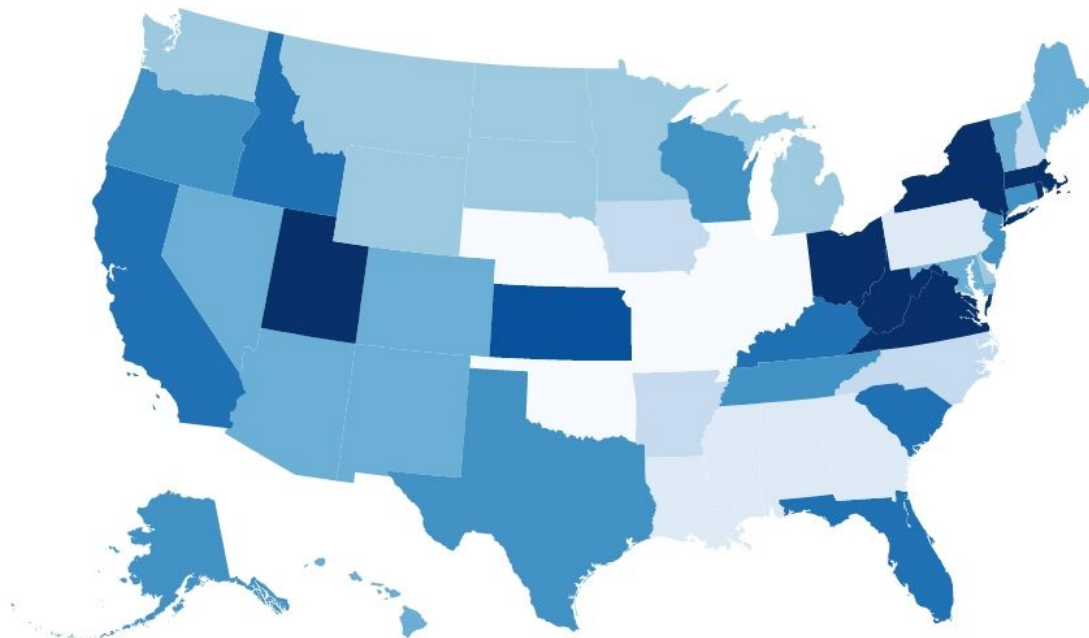


Choropleth map

A choropleth map is a thematic map. It provides an easy way to visualize how a measurement varies across a geographic area, or it shows the level of variability within a region.

The following is a simple choropleth map of the US:

```
x = 1:60
y = rand(Float64, 60)
a = choropleth(x = x, y = y, entity = :usstates)
```

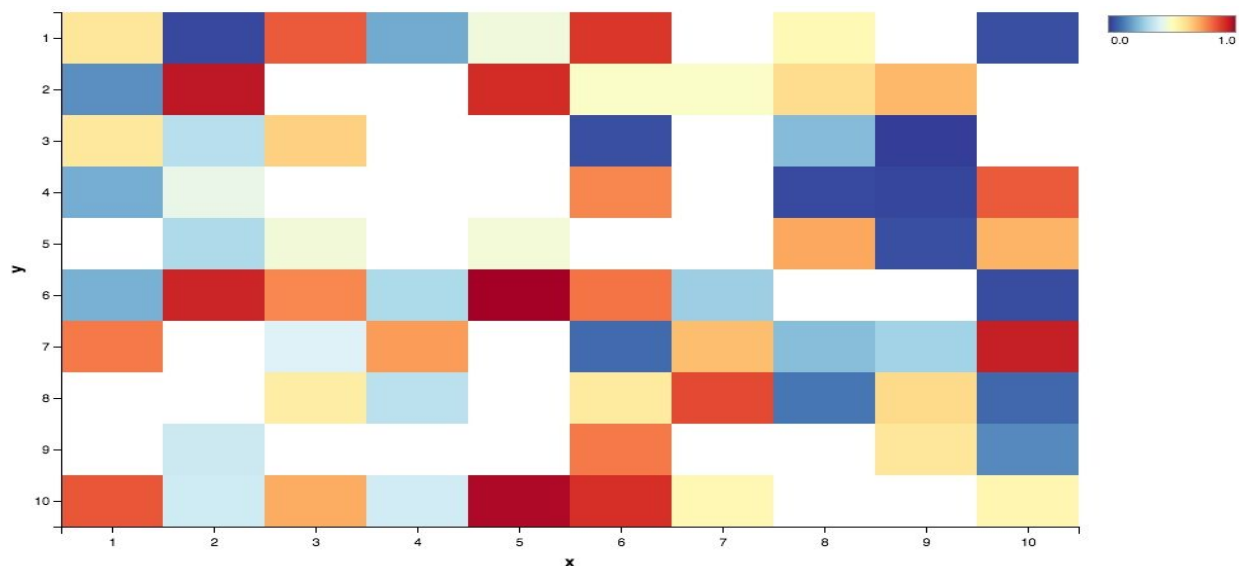
Heatmaps

A heat map (or heatmap) is a graphical representation of data where the individual values contained in a matrix are represented as colors. In Vega, we have a function named `heatmap` that does exactly this.

Here is a sample heatmap in action:

```
heatmap(x = rand(1:10,100), y = rand(1:10,100), color=rand(100))
```

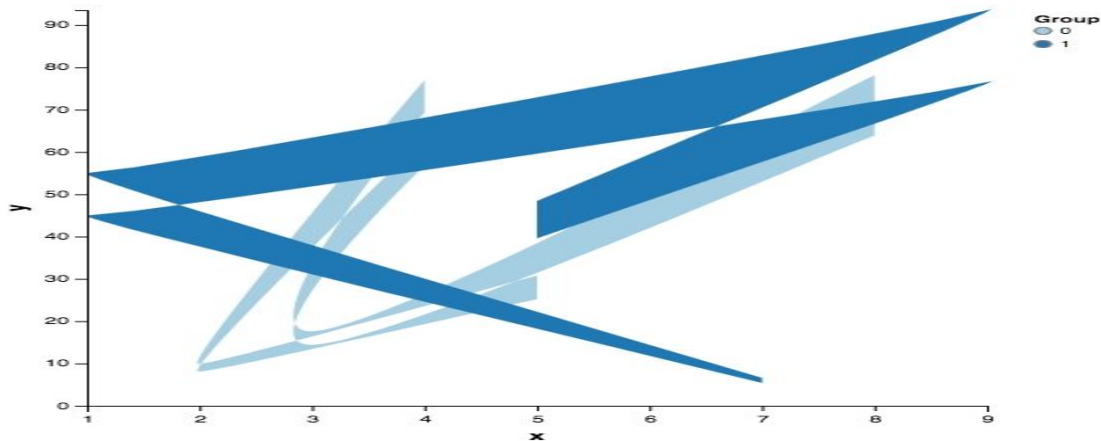
Here, `x` and `y` are in the range of 100 random numbers between 1 and 10, while the color is an array of 100 random color boxes:



Ribbon plots

Ribbon plots are made by drawing two path plots of N points each, and connecting the two paths by a sheet made of N triangles. Just to add, a ribbon does not have to be of a constant width. In Vega, we have a function named `ribbonplot`, which helps us create ribbon plots. A simple example of this is illustrated as follows:

```
X = [0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9]
Y = rand(1:100,20)
G = [0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1]
A = ribbonplot(x = x, ylow = 0.9y, yhigh=1.1y, group = g)
```



Scatter plots

We have previously seen what scatter plots are when we were working with the `PyPlot` library. Here too, we will see how to create a scatter plot in Vega using a function named `scatterplot`.

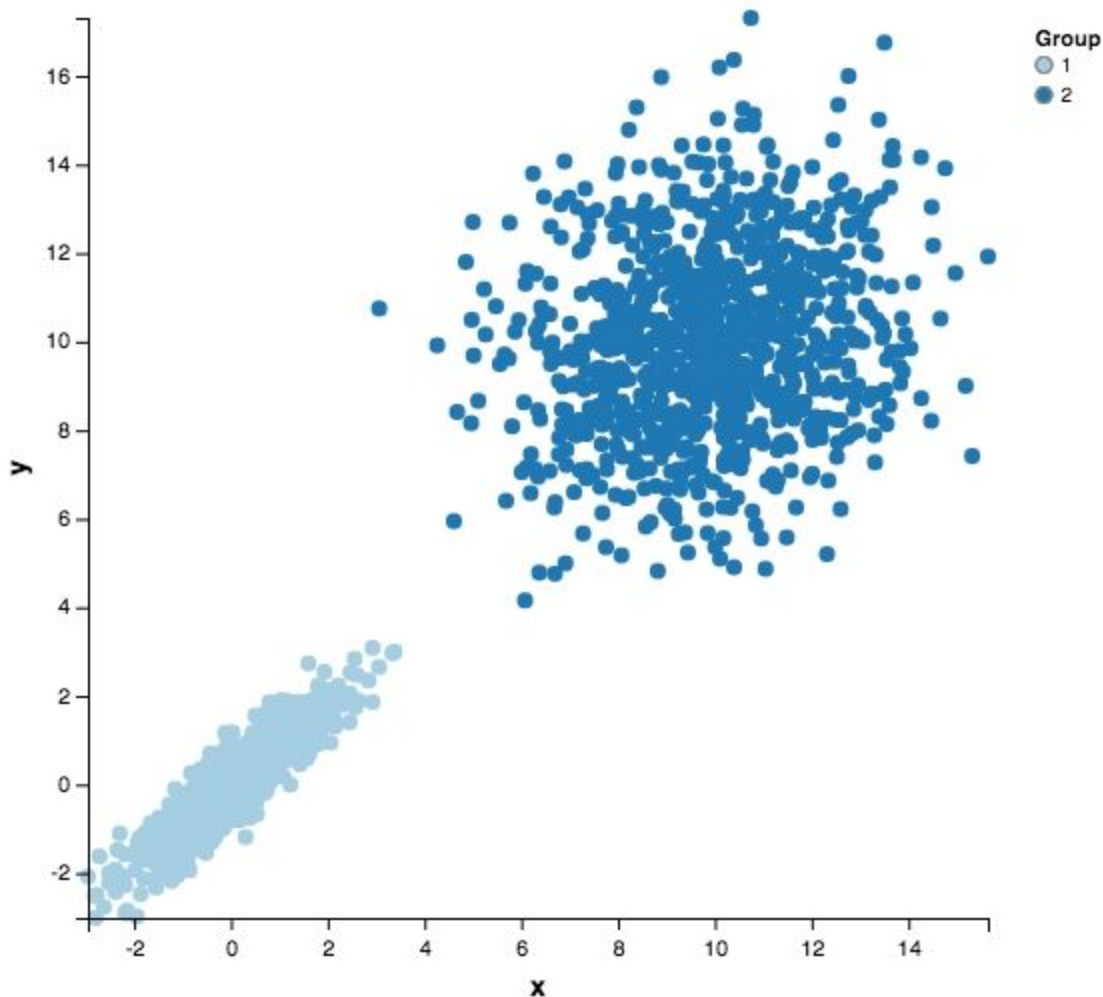
Here is a simple example of a scatter plot:

```
using Vega, Distributions

d1 = MultivariateNormal([0.0, 0.0], [1.0 0.9; 0.9 1.0])
d2 = MultivariateNormal([10.0, 10.0], [4.0 0.5; 0.5 4.0])
points = vcat(rand(d1, 1000)', rand(d2, 1000)')

x = points[:, 1]
y = points[:, 2]
group = vcat(ones(Int, 1000), ones(Int, 1000) + 1)

scatterplot(x = x, y = y, group = group)
```



Gadfly

Gadfly is an exhaustive plotting and data visualization package written in Julia by Daniel Jones. It is based on the book, *The Grammar of Graphics*, by Leland Wilkinson. It is largely inspired by `ggplot2` for R, which is another amazing package for plotting and visualizations.

Some of the great features of Gadfly that set it apart from all other libraries are listed here:

- It renders publication quality graphics to SVG, PNG, PostScript, and PDF
- It has an intuitive and consistent plotting interface
- It works with **IJulia** out of the box
- It offers tight integration with `DataFrames.jl`
- It provides interactivity, like panning, zooming, and toggling, powered by `snap.svg`
- It supports a large number of common plot types

Installing Gadfly is pretty easy and similar to Vega . The entry for this package can be found in `METADATA.jl`. There are, however, some dependencies, which will also be automatically installed by the package itself:

```
julia> Pkg.add("Gadfly")  
julia> using Gadfly
```

Interacting with Gadfly using the plot function

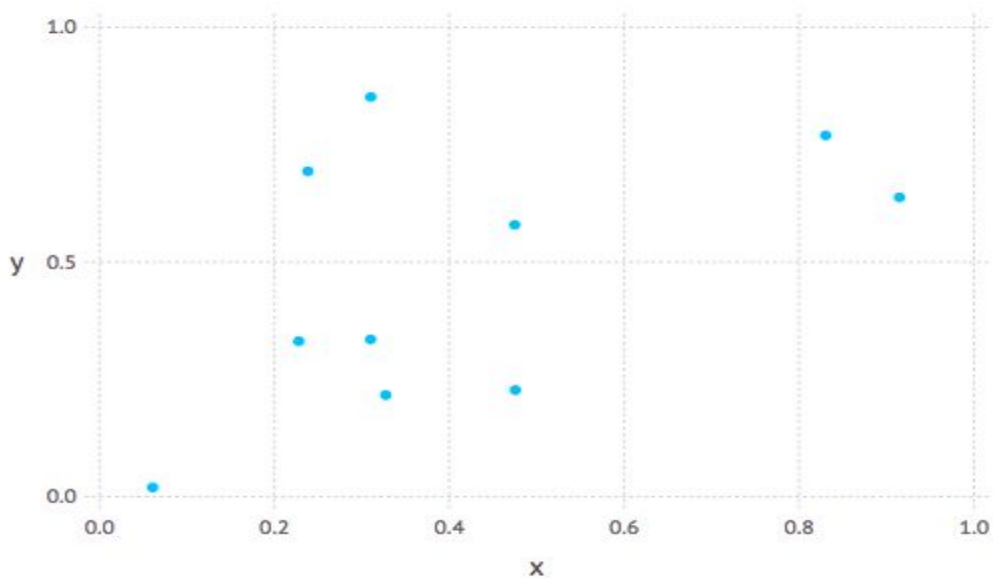
The `plot` function is used to interact with the `Gadfly` package and create the desired visualizations. Aesthetics are mapped to the plot geometry and are used to specify how the `plot` function would work. They are specially named variables. The `plot` elements can be scales, coordinates, guides, and geometries. It is defined in the grammar of graphics to avoid special cases, and aesthetics help this by approaching the problem with well-defined inputs and outputs, which produces desired results.

A plot can operate on the following data sources:

- Functions and expressions
- Arrays and collections
- DataFrames

Here is a simple example of a graph made using the `plot` function. We will be using `IJulia` to draw all our plots:

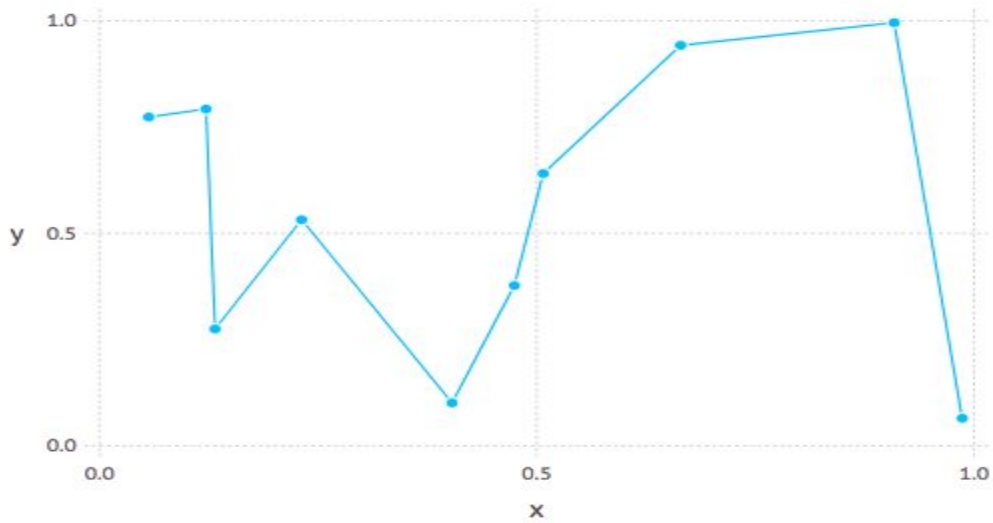
```
Gadfly.plot(x=rand(10), y=rand(10))
```



We can also add more elements to have a slightly different output. For example, to have both line and point geometries on the same dataset, we can make a layered plot using the following:

- `Geom.line`: Line plot
- `Geom.point`: Point plot

```
Gadfly.plot(x=rand(10), y=rand(10), Geom.point, Geom.line)
```

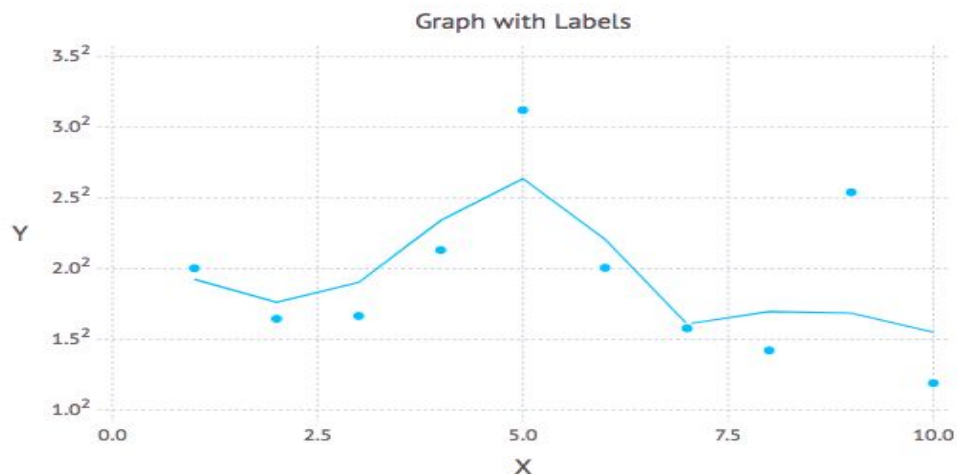


This generates a layered plot that has both lines and points. A complex plot can be generated by combining various elements:

1. Scale: Use this to scale any desired axis of the plot up or down.
2. Guide: Using xlabel and ylabel, guide can be used to give the necessary labels to the plot that we use. Title is used to provide a title for the plot.

Let's create a similar plot that includes these elements. We will add the x and y labels, add a title to the plot, and scale the plot:

```
Gadfly.plot(x=1:10, y=10.^rand(10),
  Scale.y_sqrt, Geom.point, Geom.smooth,
  Guide.xlabel("X"), Guide.ylabel("Y"), Guide.title("Graph with Labels"))
```



Plotting DataFrames with Gadfly

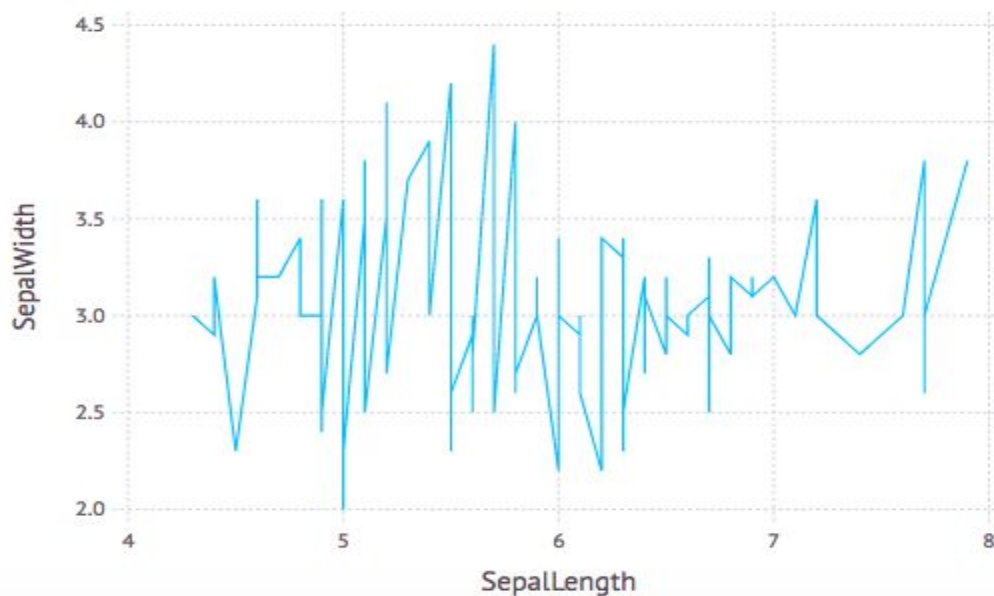
It is a powerful data structure used to represent and manipulate data. Using `Gadfly`, we can generate complex plots easily. `DataFrame` is passed to the `plot` function as the first argument.

The columns in the `DataFrame` are used by the `plot` function in the aesthetics by name or index. We will use `RDatasets` to create the `DataFrame` for the `plot` function. To install `RDatasets`, just follow the underlying method similar to how we did it in `Gadfly`:

```
julia> Pkg.add("RDatasets")  
julia> using RDatasets
```

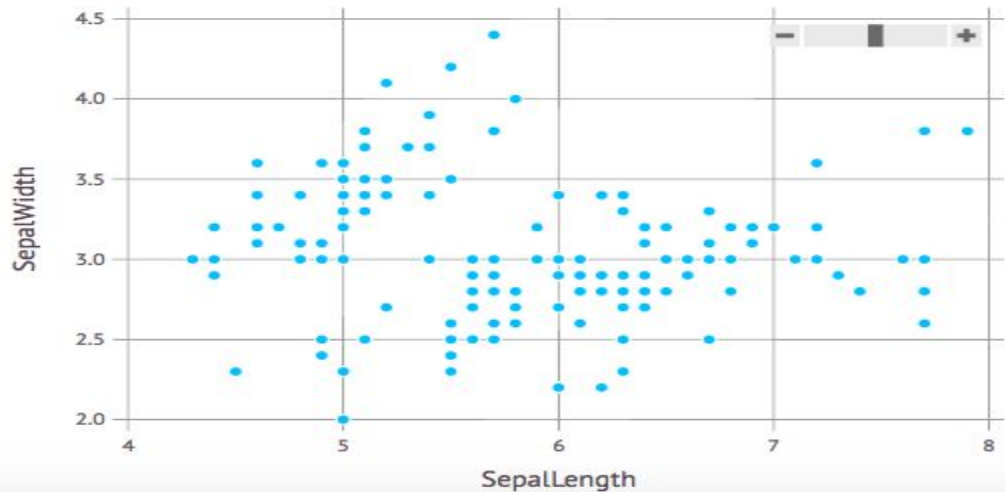
The `RDatasets` provides us with some real-life datasets, from which we can make some visualizations to understand the capabilities of the `Gadfly` package:

```
Gadfly.plot(dataset("datasets", "iris"),  
            x="SepalLength",  
            y="SepalWidth",  
            Geom.line)
```



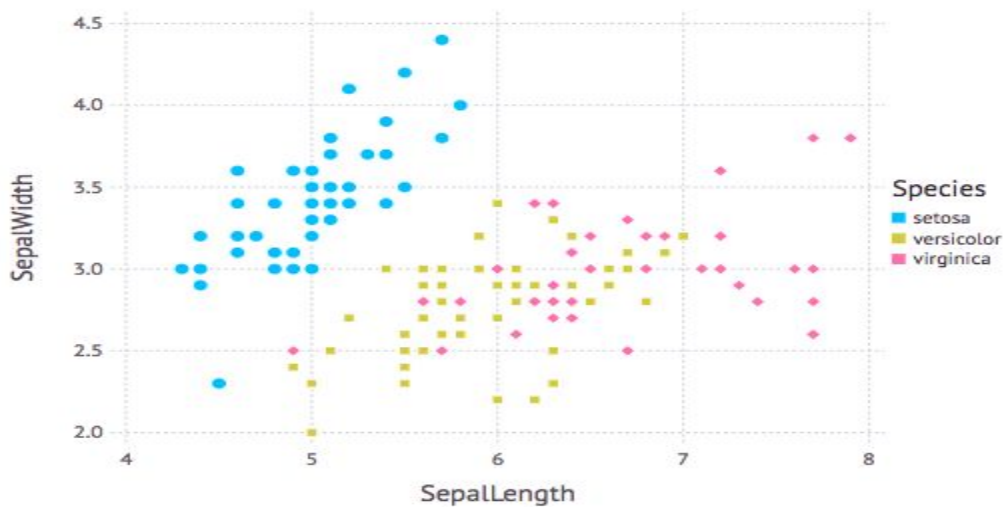
Now, here is the same plot, using the point plotting:

```
using Gadfly, RDatasets
Gadfly.plot(dataset("datasets", "iris"),
  x="SepalLength",
  y="SepalWidth",
  Geom.point)
```



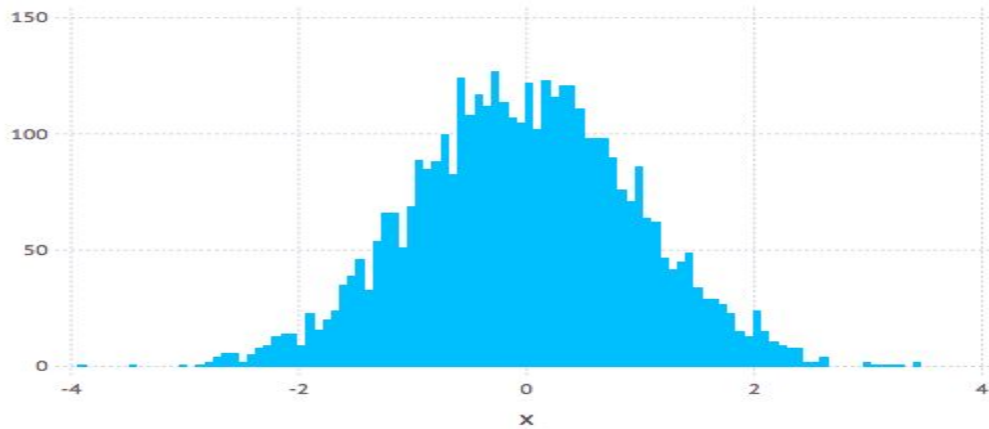
The following is a point plot that uses the Iris dataset provided by the RDatasets package. We are trying to plot a graph between SepalLength and SepalWidth :

```
using RDatasets, Gadfly
Gadfly.plot(dataset("datasets", "iris"),
  x=:SepalLength,
  y=:SepalWidth,
  color=:Species, shape=:Species, Geom.point,
  Theme(point_size=3pt))
```



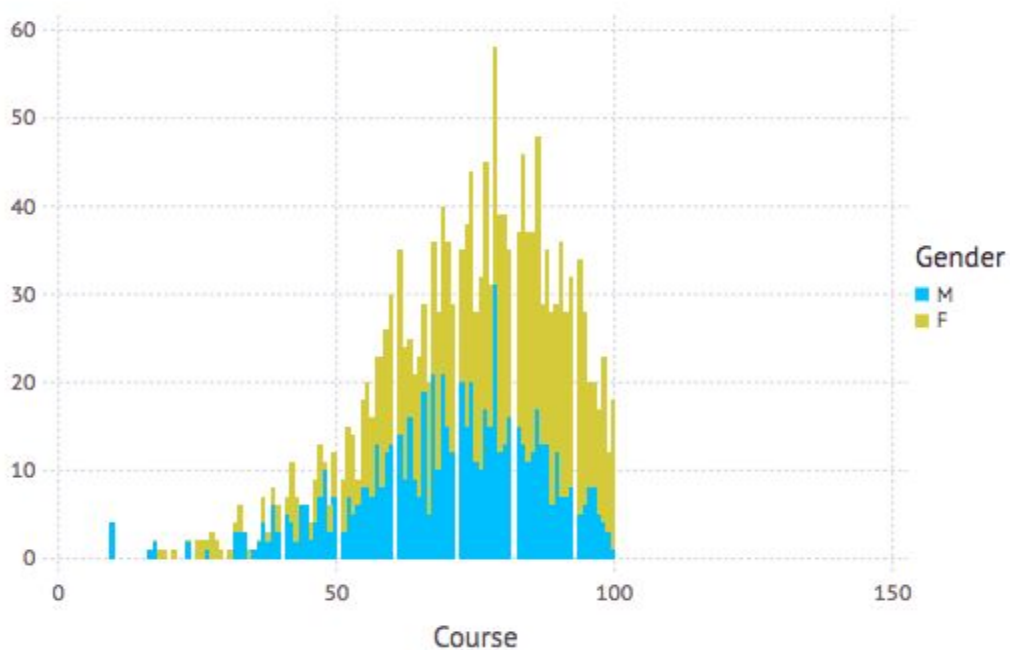
Now, let's create a histogram using a random number generator. We will pass the array, which we will create using a random number generator, and then we will create the histogram:

```
using Gadfly
Gadfly.plot(x = randn(4000), Geom.histogram(bincount = 100))
```



However, the preceding showcased histogram was a fairly simple example. We now move on to a complex example wherein we will be using a dataset provided by the RDatasets Package:

```
using Gadfly, RDatasets
Gadfly.plot(dataset("mlmRev", "Gcsemv"),
  x = "Course", color="Gender", Geom.histogram)
```



Julia Programming

Module 9 - Connecting with Databases

Learning Objective:

1. How to connect with databases?
2. Working with Relational databases and NoSQL databases.
3. Introduction to REST.

How to connect with databases?

Connecting with databases requires us to have a middle layer (that is, a database driver) that can help us establish a connection between a database and an application system. The choice of driver will depend on these two underlying factors:

- Language (in which the application is coded)
- Database

A database connection is the means by which a database server and its client software communicate with each other. This interaction usually takes place using a query language such as SQL, wherein the user writes his commands in the form of a query and the database fetches the result based upon that. However, the interaction can only take place once a database connection has been established. This is achieved by using a database string, which is the standard way of connecting using a driver's connection API.

We have the following two broader kinds of databases available for us to make use of:

- Relational DBMS
- Non-Relational DBMS

Given these two categories, the community of Julia developers has come up with a group of Database drivers that are helpful in connecting with some of the most popular databases in the world.

- ODBC.jl : This is an ODBC interface for the Julia programming language
- SQLite.jl : This is a Julia interface to the SQLite library
- MySQL.jl : This is used to access MySQL from Julia
- PostgreSQL.jl : PostgreSQL DBI driver
- JDBC.jl : Julia interface to Java database drivers
- Hive.jl : Hive, Spark SQL, Impala client. Based on Thrift and HiveServer2 protocol
- DBDSQLite.jl : DBI-compliant driver for SQLite3

Relational databases

A relational database is a kind of database that is based on the relational model of data, that is, the kind of model that organizes data into tables of rows and columns along with a unique key identifying each row. To access the data organized in such a format, we make use of Structured Query Language(SQL) that enables us to fetch data from a database in a fairly easy manner.

SQLite

SQLite is a lightweight database majorly used for study and testing purposes. You may install the database from [https:// www. sqlite. org/ download. html](https://www.sqlite.org/download.html) . Once you are done with the database installation, it's time we now move on to installing SQLite's Julia database driver. We will be making use of the package named SQLite, and since its entry is made available inside the METADATA.jl , you will be installing it easily using this: `julia> Pkg.add("SQLite")`

MySQL

MySQL is one of the most popular databases and it's currently supported and maintained by Oracle.

Although it's free and open source, it has an enterprise version too.

To install MySQL on any of the operating systems, visit [https:// www. mysql. com/downloads/](https://www.mysql.com/downloads/).

Now, we move on to installing MySQL's database driver for Julia:

```
julia> Pkg.add("MySQL")
julia> using MySQL
```

similar to SQLite, the installation is fairly simple. Let's try out creating a database connection and explore more with it:

```
using MySQL
con = mysql_connect("localhost", "username", "password",
"db_name")
command = """CREATE TABLE Employee
(
ID INT NOT NULL AUTO_INCREMENT,
Name VARCHAR(255),
Salary FLOAT,
JoinDate DATE,
PRIMARY KEY (ID)
);"""
mysql_execute(con, command)
Insert some values
mysql_execute(con, "INSERT INTO Employee (Name, Salary,
JoinDate) values
('Alpha', 25000.00, '2015-12-12'), ('Beta', 35000.00,
'2012-18-17),
('Gamma', 50000.00, '2013-12-14');")
# Get SELECT results
command = "SELECT * FROM Employee;"
dframe = mysql_execute(con, command)
# Close connection
mysql_disconnect(con)
```

NoSQL databases

A NoSQL database is a kind of database that doesn't store data in a row-column format or in other words tabular format. The most common approach used is to store data in the form of JSON document. JSON, which is a very popular format for data exchange can actually be used to organize data in key-value pairs. There is actually a wide classification under which NoSQL databases can be broadly classified. They are as follows:

- Key-value stores
- Document databases
- Wide-column stores
- Graph stores

The following are some features of NoSQL which makes it the default choice for such large-scale systems:

- Non-relational databases are schema-less. This will translate roughly into saying that they provide and offer much more flexibility when it comes to making changes in the way data is stored and organized.
- NoSQL databases are cost-effective and open sourced. This means that they are both nicely customizable by the enterprise wanting to use them and also come at a fraction of the cost of the traditional SQL giants.
- NoSQL databases is highly scalable. This is by far their USP when it comes to comparison with SQL databases, as in the age of cloud computing and on-demand scalability in no time, NoSQL data stores scale quickly and can be spread out of 100s of VM's at once. They also function nicely on low-cost hardware that means that they are even effective when the system configurations are not that great.

they also have some fare share of disadvantages as compared to RDBMS, which is listing as follows:

- Not that great when it comes to transactional systems. They are very badly in comparison to their SQL counterparts.
- Lack of standardization, when it comes to querying the data. In the case of RDBMS, there is SQL which is almost the same (given some changes here and there) for all the major vendors. NoSQL vendors still implement their own querying commands which becomes a pain while trying to do migrations and move from one database to another kind of database.

MongoDB

MongoDB is the most famous and widely used NoSQL database in the world as of now. The latest version is version 3.6. To install MongoDB on your system, you just need to go to the website <https://docs.mongodb.com/getting-started/shell/tutorial/install-mongodb-on-os-x/> and follow the instructions given there.

Once installed, you can open up two Terminals. In one of them, just type `mongod` to start the MongoDB daemon; while in the second Terminal, just type `mongo` and the shell will start.

Once you are inside the shell, you are ready to play with it. Just like the Julia REPL, mongo shell too gives you the flexibility of autocomplete. The following are some commands that are very easy to run and can get you a good feel of how a document-based database looks and runs:

```

# to show all databases
> show databases;
admin
0.000GB
local
0.000GB
test
0.000GB
>

# to change database
> use test;
switched to db test

# to insert a document
> db.test.insertOne({"name":"rahul","book":"learning julia"})
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5a118b098368a9901c9da4ff")
}

# to find all the documents
> db.test.find().forEach(printjson)
{
  "_id" : ObjectId("5a118b098368a9901c9da4ff"),
  "name" : "rahul",
  "book" : "learning julia"
}

# to find a document fulfilling a condition
> db.test.findOne({"name":"rahul"})
{
  "_id" : ObjectId("5a118b098368a9901c9da4ff"),
  "name" : "rahul",
  "book" : "learning julia"
}
>

```

For connecting and using MongoDB with Julia, we will use a package named `Mongo.jl`. Before installing the package, you need to have the latest version of the MongoDB binaries for C language. The following is how you can install and use this package:

```
julia> using Mongo, LibBSON
```

```

# Create a client connection
julia> client = MongoClient()

# Get a handle to collection named "test" in database "test".
# Client object, database name, and collection name are stored
as variables.
julia> test = MongoCollection(client, "test", "test")

# Insert a document
julia> document = insert(test, Dict("name" => "rahul", "book"
=> "learning julia"))

```

Introduction to REST

REST is the underlying architecture that powers the most modern web application running on the internet. It offers a simpler form of architecture as compared to the traditional SOAP and WSDL-based ones.

Representational State Transfer (REST) is a simple way of sending and receiving data between the client and server, which is majorly done using HTTP and the data is transported or exchanged using a JSON format for the most part. It's a truly lightweight alternative to the Simple Object Access Protocol (SOAP)-based web applications.

REST uses HTTP methods such as GET , POST , PUT , and DELETE , for updating resources on the server. It provides the simplicity of a uniform interface as well as the scalability to support a large number of components.

The following are some architectural constraints that define a RESTful system

- **Client-server architecture:** The main idea behind client-server constraints is the separation of concerns. Separating the UI from the data storage greatly improves the portability as well as scalability across multiple platforms.
- **Statelessness:** Each request from the client should be complete in itself in the sense that it should hold all the necessary information to make it possible for the server to handle and respond to the client's request.
- **Cacheability:** As on the World Wide Web, clients and intermediaries can cache responses. Responses must, therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from reusing stale or inappropriate data in response to further requests.
- **Layered system:** A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches.

- **Code on demand:** Servers can temporarily extend or customize the functionality of a client by transferring executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.
- **Uniform interface:** The uniform interface constraint is fundamental to the design of any REST service. It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:
 - Resource identification in requests
 - Resource manipulation through representations
 - Self-descriptive messages
 - Hypermedia as the engine of application state

What is JSON?

JSON, short for JavaScript Object Notation, is like the gold standard of data transfer between client and server inside a RESTful web application. The following is an example of what a JSON object looks like:

```
{
  "name": "rahul",
  "subjects_with_scores": {
    "maths": 80,
    "science": 90,
    "computers": 100
  },
  "country": "india",
}
```

As you can see, we have a very simple JSON object represented here. It uses a format which uses key:value pairs to store data.

One of the flaws that JSON has is its inability to have comments on it. However, there is a version of JSON in the JSON5 format, which is very similar to JSON but can hold comments along with some minor improvements that can increase the overall readability for the user.

Now let's take a look at the following code:

```
julia> using Requests

julia> using JSON

julia> result = Requests.get("http://httpbin.org/get")
Response(200 OK, 14 headers, 281 bytes in body)

julia> typeof(result)
```

HttpCommon.Response

```
julia> fieldnames(result)
8-element Array{Symbol,1}:
:status
:headers
:cookies
:data
:request
:history
:finished
:requests
```

```
julia> println(result.data)
```

```
julia> typeof(result.data)
Array{UInt8,1}
```

```
julia> output = JSON.parse(convert(String, result.data))
Dict{String,Any} with 4 entries:
"headers" => Dict{String,Any} (Pair{String,Any} ("Connection",
"close"), Pair{St...
"args"      => Dict{String,Any} ()
"Url"       => "http://httpbin.org/get"
"origin"    => "35.188.142.162"
```

```
julia> output["headers"]
Dict{String,Any} with 4 entries:
"Connection" => "close"
"Host"       => "httpbin.org"
"Accept"     =>
"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=..
.
"User-Agent" => "Requests.jl/0.0.0"
```

Similarly, we can also POST data to the server by using the `Requests.post` function:

```
julia> result = Requests.post("http://httpbin.org/post"; data =
"this is julia")
Response(200 OK, 14 headers, 436 bytes in body)

julia> JSON.parse(convert(String, result.data))Dict{String,Any}
with 8 entries:
```



```
"headers" => Dict{String,Any}(Pair{String,Any}("Connection",
"close"),Pair{St...
"json" => nothing
"files"=> Dict{String,Any}()
"args" => Dict{String,Any}()
"data"=> "this is julia"
"url" => "http://httpbin.org/post"
"form" => Dict{String,Any}()
"origin" => "35.188.142.162"
```