Julia Cheatsheet

The **Julia cheatsheet** provides a quick reference to all its fundamental topics. Julia is a high-performance, dynamic language designed for fast numerical and scientific computing. By learning this cheat sheet, the user can prepare for the interviews and exams. Go through this cheat sheet and learn the **Julia programming language**.

Table of Content

- Tuples
- DataFrames.jl for Data Manipulation

1. Introduction to Julia

Julia is a high−level programming language designed for scientific and technical computing. Its program is used in various fields such as machine learning, data analysis, numerical computing, and distributed computing.

```
println("Hello, World!")
```

2. Comments

Comments are used to show the information about the code.

```
# single line comments

#=
multi-line comments
=#
```

3. Julia REPL Shortcuts

The Julia REPL provides various shortcuts to improve user productivity. Below are a few commands −

- **Ctrl + C** − To interrupt the execution.
- **Ctrl + D** − To exit the REPL.

- **Tab** – For autocompletion of commands.
- **?command** – To get help on a command.

```
# Help on println function
?println
```

4. Variable Assignment and Scope

In Julia, users can assign values to variables using the **=** operator. The global variables are defined under the scope, whereas local variables are defined inside the function. The local variable is also known as the actual variable.

```
# Global variable
x = 10
y = 5
function example()
    # Local variable
    z = 20
    println(x, y, z)
end
example()
```

5. Data Types

In Julia, we have various data types such as Int, Float, String, and more.

```
# Data types
a = 10   # Integer
b = 3.14 # Float
c = "Hello" # String
```

6. Type Conversion

In Julia, we can perform the type conversion using functions like Int(), Float(), etc.

```
# Type conversion
x = 3.14
# Converts float to integer
y = Int(x)
```

7. Constants in Julia

In Julia, constant are defined by the keyword consts and their values cannot be reassigned.

```
# Constant definition
const Pi = 3.14159
```

8. Operators

Julia supports various operators, including arithmetic, logical, and comparison operators.

Operators	Description	Example
Arithmetic Operators	Basic mathematical operations.	'a + b', 'a - b', 'a * b', 'a / b', 'a % b'
Relational Operators	Compare two values.	'a == b', 'a != b', 'a > b', 'a = b', 'a
Logical Operators	Combine conditional statements.	'a && b', 'a || b', '!a'
Bitwise Operators	Operate at the bit level.	'a & b', 'a | b', 'a b', '~a', 'a > b'
Assignment Operators	Assign values to variables.	'a = b', 'a += b', 'a -= b', 'a *= b', 'a /= b', 'a %= b'

9. Conditional Statements (if, else, elseif)

In Julia, conditional statements allow users to control the flow of the program based on conditions.

```
# Conditional statements
if x > y
    println("x is greater than y")
elseif x == y
    println("x is equal to y")
else
    println("x is less than y")
end
```

10. Loops (for and while)

Loops are used for the repeated execution of a block of code.

```
for i in 1:5
    println(i)
end
```

The implementation of while loop –

```
i = 1
while i <= 5
    println(i)
    i += 1
end
```

11. Exception Handling (try, catch, finally)

In Julia, **exception handling** is the process to handle the error in a controlled way. This prevents the program from crashing.

```
# Exception handling
try
    x = 10 / 0
catch e
    println("Error: ", e)
finally
```

```
    println("Execution finished")
end
```

12. Defining and Calling Functions

In Julia, **functions** are defined using the keyword "function" and can be called directly by the function name.

```
# Defining the greet function
function fun(name)
    println("Hello, $name!")
end

# Calling the greet function
fun("Julia")
```

13. Anonymous and Higher-Order Functions

The **anonymous functions** are defined without names, whereas higher-order functions accept other functions as arguments.

```
# Anonymous function
f = x -> x^2
println(f(3))

# Higher-order function
function apply_function(f, x)
    return f(x)
```

```
end
println(apply_function(f, 4))
```

14. Multiple Dispatch

The **multiple dispatch** in Julia means the method is determined by the types of all its arguments.

```
# Multiple dispatch
function area(r::Int)
    return  * r^2
end

function area(l::Int, w::Int)
    return l * w
end

# Circle
println(area(5))
# Rectangle
println(area(4, 6))
```

15. Type Annotations

Type annotations are used to specify the types of function arguments and return values.

```
```

```
# Type annotations
function add(x::Int, y::Int)::Int
    return x + y
end
println(add(2, 3))
```

16. Arrays

In Julia, **arrays** are defined by the ordered collections of elements. They can hold any type of data.

```
# Array
arr = [1, 2, 3, 4]
# Accessing the first element
println(arr[1])
```

17. Matrices

Matrices are two−dimensional arrays that are particularly useful for linear algebra.

```
# Define two 2D matrices
A = [1 2 3; 4 5 6; 7 8 9]
B = [9 8 7; 6 5 4; 3 2 1]
```

```
# Add the matrices
C = A + B

# Display the result
println("Matrix A:")
println(A)
println("\nMatrix B:")
println(B)
println("\nSum of A and B:")
println(C)
```

18. Indexing

In Julia, **indexing** is used to access elements from arrays or matrices.

```
# Indexing
# Access second element
println(arr[2])

# Access element in row 1, column 2
println(mat[1, 2])
```

19. Tuples

Tuples are defined by immutable ordered collections. They are useful for grouping different types of data.

```
# Tuple
t = (1, "Hello", 3.14)

# Access first element
println(t[1])
```

20. Named Tuples

In Julia, **named tuples** are tuples where each element has a name.

```
# Named Tuple
nt = (x = 100, y = 2)
println(nt.x)
```

21. Dictionaries

In Julia, **dictionaries** are defined by key-value pairs that are used to store data.

```
# Dictionary
d = Dict("a" => 10, "b" => 20)
```

```
# Access value by key
println(d["a"])
```

22. Sets

In Julia, **sets** are unordered collections of unique elements.

```
# Set
s = Set([1, 2, 3, 4])

# Check if element exists
println(3 in s)
```

23. String Manipulation

Below is the execution of string manipulation −

```
# String manipulation
str = "Hello, Julia!"
# Get length of string
println(length(str))
println(string(str, " is awesome"))
```

24. String Interpolation

In Julia, **string interpolation** is the process of inserting the variable into the string.

```
# Define a variable
name = "Julia"
age = 25

# Use string interpolation to create a sentence
result = "Hello, my name is $name and I am $age years old."

# Display the result
println(result)
```

25. Regular Expressions in Julia

In Julia, **regular expressions** are used for pattern matching.

```
# Regular expressions
str = "TOM MIKE RAM SHYAM"
r = r"\b\w+"
matches = eachmatch(r, str)
for m in matches
```

```
    println(m.match)
end
```

26. File Handling

Julia provides functions for reading from and writing to files.

```
open("example.txt", "w") do f
    write(f, "Hello, World!")
end
```

27. Working with CSV and JSON Files

In Julia, a **CSV file** is a text file that stores tabular data, where each line represents a row, and values are separated by commas. While JSON files are commonly used to store data as key-value pairs.

```
# Reading CSV
using CSV
data = CSV.File("data.csv")
for row in data
    println(row)
end
```

28. Creating Modules

Modules in Julia allow users to organize code into namespaces.

```
# Creating module
module MyModule
export my_function
function my_function(x)
    return x^2
end
end
```

29. Installing and Managing Packages

Julia is a **package manager** that allows users to easily install and manage packages.

```
# Installing a package
using Pkg
Pkg.add("Plots")
```

30. Metaprogramming and Macros

Metaprogramming allows users to write code for generating program, while macros provide a way to transform code before evaluation.

```
# Simple macro
macro say_hello()
    return quote
        println("Hello, Tutorialspoint!")
```

```
    end
end
@say_hello
```

31. Multithreading and Parallel Computing

Julia has **multithreading** and **parallel computing** for faster execution.

```
# Multithreading
Threads.@threads for i in 1:10
    println(i)
end
```

32. Benchmarking and Performance Optimization

The **benchmarking** defines the process to check the performance of the program. While performance optimization improves the efficiency of the program.

```
# Benchmarking
using BenchmarkTools
@btime sum(1:1000)
```

33. Just-In-Time (JIT) Compilation with LLVM

Julia uses **LLVM** for Just-In-Time (JIT) compilation, providing high-performance execution.

```
# JIT compilation
function fib(n)
    if n <= 1
        return n
    else
        return fib(n-1) + fib(n-2)
    end
end
println(fib(10))
```

34. Broadcasting and Vectorized Operations

Broadcasting allows for element-wise operations on arrays and matrices without explicitly writing loops.

```
# Example: Broadcasting
a = [1, 2, 3]
b = [4, 5, 6]
println(a .+ b)  # Element-wise addition
```

35. Working with Dates and Time

In Julia, we can use the package along with its built-in function to get the current **time and date**.

```
# Dates and Time
using Dates
today = today()
println(today)
```

36. Random Number

In Julia, the function **rand()** is used to generate random numbers within a specified range.

```
# Random number generation
using Random
# Random integer between 1 and 10
println(rand(1:10))
```

37. Linear Algebra and Matrix Operations

Below is the implementation to perform the **matrix operation** −

```
# Example: Matrix operations
A = [1 2; 3 4]
B = [5 6; 7 8]
```

```
# Matrix multiplication
println(A * B)
```

38. Plotting with Plots.jl

Julia provides excellent **plotting** capabilities through the Plots.jl package.

```
# Plotting
using Plots
plot([1, 2, 3], [4, 5, 6])
```

39. DataFrames.jl for Data Manipulation

In Julia, **DataFrames.jl** is a powerful package for handling and manipulating data in Julia.

```
# DataFrames
using DataFrames
df = DataFrame(A = 1:3, B = 4:6)
println(df)
```