Julia Cheatsheet

Basic Syntax and Execution

Julia is a modern programming language designed for high-performance numerical computing and data science. It aims for the ease of Python and the performance of C.

Variable Definition

Julia supports dynamic typing, but explicit type annotations can be used for optimization or clarity.

```
x = 10              # Int64
y = "Hello!"         # String
z = 3.14            # Float64
```

Function Definition and Call

Functions are defined using the `function` keyword, and the last evaluated value is returned automatically.

```
# Standard definition
function greet(name)
  return "Hello, $name!"
end

# Assignment form (short form)
add(x, y) = x + y

# Calling
println(greet("Julia"))
```

Control Flow

If-Else

```
a = 5
if a > 0
  println("Positive")
elseif a < 0
  println("Negative")
else
  println("Zero")
end
```

Loops (For & While)

```
# For loop
for i in 1:5
  println(i)
end

# While loop
i = 1
while i ≤ 5
  println(i)
  i += 1
end
```

Data Structures

Arrays

Julia's array indexing **starts from 1**, and it is optimized for multi-dimensional array processing.

```
# 1D Array (Vector)
arr = [1, 2, 3, 4, 5]

# Multi-dimensional Array (Matrix)
# Space separates columns, semicolon
separates rows.
matrix = [1 2; 3 4]
```

Dictionaries

Mutable collection of key-value pairs.

```
# Create
dict = Dict("name" ⇒ "Julia", "version"
⇒ 1.9)

# Access
println(dict["name"])
```

Strings

Concatenate using `*` operator, and support interpolation with `$`.

```
str1 = "Hello"
str2 = "Julia"

# Concatenation
combined_str = str1 * " " * str2

# Interpolation
name = "Alice"
message = "Hello, $(name)!"
```

Tuples

Fixed-size, immutable collection.

```
# Create and access
tup = (1, "hello", 3.14)
println(tup[2]) # "hello"

# Unpacking
a, b, c = tup
```

Structs

Used for user-defined data types. Immutable by default.

```
# Immutable struct
struct Point
  x::Int
  y::Int
end

# Use 'mutable struct' if modification
is needed
mutable struct MutablePoint
  x::Int
end

# Instance creation
p = Point(10, 20)
println(p.x)
```

Advanced Features

Exception Handling

Use `try-catch` blocks to manage runtime errors.

```
function divide(a, b)
  try
    return a / b
  catch e
    println("Error occurred: $e")
  end
end
```

Modules and Package Management

Use `using` to load modules and `Pkg` for package management.

```
# Use standard library or installed
module
using LinearAlgebra
println(det([1 2; 3 4])) # Determinant

# Add package (via REPL ']' mode or Pkg)
# using Pkg
# Pkg.add("DataFrames")
```

Broadcasting

The dot (`.`) operator is a powerful feature to apply functions or operations element-wise over arrays.

```
arr = [1, 2, 3]
result = arr .+ 10
println(result) # [11, 12, 13]
```

Comprehensions

Concise way to create new arrays or dictionaries based on existing collections.

```
# Array comprehension
squares = [i^2 for i in 1:5] # [1, 4, 9,
16, 25]

# Dictionary comprehension
dict_comp = Dict(i ⇒ i^2 for i in 1:3)
```

Useful Tips

```
# Help mode: Enter '?' in REPL then a
function name to see docs.
# ?println

# Check package status
using Pkg
Pkg.status()
```