

Julia Syntax Cheatsheet

Applied Optimization with Julia

Variables and Basic Types

Variable Declaration and Types

```
# Basic variable declaration
x = 1                # Implicit typing
y::Int64 = 5         # Explicit type annotation

# Common types
num_int = 42         # Integer
num_float = 19.99    # Float
is_student = true    # Boolean
name = "Julia"       # String

# Check type
typeof(num_int)      # Returns Int64
typeof(num_float)    # Returns Float64
```

String Interpolation

```
name = "Julia"
age = 30
# Basic interpolation
message = "I am $age years old"
# Complex interpolation
greeting = "Hello, my name is $name and I am $age years old"
```

Type Conversion

```
# Convert between types
float_num = Float64(42)  # Int to Float
int_num = Int64(3.14)    # Float to Int
str_num = string(42)     # Number to String
```

Key Points

- Variables are dynamic, types are not
 - Use `typeof()` to check variable type
 - String interpolation is powerful for formatted output
-

Vectors, Matrices, and Tuples

Vectors

```
# Create vectors
grades = [95, 87, 91, 78, 88] # Numeric vector
names = ["Mike", "Yola", "Elio"] # String vector

# Vector operations
push!(grades, 82) # Add element to end
pop!(grades) # Remove last element
popfirst!(grades) # Remove first element

# Vector indexing
first = grades[1] # Access first element
subset = grades[1:3] # Access first three elements
```

Matrices

```
# Create matrices
matrix = [1 2 3; 4 5 6] # 2x3 matrix
# Matrix operations
matrix[2,3] = 17 # Change specific element

# Matrix arithmetic
matrix1 = [2 2; 3 3]
matrix2 = [1 2; 3 4]
sum_matrix = matrix1 + matrix2 # Matrix addition
prod_matrix = matrix1 * matrix2 # Matrix multiplication
element_prod = matrix1 .* matrix2 # Element-wise multiplication

# Broadcasting
matrix .+ 10 # Add 10 to each element
```

Tuples

```
# Create tuples (immutable)
person = ("Elio Smith", 18, "Hamburg")
rgb = (255, 0, 0)

# Tuple operations
name = person[1] # Access first element
age, city = person[2:3] # Multiple assignment
```

Key Differences

- Vectors: Mutable, 1-dimensional, good for lists
 - Matrices: Mutable, 2-dimensional, good for linear algebra
 - Tuples: Immutable, fixed-size, good for grouping related constants
-

Comparison and Logical Operators

Basic Comparisons

```
# Comparison operators
x == y    # Equal to
x != y    # Not equal to
x < y     # Less than
x > y     # Greater than
x <= y    # Less than or equal to
x >= y    # Greater than or equal to

# Examples
password_correct = (input == "secret123")
is_adult = (age >= 18)
can_afford = (price <= budget)
```

Logical Operators

```
# AND operator (&&)
can_buy = (age >= 18) && (money >= price)    # Both conditions must be true

# OR operator (||)
need_coat = (temp < 10) || is_raining        # At least one must be true

# NOT operator (!)
is_closed = !is_open                          # Inverts boolean value
```

Chained Comparisons

```
# Instead of
x >= 0 && x <= 10    # Check if x is between 0 and 10

# You can write
0 <= x <= 10         # More natural syntax

# Real-world examples
normal_temp = 36.5 <= body_temp <= 37.5
work_hours = 9 <= current_hour < 17
```

Key Points

- Comparisons return boolean values (`true` or `false`)
 - `&&` requires all conditions to be true
 - `||` requires at least one condition to be true
 - `!` inverts a boolean value
 - Chained comparisons make range checks more readable
-

Loops and Iterations

For Loops

```
# Basic for loop with range
for i in 1:3
    println(i)      # Prints 1, 2, 3
end

# Iterating over array
fruits = ["apple", "banana", "cherry"]
for fruit in fruits
    println(fruit)  # Prints each fruit
end

# For loop with break
for x in 1:10
    if x == 4
        break      # Exits loop when x is 4
    end
end

# For loop with conditions
for x in 1:10
    if x <= 2
        println(x)
    elseif x == 3
        println("Three!")
    else
        break
    end
end
```

While Loops

```
# Basic while loop
number = 10
while number >= 5
    number -= 1      # Decrements until < 5
end

# Infinite loop with break
current = 0
while true
    current += 1
    if current == 5
        break      # Exits when condition met
    end
end

# While loop with condition
lives = 3
while lives > 0
```

```

    lives -= 1      # Continues until lives = 0
end

```

Nested Loops

```

# Nested loop example
sizes = ["S", "M", "L"]
colors = ["Red", "Blue"]
for size in sizes
    for color in colors
        println("$color $size")
    end
end

# Matrix iteration
for i in 1:3
    for j in 1:2
        println("Position: $i,$j")
    end
end

```

List Comprehensions

```

# Basic list comprehension
squares = [n^2 for n in 1:5]      # [1,4,9,16,25]

# With condition
evens = [n for n in 1:10 if n % 2 == 0]      # [2,4,6,8,10]

# Nested comprehension
matrix = [i*j for i in 1:3, j in 1:3]      # 3x3 multiplication table

```

Key Points

- `for` loops are best when you know the number of iterations
 - `while` loops are useful for unknown iteration counts
 - Use `break` to exit loops early
 - List comprehensions offer concise array creation
 - Nested loops are useful for multi-dimensional iteration
-

Dictionaries

Basic Dictionary Operations

```

# Create a dictionary
student_ids = Dict{
    "Elio" => 1001,
    "Bob" => 1002,
    "Yola" => 1003
}

```

```

# Access values
id = student_ids["Elio"]      # Get value by key
student_ids["David"] = 1004   # Add new key-value pair
delete!(student_ids, "Bob")   # Remove entry

# Check key existence
if haskey(student_ids, "Eve")
    println(student_ids["Eve"])
end

```

Advanced Operations

```

# Dictionary with array values
grades = Dict{
    "Elio" => [85, 92, 78],
    "Bob" => [76, 88, 94]
}

# Get all keys and values
names = keys(grades)      # Get all keys
scores = values(grades)    # Get all values

# Iterate over dictionary
for (student, grade_list) in grades
    avg = sum(grade_list) / length(grade_list)
    println("$student: $avg")
end

```

Common Methods

```

# Dictionary methods
length(dict)      # Number of entries
empty!(dict)      # Remove all entries
get(dict, key, default) # Get value or default if key missing
merge(dict1, dict2) # Combine two dictionaries
copy(dict)        # Create shallow copy

```

Key Points

- Keys must be unique
- Values can be of any type (including arrays)
- Use `haskey()` to safely check for key existence
- Dictionaries are mutable (can be changed)
- Keys are accessed with square brackets `dict["key"]`

Functions

Basic Function Definition

```

# Basic function with explicit return
function say_hello(name)
    return "Hello, $(name)!"
end

# Function with implicit return
function multiply(a, b)
    a * b    # Last expression is automatically returned
end

# Conditional return
function do_something(a, b)
    if a > b
        return a * b
    else
        return a + b
    end
end

```

Function Scope

```

# Local scope example
function bake_cake()
    secret_ingredient = "vanilla"    # Only exists inside function
    return secret_ingredient         # Must return to access outside
end

# Variables outside function not accessible inside
global_var = 10
function scope_example()
    # Can read global_var but can't modify it
    return global_var + 5
end

```

Multiple Dispatch

```

# Generic operation for all types
function operation(a, b)
    "Generic operation for $(typeof(a)) and $(typeof(b))"
end

# Type-specific implementations
operation(a::Number, b::Number) = a + b    # For numbers
operation(a::String, b::String) = string(a, b) # For strings

# Usage examples
operation(10, 20)    # Returns 30
operation("Hello", "!") # Returns "Hello!"
operation("Hi", 42)   # Uses generic operation

```

Key Points

- Functions can have explicit or implicit returns

- Last expression is automatically returned if no `return` statement
 - Variables inside functions are local by default
 - Multiple dispatch allows different behavior based on argument types
 - Use `return` for early exits or conditional
-

Package Management

Basic Package Operations

```
# Import package manager
import Pkg          # Access as Pkg.function()
using Pkg           # Import all exported names

# Add packages
Pkg.add("DataFrames") # Add single package
Pkg.add(["Package1", "Package2"]) # Add multiple packages

# Update packages
Pkg.update()          # Update all packages
Pkg.update("DataFrames") # Update specific package

# Remove packages
Pkg.rm("DataFrames")  # Remove package
```

Package Usage

```
# Import packages
import DataFrames    # Access as DataFrames.function()
using DataFrames     # Import all exported names

# Check installed packages
Pkg.status()         # List all installed packages
```

Environment Management

```
# Environment operations
Pkg.activate("new_environment") # Create/activate environment
Pkg.activate()                 # Activate default environment

# Project files
# Project.toml    - Lists direct dependencies
# Manifest.toml   - Complete dependency graph
```

Key Points

- Use `import` for namespace control, `using` for direct access
- Always update packages regularly with `Pkg.update()`
- Create separate environments for different projects
- Project.toml and Manifest.toml track dependencies
- Package manager commands typically run in REPL

DataFrames

Creating DataFrames

```
using DataFrames

# Basic DataFrame creation
df = DataFrame(
    Name = ["John", "Mike", "Frank"],
    Age = [28, 23, 37],
    Salary = [50000, 62000, 90000]
)

# Empty DataFrame with specified columns
df_empty = DataFrame(
    Name = String[],
    Age = Int[]
)
```

Accessing and Modifying Data

```
# Access columns
ages = df.Age                # Get Age column
first_name = df.Name[1]      # First name in Name column

# Modify values
df.Salary[1] = 59000          # Update John's salary
df.NewColumn = zeros(3)      # Add new column

# Access multiple columns
subset = df[:, [:Name, :Age]] # Select specific columns
row = df[1, :]                # Select first row
```

Filtering Data

```
# Filter with boolean indexing
high_earners = df[df.Salary .> 60000, :]

# Using filter function
high_earners = filter(row -> row.Salary > 60000, df)

# Multiple conditions
senior_high_earners = df[(df.Age .> 30) .& (df.Salary .> 60000), :]
```

Data Manipulation

```
# Sort DataFrame
sorted_df = sort(df, :Age)      # Sort by Age
sorted_df = sort(df, [:Age, :Salary]) # Sort by multiple columns

# Add calculated column
```

```
df.Bonus = [row.Age > 30 ? row.Salary * 0.1 : row.Salary * 0.05 for row in
eachrow(df)]

# Iterate over rows
for row in eachrow(df)
    println("${row.Name}: ${row.Age} years old")
end
```

Key Functions

nrow(df)	# Number of rows
ncol(df)	# Number of columns
names(df)	# Column names
describe(df)	# Summary statistics
push!(df, row)	# Add new row
select(df, :Name)	# Select columns

Key Points

- Column access with dot notation (df.column)
 - Use eachrow() for row iteration
 - Boolean indexing for filtering
 - push! to add new rows
 - Broadcasting with dot operators (.>, .+, etc.)
-

File Input/Output

DelimitedFiles Operations

```
using DelimitedFiles

# Write matrix to CSV
data = [1 2 3; 4 5 6]
writedlm("data.csv", data, ',')      # Write with comma delimiter
writedlm("data.txt", data, '\t')     # Write with tab delimiter

# Read delimited files
matrix = readdlm("data.csv", ',')    # Read CSV file
matrix = readdlm("data.txt", '\t')   # Read tab-delimited file
```

CSV and DataFrame Operations

```
using CSV, DataFrames

# Write DataFrame to CSV
df = DataFrame(
    Name = ["John", "Alice"],
    Age = [25, 30]
)
CSV.write("data.csv", df)            # Basic write
```

```

CSV.write("data.csv", df,           # Write with options
          delim = ';',              # Custom delimiter
          header = false             # No header
)

# Read CSV to DataFrame
df = CSV.read("data.csv", DataFrame) # Basic read
df = CSV.read("data.csv", DataFrame, # Read with options
          delim = ';',              # Custom delimiter
          header = ["Col1", "Col2"] # Custom headers
)

```

File Path Management

```

# Get current directory
@__DIR__           # Directory of current file
pwd()              # Current working directory

# Path operations
path = joinpath(@__DIR__, "data") # Join path components
mkdir(path)                  # Create directory
isfile(path)                 # Check if file exists

```

Key Points

- Use DelimitedFiles for simple matrix I/O
 - CSV package for advanced DataFrame I/O
 - Always use `@__DIR__` for relative paths
 - Check file existence before operations
 - Consider using try-catch for file operations
-

Plotting with Plots.jl

Basic Plots

```

using Plots, StatsPlots

# Line plot
plot(x, y,
      title="Line Plot",
      xlabel="X Label",
      ylabel="Y Label",
      legend=false
)

# Scatter plot
scatter(x, y,
        title="Scatter Plot",
        marker=(:circle, 8)
)

```

```

# Bar plot
bar(categories, values,
     title="Bar Plot"
)

# Histogram
histogram(data,
     bins=30,
     title="Histogram"
)

# Box plot
boxplot(group, values,
     title="Box Plot"
)

```

Plot Customization

```

# Customize plot appearance
plot(x, y,
     title="Custom Plot",
     line=(:dash, 2),      # Line style and width
     color=:red,           # Line color
     marker=(:circle, 8),  # Marker style and size
     label="Data Series"    # Legend label
)

# Multiple series
plot(x, y1, label="Series 1")
plot!(x, y2, label="Series 2") # Add to existing plot

```

Saving Plots

```

# Save plot to file
savefig(plot_name, "path/plot.png") # Save as PNG
savefig(plot_name, "path/plot.pdf") # Save as PDF
savefig(plot_name, "path/plot.svg") # Save as SVG

```

Common Options

```

# Plot options
plot(
     legend=true/false,    # Show/hide legend
     grid=true/false,      # Show/hide grid
     size=(width,height),   # Plot dimensions
     dpi=300               # Resolution
)

# Line styles
:solid, :dash, :dot

```

```
# Colors
:red, :blue, :green

# Markers
:circle, :square, :diamond
```

Key Points

- Use `plot()` for new plots, `plot!()` to add to existing
- Customize with named arguments
- Save plots in various formats
- StatsPlots extends plotting capabilities
- Multiple series can share one plot

Bibliography