FOR/WHILE/IF Guides for MSHELL.

About mshell.

mshell is a minimalistic Unix shell designed for resource-constrained environments, combining traditional shell functionality with integrated AI capabilities through local LLM models via Ollama or Linux LLM evaluation framework (current version is 1.3). Built in C with embedded Lua for mathematical operations, mshell provides a lightweight alternative to traditional shells while adding powerful AI-assisted command execution and code generation.

Platform Support:

- Works on various Linux distributions including Debian and Raspberry Pi OS
- Developed and tested on Ubuntu 24.04 and Raspberry Pi OS
- Minimal resource requirements for complex tasks in mathematics, visualization, and AI
- Optimized for use with Linux LLM evaluation frameworks or small LLM models

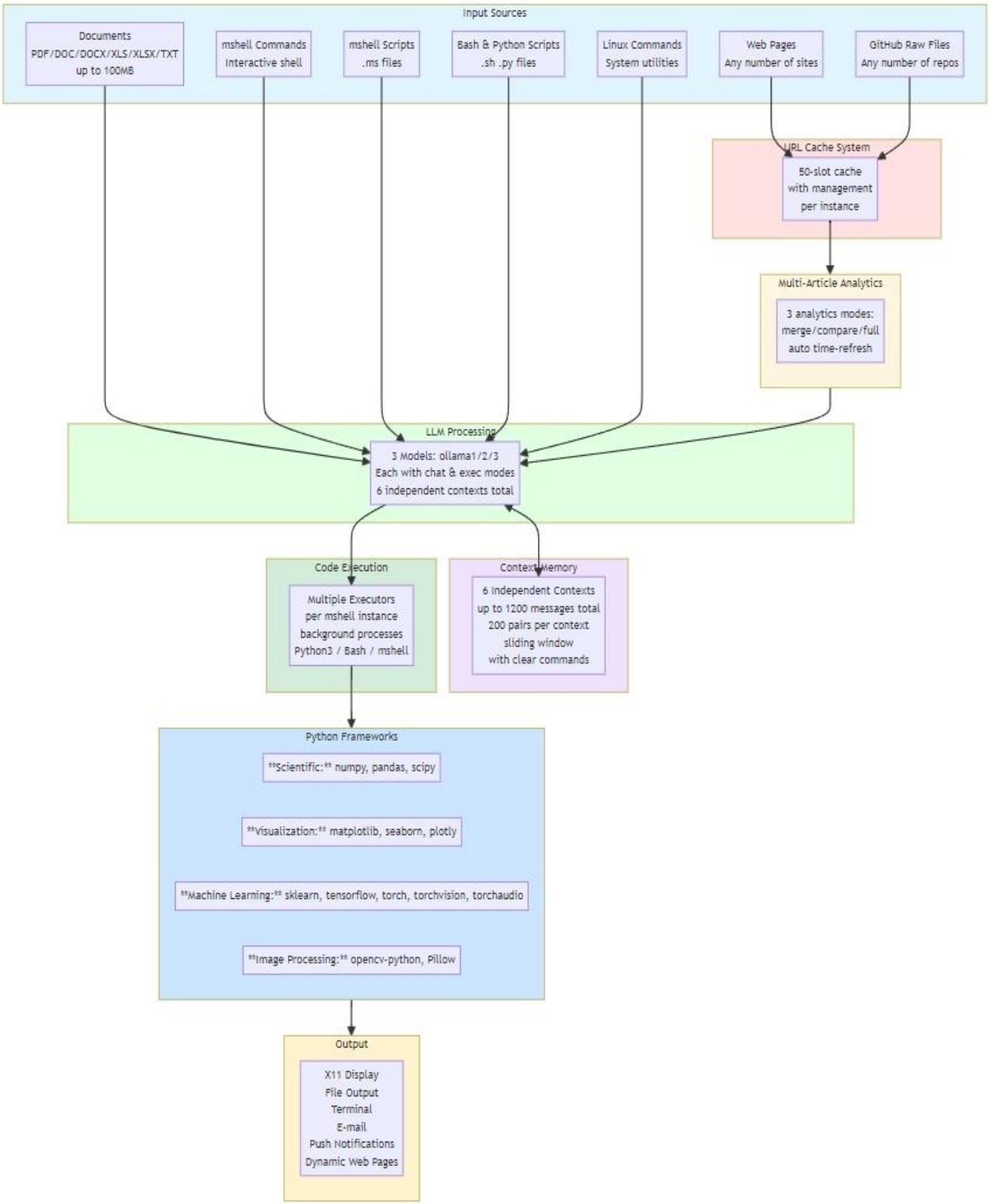
Key Capabilities:

- **AI Integration:** Direct access to multiple Ollama models (ollama1/2/3) in both chat and exec modes
- **Smart Code Execution:** Generate and automatically execute code using LLMs
- **Context Memory:** Persistent conversation history (up to 200 message pairs per model)
- **Web Content Analysis:** Built-in URL caching system (50 slots) for analyzing web pages with LLMs
- **Multi-Language Support:** Native execution of Python, Bash, Lua, and .ms scripts
- **Advanced Control Structures:** AST-based parsing for IF/FOR/WHILE loops and functions
- **Mathematical Operations:** Embedded Lua engine for complex calculations
- **Bash Compatibility:** Works with standard bash commands, pipes, and redirections

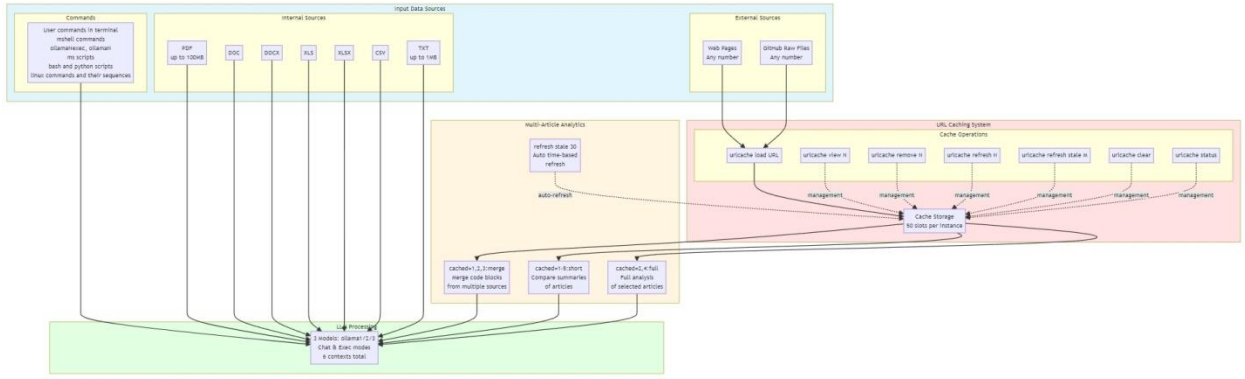
Ideal For:

- Developers working with AI/LLM-assisted workflows
- System administrators needing lightweight shell with AI capabilities
- Embedded systems and resource-constrained environments
- Rapid prototyping and interactive development
- Automated scripting with LLM integration

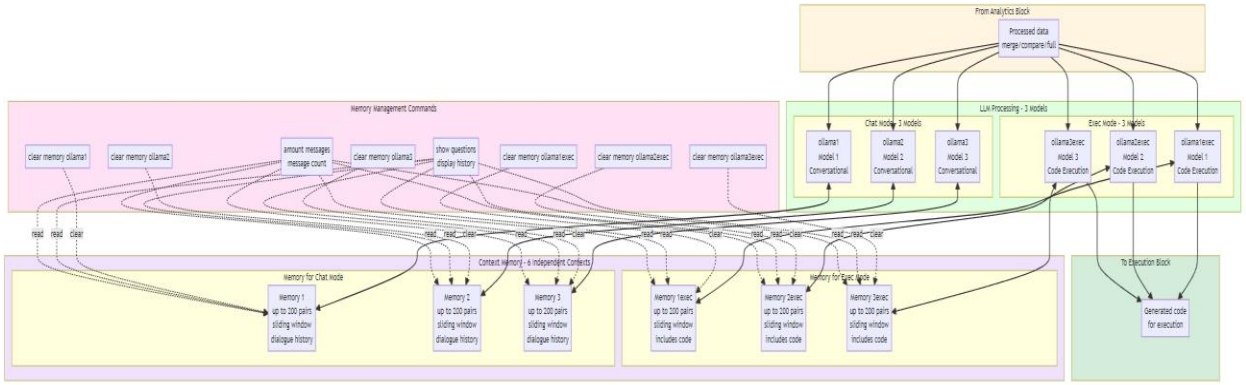
Common Simplified Diagram of mshell:


Input Sources
Documents
PDF/DOC/DOCX/XLS/XLSX/TXT
up to 100MB
mshell Commands
Interactive shell
mshell Scripts
.ms files
Bash & Python Scripts
.sh .py files
Linux Commands
System utilities
Web Pages
Any number of sites
GitHub Raw Files
Any number of repos
URL Cache System
50-slot cache
with management
per instance
Multi-Article Analytics
3 analytics modes:
merge/compare/full
auto time-refresh
LLM Processing
3 Models: ollama1/2/3
Each with chat & exec modes
6 independent contexts total
Code Execution
Multiple Executors
per mshell instance
background processes
Python3 / Bash / mshell
Context Memory
6 independent Contexts
up to 1200 messages total
200 pairs per context
sliding window
with clear commands
Python Frameworks
Scientific: numpy, pandas, scipy
Visualization: matplotlib, seaborn, plotly
Machine Learning: sklearn, tensorflow, torch, torchvision, torchaudio
Image Processing: opencv-python, Pillow
Output
X11 Display
File Output
Terminal
E-mail
Push Notifications
Dynamic Web Pages

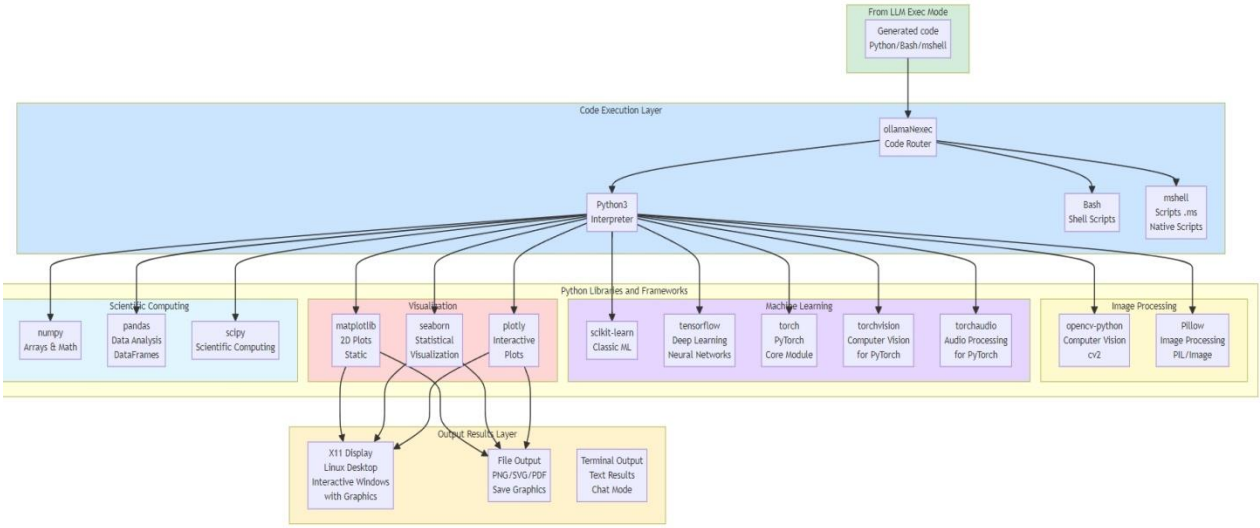
Input Sources – URL (omni) option – Cache - Analitics Communication Diagram of mshell:



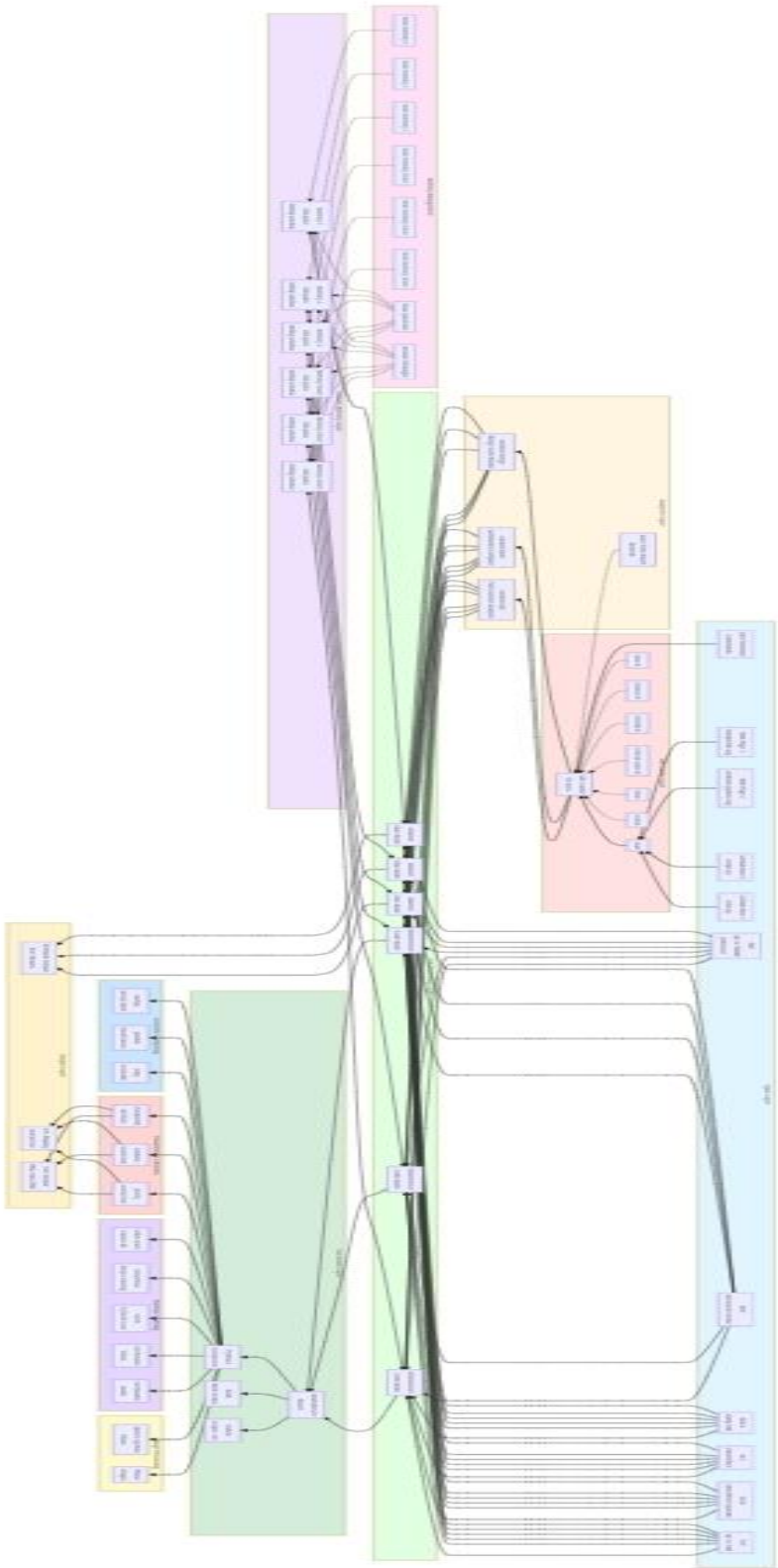
LLM Models – Context Memory Management Relationship Diagram of mshell:



Execution frameworks – Outputs Diagram for mshell:



General diagram of elements for mshell:



FOR_GUIDE - What's Included:

1. **Basic Syntax** - core FOR syntax with variable and list
2. **List Types** - literal words, numbers, mixed types, variable expansion, file paths
3. **Loop Variable** - how to access variable (with $), automatic type detection
4. **FOR with IF Conditions** - IF/IF-ELSE inside FOR, string/numeric/file tests
5. **Multiple Commands per Iteration** - multiple commands in FOR body
6. **Using Counters** - working with incr to count iterations
7. **String Processing** - splitting sentences into words, text processing
8. **File Operations** - checking file/directory existence (-f/-d/-e)
9. **Edge Cases** - empty list, single item, large lists
10. **What Works** - all list types, IF inside FOR, counters, file tests
11. **Common Patterns** - counting items, filtering with conditions, processing with status, string processing
12. **Best Practices** - descriptive variable names, proper spacing, initialize counters
13. **Common Mistakes** - forgetting $ when accessing, using $ in declaration, nested loops, wildcards
14. **Performance Tips** - minimize iterations, efficient conditions
15. **Comparison: FOR vs WHILE** - when to use list-based vs condition-based
16. **Summary** - formats with literal list, counter, condition, variable expansion

---

WHILE_GUIDE - What's Included:

1. **Basic Syntax** - core WHILE syntax with condition
2. **Condition Operators** - all comparison operators (numeric: -lt/-le/-gt/-ge/-eq/-ne, symbol: </<=/>=/>=, string: =/!=)
3. **Loop Control with incr** - working with counters and increment
4. **Comparing Variables** - comparing two variables in condition
5. **Multiple Commands per Iteration** - multiple commands in loop body
6. **Edge Cases** - zero iterations, single iteration, many iterations
7. **What Works** - all comparison operators, counters, variables
8. **Common Patterns** - simple counter, processing with limit, comparing variables, multiple operations
9. **Best Practices** - always increment, initialize variables, avoid infinite loops
10. **Common Mistakes** - forgetting incr, wrong initial value, undefined variables, IF in WHILE
11. **Performance Tips** - minimize iterations, use appropriate operators
12. **Comparison: WHILE vs FOR** - when to use condition-based vs list-based
13. **Summary** - key formats with counter and safeguard against infinite loop

---

IF_GUIDE - What's Included:

1. **Basic Syntax** - IF, IF-ELSE, IF-ELIF-ELSE formats
2. **Condition Operators** - numeric (-lt/-le/-gt/-ge/-eq/-ne), symbol (</<=/>/>=), string (=/!=/-z/-n), file tests (-f/-d/-e/-r/-w/-x)
3. **Comparing Variables** - comparing two numeric/string variables, variable with literal
4. **Multiple Commands** - multiple commands in THEN/ELSE blocks
5. **IF-ELSE Patterns** - simple IF-ELSE, IF-ELIF-ELSE, multiple ELIF branches
6. **Edge Cases** - empty string, zero value, negative numbers
7. **What Works** - all comparison operators, IF-ELSE, IF-ELIF-ELSE, multiple ELIF, nested IF
8. **Common Patterns** - range check, status check, file processing, grade calculator
9. **Best Practices** - clear conditions, spaces around brackets, descriptive names, handle both cases, ELIF instead of nested
10. **Common Mistakes** - forgetting $, missing spaces, forgetting then/fi, IF in WHILE
11. **Performance Tips** - order conditions by likelihood, early returns
12. **Comparison: IF vs Other Constructs** - IF vs WHILE, IF vs FOR, IF-ELIF vs multiple IF
13. **Testing Your IF Statements** - how to test TRUE/FALSE/ELIF branches
14. **Summary** - IF, IF-ELSE, IF-ELIF formats with numeric/string/file examples

---

Key Differences in Nuances:

WHILE is specific for:

- Condition-based looping
- Mandatory `incr` for counters
- Problem with IF inside WHILE
- Missing `decr` operator
- Safeguard against infinite loops

FOR is specific for:

- List-based iteration
- Different list types (words, numbers, paths)
- Variable expansion for lists
- IF WORKS inside FOR (unlike WHILE!)
- String processing (splitting into words)
- File operations checks

IF is specific for:

- Decision making (not loops)

- IF-ELIF-ELSE chains
- File test operators (-f/-d/-e/-r/-w/-x)
- String tests (-z/-n)
- Nested IF is possible
- DOES NOT work inside WHILE
- WORKS inside FOR

FOR Loop Guide for mshell

Overview

This guide explains how to write FOR loops in mshell, what works, best practices, and common use cases.

Basic Syntax

Standard FOR loop

mshell:

```
for variable in list
do
    command1
    command2
done
```

Components:

- `for` - keyword to start the loop
- `variable` - loop variable name (without $)
- `in` - keyword separator
- `list` - space-separated list of items
- `do` - marks start of loop body
- `done` - marks end of loop

List Types

1. Literal List of Words

mshell:

```
for fruit in apple banana cherry
do
    echo "Fruit: $fruit"
done
# Output: apple, banana, cherry
```

2. Numbers

mshell:

```
for num in 1 2 3 4 5
do
    echo "Number: $num"
done
# Output: 1, 2, 3, 4, 5
```

3. Mixed Data Types

mshell:

```
for item in 1 hello 3.14 world 5
do
    echo "Item: $item"
done
# Output: 1, hello, 3.14, world, 5
```

4. Variable Expansion

mshell:

```
colors="red green blue yellow"
for color in $colors
do
    echo "Color: $color"
done
```

```
# Output: red, green, blue, yellow
```

5. File Paths

mshell:

```
for dir in /home /etc /var /tmp
do
    echo "Directory: $dir"
done
```

Loop Variable

Accessing the Variable

Always use `$` when referencing the loop variable:

mshell:

```
for name in John Jane Bob
do
    echo "Hello, $name!"  # CORRECT - with $
done
```

mshell:

```
for name in John Jane Bob
do
    echo "Hello, name!"   # WRONG - prints "name" literally
done
```

Variable Type Detection

The loop variable automatically detects type:

mshell:

```
for item in 1 hello 3.14
do
    # item is numeric for 1 and 3.14
    # item is string for "hello"
```

```
    echo "$item"
done
```

FOR with IF Conditions

Basic IF inside FOR

```
mshell:
for num in 1 2 3 4 5 6 7 8 9 10
do
    if [ $num > 5 ]; then
        echo "$num is greater than 5"
    fi
done
# Output: 6, 7, 8, 9, 10
```

IF-ELSE inside FOR

```
mshell:
for num in 1 2 3 4 5
do
    if [ $num > 3 ]; then
        echo "$num is big"
    else
        echo "$num is small"
    fi
done
```

String Comparison

```
mshell:
for status in active inactive pending active
do
    if [ $status = active ]; then
        echo "Status is ACTIVE"
```

```
    fi
done
```

File/Directory Tests

mshell:

```
for path in /home /etc /var /nonexistent
do
    if [ -d $path ]; then
        echo "$path exists"
    else
        echo "$path not found"
    fi
done
```

Multiple Commands per Iteration

mshell:

```
for i in 1 2 3
do
    echo "Processing item $i"
    echo "Item $i completed"
    echo "---"
done
```

Using Counters

Simple Counter with incr

mshell:

```
count=0
for item in apple banana cherry
do
    incr count
```

```
    echo "$count: $item"
done
# Output: 1: apple, 2: banana, 3: cherry
```

Multiple Counters

```
mshell:
total=0
processed=0
for file in file1 file2 file3
do
    incr total
    if [ -f $file ]; then
        incr processed
    fi
done
echo "Processed $processed out of $total files"
```

String Processing

Processing Sentence into Words

```
mshell:
sentence="The quick brown fox jumps"
for word in $sentence
do
    echo "Word: $word"
done
# Output: The, quick, brown, fox, jumps
```

String Operations

```
mshell:
for name in John Jane Bob Alice
do
```

```
    echo "Hello, $name!"
    echo "Goodbye, $name!"
done
```

File Operations

Directory Existence Check

```
mshell:
for dir in /home /etc /var /tmp
do
    if [ -d $dir ]; then
        echo "$dir exists"
    fi
done
```

File Existence Check

```
mshell:
for file in /etc/hosts /etc/passwd /etc/shadow
do
    if [ -f $file ]; then
        echo "$file exists"
    else
        echo "$file not found"
    fi
done
```

Path Existence Check

```
mshell:
for path in /home /etc /nonexistent
do
    if [ -e $path ]; then
        echo "$path exists"
```

```
    fi
done
```

Edge Cases

Empty List

```
mshell:
empty=""
for item in $empty
do
    echo "This won't print"
done
# Loop executes 0 times
```

Single Item

```
mshell:
for item in only
do
    echo "Item: $item"
done
# Loop executes 1 time
```

Large Lists

```
mshell:
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
do
    echo "Number: $i"
done
# Works fine with many items
```

What Works

 WORKS:

- Literal lists of words
- Numbers (integers and floats)
- Mixed data types
- Variable expansion (`$variable`)
- IF statements inside FOR
- IF-ELSE statements inside FOR
- Multiple commands per iteration
- String comparisons
- Numeric comparisons
- File/directory tests
- Counter operations with `incr`
- Empty lists (0 iterations)
- Single item lists
- Large lists (tested with 10+ items)

Common Patterns

Pattern 1: Counting Items

```
mshell:
count=0
for item in list1 list2 list3
do
    incr count
done
echo "Total items: $count"
```

Pattern 2: Filtering with Conditions

```
mshell:
found=0
for num in 1 2 3 4 5 6 7 8 9 10
do
    if [ $num > 5 ]; then
        echo "Found: $num"
        incr found
```

```
    fi
done
echo "Found $found items"
```

Pattern 3: Processing with Status

```
mshell:
for file in file1 file2 file3
do
    if [ -f $file ]; then
        echo "Processing $file..."
        # do work
        echo "Completed $file"
    else
        echo "Skipping $file (not found)"
    fi
done
```

Pattern 4: String Processing

```
mshell:
text="Hello World from mshell"
for word in $text
do
    echo "Word: $word"
done
```

Best Practices

1. Use Descriptive Variable Names

```
mshell:
# GOOD
for username in john jane bob
do
```

```
    echo "User: $username"
done

# BAD
for u in john jane bob
do
    echo "User: $u"
done
```

2. Always Use Proper Spacing

mshell:

```
# CORRECT
for item in list
do
    command
done

# AVOID (may not parse correctly)
for item in list;do command;done
```

3. Use Quotes for Variable Expansion with Spaces

mshell:

```
# SAFE - will split by spaces
list="red green blue"
for color in $list
do
    echo "$color"
done
```

4. Initialize Counters Before the Loop

mshell:

```
# GOOD
count=0
```

```
for item in a b c
do
    incr count
done

# BAD - undefined behavior
for item in a b c
do
    incr count  # count not initialized
done
```

5. Test File Existence Before Operations

```
mshell:
for file in /etc/hosts /etc/passwd /etc/nofile
do
    if [ -f $file ]; then
        echo "Processing $file"
        # operations here
    fi
done
```

Common Mistakes

1. Forgetting $ When Accessing Variable

```
mshell:
# WRONG
for num in 1 2 3
do
    echo "Number: num"  # prints "num" literally
done

# CORRECT
```

```
for num in 1 2 3
do
    echo "Number: $num"  # prints the value
done
```

2. Using $ in Variable Declaration

mshell:

```
# WRONG
for $num in 1 2 3  # syntax error
do
    echo $num
done

# CORRECT
for num in 1 2 3  # no $ here
do
    echo $num  # $ here when accessing
done
```

3. Not Initializing Counters

mshell:

```
# WRONG - counter not initialized
for item in a b c
do
    incr count  # undefined
done

# CORRECT
count=0
for item in a b c
do
    incr count
done
```

4. Expecting Nested Loops to Work

mshell:

```
# DOES NOT WORK
for letter in A B C
do
   for num in 1 2 3
   do
      echo "$letter$num"  # inner variable may not work
   done
done
```

5. Trying to Use Wildcards

mshell:

```
# DOES NOT WORK
for file in *.txt
do
    echo $file
done

# WORKAROUND - list files explicitly
for file in file1.txt file2.txt file3.txt
do
    echo $file
done
```

Limitations

No Nested Loops

mshell:

```
# CANNOT DO THIS
for i in A B C
do
    for j in 1 2 3
```

```
    do
        echo "$i$j"
    done
done
```

Workaround: Use single loop with combined values

```
mshell:
for item in A1 A2 A3 B1 B2 B3 C1 C2 C3
do
    echo "$item"
done
```

No Command Substitution

```
mshell:
# CANNOT DO THIS
for i in $(seq 1 5)
do
    echo $i
done
```

Workaround: List numbers explicitly

```
mshell:
for i in 1 2 3 4 5
do
    echo $i
done
```

No Wildcards

```
mshell:
# CANNOT DO THIS
for file in *.ms
do
    echo $file
done
```

Workaround: List files explicitly or use external command

Performance Tips

1. Minimize Loop Iterations

mshell:

```
# BETTER - iterate only what you need
for i in 1 2 3
do
    process $i
done

# WORSE - unnecessary iterations
for i in 1 2 3 4 5 6 7 8 9 10
do
    if [ $i -le 3 ]; then
        process $i
    fi
done
```

2. Use Conditions Efficiently

mshell:

```
# BETTER - early filtering
for file in file1 file2 file3
do
    if [ -f $file ]; then
        process $file
    fi
done
```

Comparison: FOR vs WHILE

Use FOR when:

- You have a list of items to iterate
- You know the items to process
- You want to iterate over words/numbers
- You need simpler syntax for lists
- You want to split a string into words

Use WHILE when:

- You need condition-based looping
- You don't know iteration count beforehand
- You want to loop until a condition changes
- You need fine control over increment
- You're counting with numeric comparisons

FOR Example:

```
mshell:
for num in 1 2 3 4 5
do
    echo "Count: $num"
done
```

WHILE Example:

```
mshell:
counter=0
while [ $counter -lt 5 ]
do
    echo "Count: $counter"
    incr counter
done
```

Testing Your FOR Loops

Use the provided test scripts to verify functionality:

```
mshell:
# Run comprehensive test suite
```

./test_for_loops.ms

 # Run examples

./for_examples.ms

Summary

Key Points:

1. Use `for variable in list` format
2. Always use `$` when accessing the loop variable
3. Use `do` and `done` to mark loop body
4. Works with numbers, strings, and mixed types
5. Can contain IF statements and multiple commands
6. Initialize counters before the loop

Most Common Format:

mshell:

```
for item in value1 value2 value3
do
    command $item
done
```

With Counter:

mshell:

```
count=0
for item in list
do
    incr count
    echo "$count: $item"
done
```

With Condition:

mshell:

```
for item in list
```

```
do
    if [ condition ]; then
        command $item
    fi
done
```

With Variable Expansion:

```
mshell:
items="apple banana cherry"
for item in $items
do
    echo "Item: $item"
done
```

WHILE Loop Guide for mshell

Overview

This guide explains how to write WHILE loops in mshell, what works, best practices, and common use cases.

Basic Syntax

Standard WHILE loop

```
mshell:
while [ condition ]
```

```
do
    command1
    command2
done
```

Components:

- `while` - keyword to start the loop
- `[ condition ]` - test condition (must be in square brackets)
- `do` - marks start of loop body
- `done` - marks end of loop

Condition Operators

Numeric Comparison Operators

-lt (less than)

mshell:

```
i=1
while [ $i -lt 5 ]
do
    echo "i = $i"
    incr i
done
# Output: 1, 2, 3, 4
```

-le (less than or equal)

mshell:

```
i=1
while [ $i -le 5 ]
do
    echo "i = $i"
    incr i
done
# Output: 1, 2, 3, 4, 5
```

-gt (greater than)

mshell:

```
i=10
while [ $i -gt 5 ]
do
    echo "i = $i"
    # DOESN”T WORK Note: no decrement operator at WHILE
done
```

-ge (greater than or equal)

mshell:

```
i=10
while [ $i -ge 5 ]
do
    echo "i = $i"
    # DOESN”T WORK Note: no decrement operator at WHILE
done
```

-eq (equal)

mshell:

```
running=1
while [ $running -eq 1 ]
do
    echo "Running..."
    # Change running to 0 to stop (requires IF)
done
```

-ne (not equal)

mshell:

```
i=1
while [ $i -ne 5 ]
do
    echo "i = $i"
    incr i
done
# Output: 1, 2, 3, 4
```

Symbol Operators

< (less than)

mshell:

```
count=1
while [ $count < 6 ]
do
    echo "Count: $count"
    incr count
done
# Output: 1, 2, 3, 4, 5
```

<= (less than or equal)

mshell:

```
n=1
while [ $n <= 5 ]
do
    echo "n = $n"
    incr n
done
# Output: 1, 2, 3, 4, 5
```

> (greater than)

mshell:

```
# Use with caution - needs decrement
```

>= (greater than or equal)

mshell:

```
# Use with caution - needs decrement
```

String Comparison Operators

= (string equality)

mshell:

```
status=active
count=0
while [ $status = active ]
do
```

```
    incr count
    # Need to change status to stop
    # WARNING: Can cause infinite loop without IF
done
```

!= (string inequality)

mshell:

```
name=start
count=0
while [ $name != stop ]
do
    incr count
    # Need to change name to stop
    # WARNING: Can cause infinite loop without IF
done
```

Loop Control with incr

Basic Counter

mshell:

```
i=1
while [ $i -le 5 ]
do
    echo "Count: $i"
    incr i
done
```

Starting from Zero

mshell:

```
i=0
while [ $i -lt 5 ]
do
    echo "Count: $i"
```

```
    incr i
done
# Output: 0, 1, 2, 3, 4
```

Starting from Any Value

```
mshell:
i=10
while [ $i -lt 13 ]
do
    echo "Count: $i"
    incr i
done
# Output: 10, 11, 12
```

Comparing Variables

Two Variables

```
mshell:
start=1
end=5
while [ $start < $end ]
do
    echo "start=$start, end=$end"
    incr start
done
```

Using Limit Variable

```
mshell:
counter=0
limit=5
while [ $counter -lt $limit ]
do
```

```
    echo "Counter: $counter"
    incr counter
done
```

Multiple Commands per Iteration

Two Commands

```
mshell:
i=1
while [ $i -le 3 ]
do
    echo "Processing..."
    echo "Working..."
    incr i
done
```

Multiple Counters

```
mshell:
i=1
count1=0
count2=0
while [ $i -le 3 ]
do
    incr count1
    incr count2
    incr i
done
# Both count1 and count2 will be 3
```

Edge Cases

Zero Iterations

```
mshell:
i=10
count=0
while [ $i -lt 5 ]
do
    incr count
    incr i
done
# count will be 0 - loop never executes
```

Single Iteration

```
mshell:
n=1
while [ $n -lt 2 ]
do
    echo "Once"
    incr n
done
# Executes exactly once
```

Many Iterations

```
mshell:
i=1
while [ $i -le 100 ]
do
    echo "Iteration $i"
    incr i
done
# Executes 100 times
```

What Works

 WORKS:

- Numeric comparisons: `-lt`, `-le`, `-gt`, `-ge`, `-eq`, `-ne`
- Symbol comparisons: `<`, `<=`, `>`, `>=`
- String comparisons: `=`, `!=`
- Variable comparisons: `[ $var1 < $var2 ]`
- Multiple commands per iteration
- Counter with `incr`
- Zero iterations (condition false from start)
- Single iteration
- Many iterations (tested with 10+)

Common Patterns

Pattern 1: Simple Counter

```
mshell:
i=1
while [ $i -le 10 ]
do
    echo "Count: $i"
    incr i
done
```

Pattern 2: Processing with Limit

```
mshell:
counter=0
limit=5
while [ $counter -lt $limit ]
do
    echo "Processing item $counter"
    incr counter
done
```

Pattern 3: Comparing Two Variables

```
mshell:
current=1
target=10
```

```
while [ $current < $target ]
do
    echo "Progress: $current/$target"
    incr current
done
```

Pattern 4: Multiple Operations

```
mshell:
i=1
total=0
while [ $i -le 5 ]
do
    echo "Adding $i"
    total=$((total + i))  # If arithmetic works
    incr i
done
```

Best Practices

1. Always Increment the Counter

```
mshell:
# GOOD - will terminate
i=1
while [ $i -le 5 ]
do
    echo "Count: $i"
    incr i
done

# BAD - infinite loop!
i=1
while [ $i -le 5 ]
```

```
do
    echo "Count: $i"
    # Forgot incr i - INFINITE LOOP!
done
```

2. Initialize Variables Before Loop

```
mshell:
# GOOD
counter=0
while [ $counter -lt 5 ]
do
    incr counter
done

# BAD - undefined behavior
while [ $counter -lt 5 ]
do
    incr counter
done
```

3. Use Clear Condition

```
mshell:
# GOOD - clear condition
i=1
while [ $i -le 10 ]
do
    incr i
done

# CONFUSING - hard to read
while [ $x -ne 99 ]
do
    incr x
```

```
done
```

4. Avoid Infinite Loops

mshell:

```
# DANGEROUS - can loop forever
status=running
while [ $status = running ]
do
    echo "Running..."
    # Need way to change status!
done

# SAFER - use counter limit
status=running
count=0
while [ $count -lt 100 ]
do
    echo "Running..."
    incr count
    # Will stop after 100 iterations max
done
```

5. Use Appropriate Comparison

mshell:

```
# GOOD - use -lt for "less than"
i=0
while [ $i -lt 5 ]
do
    incr i
done

# ALSO GOOD - use < symbol
i=0
```

```
while [ $i < 5 ]
do
    incr i
done
```

Common Mistakes

1. Forgetting to Increment

mshell:

```
# WRONG - infinite loop
i=1
while [ $i -le 5 ]
do
    echo "Count: $i"
    # FORGOT: incr i
done

# CORRECT
i=1
while [ $i -le 5 ]
do
    echo "Count: $i"
    incr i
done
```

2. Wrong Initial Value

mshell:

```
# WRONG - won't execute
i=10
while [ $i -lt 5 ]
do
    echo "This never prints"
```

```
    incr i
done

# CORRECT
i=1
while [ $i -lt 5 ]
do
    echo "This prints"
    incr i
done
```

3. Using Undefined Variables

mshell:

```
# WRONG
while [ $count -lt 5 ]
do
    echo "Count: $count"
    incr count
done

# CORRECT
count=0
while [ $count -lt 5 ]
do
    echo "Count: $count"
    incr count
done
```

4. Trying to Use IF Inside WHILE

mshell:

```
# WRONG
count=0
while [ $count -lt 10 ]
```

```
do
    if [ $count -ge 5 ]; then
        echo "Big number"
    fi
    incr count
done
# Variables inside IF don't update correctly
```

5. Forgetting $ in Condition

```
mshell:
# WRONG
i=1
while [ i -lt 5 ]
do
    incr i
done

# CORRECT
i=1
while [ $i -lt 5 ]
do
    incr i
done
```

Performance Tips

1. Minimize Iterations

```
mshell:
# BETTER - only iterate what you need
i=0
while [ $i < 10 ]
do
```

```
    process $i
    incr i
done

# WORSE - unnecessary iterations
i=0
while [ $i < 1000 ]
do
    if [ $i < 10 ]; then
        process $i
    fi
    incr i
done
```

2. Use Appropriate Operators

mshell:

```
# GOOD - simple comparison
i=0
while [ $i -lt 10 ]
do
    incr i
done
```

Testing Your WHILE Loops

Use the provided test scripts to verify functionality:

mshell:

```
# Run comprehensive test suite
./test_while_loops.ms

# Run examples
./while_examples.ms
```

Comparison: WHILE vs FOR

Use WHILE when:

- You need condition-based looping
- You don't know iteration count beforehand
- You want to loop until a condition changes
- You need fine control over increment

Use FOR when:

- You have a list of items to iterate
- You know the items to process
- You want to iterate over words/numbers
- You need simpler syntax for lists

WHILE Example:

```
mshell:
counter=0
while [ $counter -lt 5 ]
do
    echo "Count: $counter"
    incr counter
done
```

FOR Example:

```
mshell:
for num in 1 2 3 4 5
do
    echo "Count: $num"
done
```

Summary

Key Points:

1. Use `while [ condition ]` format
2. Always use `$` when accessing variables
3. Use `do` and `done` to mark loop body
4. Always increment counter with `incr`
5. Initialize all variables before the loop
6. Avoid infinite loops

Most Common Format:

```
mshell:
i=1
while [ $i -le 10 ]
do
    command $i
    incr i
done
```

With Two Variables:

```
mshell:
counter=0
limit=10
while [ $counter -lt $limit ]
do
    command $counter
    incr counter
done
```

Safeguard Against Infinite Loop:

```
mshell:
max_iterations=1000
counter=0
while [ $counter -lt $max_iterations ]
do
    # your code here
    incr counter
done
```

FOR Loop Guide for mshell

Overview

This guide explains how to write FOR loops in mshell, what works, best practices, and common use cases.

Basic Syntax

Standard FOR loop

```
mshell:
for variable in list
do
    command1
    command2
done
```

Components:

- `for` - keyword to start the loop
- `variable` - loop variable name (without $)
- `in` - keyword separator
- `list` - space-separated list of items
- `do` - marks start of loop body
- `done` - marks end of loop

List Types

1. Literal List of Words

mshell:

```
for fruit in apple banana cherry
do
    echo "Fruit: $fruit"
done
# Output: apple, banana, cherry
```

2. Numbers

mshell:

```
for num in 1 2 3 4 5
do
    echo "Number: $num"
done
# Output: 1, 2, 3, 4, 5
```

3. Mixed Data Types

mshell:

```
for item in 1 hello 3.14 world 5
do
    echo "Item: $item"
done
# Output: 1, hello, 3.14, world, 5
```

4. Variable Expansion

mshell:

```
colors="red green blue yellow"
for color in $colors
do
    echo "Color: $color"
done
# Output: red, green, blue, yellow
```

5. File Paths

mshell:

```
for dir in /home /etc /var /tmp
do
    echo "Directory: $dir"
done
```

Loop Variable

Accessing the Variable

Always use `$` when referencing the loop variable:

mshell:
```
for name in John Jane Bob
do
    echo "Hello, $name!"  # CORRECT - with $
done
```

mshell:
```
for name in John Jane Bob
do
    echo "Hello, name!"   # WRONG - prints "name" literally
done
```

Variable Type Detection

The loop variable automatically detects type:

mshell:
```
for item in 1 hello 3.14
do
    # item is numeric for 1 and 3.14
    # item is string for "hello"
    echo "$item"
done
```

FOR with IF Conditions

Basic IF inside FOR

```
mshell:
for num in 1 2 3 4 5 6 7 8 9 10
do
    if [ $num > 5 ]; then
        echo "$num is greater than 5"
    fi
done
# Output: 6, 7, 8, 9, 10
```

IF-ELSE inside FOR

```
mshell:
for num in 1 2 3 4 5
do
    if [ $num > 3 ]; then
        echo "$num is big"
    else
        echo "$num is small"
    fi
done
```

String Comparison

```
mshell:
for status in active inactive pending active
do
    if [ $status = active ]; then
        echo "Status is ACTIVE"
    fi
done
```

File/Directory Tests

mshell:

```
for path in /home /etc /var /nonexistent
do
    if [ -d $path ]; then
        echo "$path exists"
    else
        echo "$path not found"
    fi
done
```

Multiple Commands per Iteration

mshell:

```
for i in 1 2 3
do
    echo "Processing item $i"
    echo "Item $i completed"
    echo "---"
done
```

Using Counters

Simple Counter with incr

mshell:

```
count=0
for item in apple banana cherry
do
    incr count
    echo "$count: $item"
done
# Output: 1: apple, 2: banana, 3: cherry
```

Multiple Counters

```
mshell:
total=0
processed=0
for file in file1 file2 file3
do
    incr total
    if [ -f $file ]; then
        incr processed
    fi
done
echo "Processed $processed out of $total files"
```

String Processing

Processing Sentence into Words

```
mshell:
sentence="The quick brown fox jumps"
for word in $sentence
do
    echo "Word: $word"
done
# Output: The, quick, brown, fox, jumps
```

String Operations

```
mshell:
for name in John Jane Bob Alice
do
    echo "Hello, $name!"
    echo "Goodbye, $name!"
done
```

File Operations

Directory Existence Check

mshell:

```
for dir in /home /etc /var /tmp
do
    if [ -d $dir ]; then
        echo "$dir exists"
    fi
done
```

File Existence Check

mshell:

```
for file in /etc/hosts /etc/passwd /etc/shadow
do
    if [ -f $file ]; then
        echo "$file exists"
    else
        echo "$file not found"
    fi
done
```

Path Existence Check

mshell:

```
for path in /home /etc /nonexistent
do
    if [ -e $path ]; then
        echo "$path exists"
    fi
done
```

Edge Cases

Empty List

mshell:

```
empty=""
for item in $empty
do
    echo "This won't print"
done
# Loop executes 0 times
```

Single Item

mshell:

```
for item in only
do
    echo "Item: $item"
done
# Loop executes 1 time
```

Large Lists

mshell:

```
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
do
    echo "Number: $i"
done
# Works fine with many items
```

What Works

✅ **WORKS:**

- Literal lists of words
- Numbers (integers and floats)
- Mixed data types
- Variable expansion (`$variable`)
- IF statements inside FOR
- IF-ELSE statements inside FOR
- Multiple commands per iteration
- String comparisons

- Numeric comparisons
- File/directory tests
- Counter operations with `incr`
- Empty lists (0 iterations)
- Single item lists
- Large lists (tested with 10+ items)

Common Patterns

Pattern 1: Counting Items

```
mshell:
count=0
for item in list1 list2 list3
do
    incr count
done
echo "Total items: $count"
```

Pattern 2: Filtering with Conditions

```
mshell:
found=0
for num in 1 2 3 4 5 6 7 8 9 10
do
    if [ $num > 5 ]; then
        echo "Found: $num"
        incr found
    fi
done
echo "Found $found items"
```

Pattern 3: Processing with Status

```
mshell:
for file in file1 file2 file3
do
```

```
    if [ -f $file ]; then
        echo "Processing $file..."
        # do work
        echo "Completed $file"
    else
        echo "Skipping $file (not found)"
    fi
done
```

Pattern 4: String Processing

```
mshell:
text="Hello World from mshell"
for word in $text
do
    echo "Word: $word"
done
```

Best Practices

1. Use Descriptive Variable Names

```
mshell:
# GOOD
for username in john jane bob
do
    echo "User: $username"
done

# BAD
for u in john jane bob
do
    echo "User: $u"
done
```

2. Always Use Proper Spacing

mshell:

```
# CORRECT
for item in list
do
    command
done

# AVOID (may not parse correctly)
for item in list;do command;done
```

3. Use Quotes for Variable Expansion with Spaces

mshell:

```
# SAFE - will split by spaces
list="red green blue"
for color in $list
do
    echo "$color"
done
```

4. Initialize Counters Before the Loop

mshell:

```
# GOOD
count=0
for item in a b c
do
    incr count
done

# BAD - undefined behavior
for item in a b c
do
    incr count  # count not initialized
```

```
done
```

5. Test File Existence Before Operations

mshell:

```
for file in /etc/hosts /etc/passwd /etc/nofile
do
    if [ -f $file ]; then
        echo "Processing $file"
        # operations here
    fi
done
```

Common Mistakes

1. Forgetting $ When Accessing Variable

mshell:

```
# WRONG
for num in 1 2 3
do
    echo "Number: num"  # prints "num" literally
done

# CORRECT
for num in 1 2 3
do
    echo "Number: $num"  # prints the value
done
```

2. Using $ in Variable Declaration

mshell:

```
# WRONG
for $num in 1 2 3  # syntax error
```

```
do
    echo $num
done

# CORRECT
for num in 1 2 3  # no $ here
do
    echo $num  # $ here when accessing
done
```

3. Not Initializing Counters

```
mshell:
# WRONG - counter not initialized
for item in a b c
do
    incr count  # undefined
done

# CORRECT
count=0
for item in a b c
do
    incr count
done
```

Performance Tips

1. Minimize Loop Iterations

```
mshell:
# BETTER - iterate only what you need
for i in 1 2 3
do
```

```
    process $i
done

# WORSE - unnecessary iterations
for i in 1 2 3 4 5 6 7 8 9 10
do
    if [ $i -le 3 ]; then
        process $i
    fi
done
```

2. Use Conditions Efficiently

```
mshell:
# BETTER - early filtering
for file in file1 file2 file3
do
    if [ -f $file ]; then
        process $file
    fi
done
```

Comparison: FOR vs WHILE

Use FOR when:

- You have a list of items to iterate
- You know the items to process
- You want to iterate over words/numbers
- You need simpler syntax for lists
- You want to split a string into words

Use WHILE when:

- You need condition-based looping
- You don't know iteration count beforehand
- You want to loop until a condition changes

- You need fine control over increment
- You're counting with numeric comparisons

FOR Example:

```
mshell:
for num in 1 2 3 4 5
do
    echo "Count: $num"
done
```

WHILE Example:

```
mshell:
counter=0
while [ $counter -lt 5 ]
do
    echo "Count: $counter"
    incr counter
done
```

Testing Your FOR Loops

Use the provided test scripts to verify functionality:

```
mshell:
# Run comprehensive test suite
./test_for_loops.ms

# Run examples
./for_examples.ms
```

Summary

Key Points:

1. Use `for variable in list` format
2. Always use `$` when accessing the loop variable
3. Use `do` and `done` to mark loop body
4. Works with numbers, strings, and mixed types
5. Can contain IF statements and multiple commands
6. Initialize counters before the loop

Most Common Format:

```
mshell:
for item in value1 value2 value3
do
    command $item
done
```

With Counter:

```
mshell:
count=0
for item in list
do
    incr count
    echo "$count: $item"
done
```

With Condition:

```
mshell:
for item in list
do
    if [ condition ]; then
        command $item
    fi
done
```

With Variable Expansion:

```
mshell:
items="apple banana cherry"
```

```
for item in $items
do
    echo "Item: $item"
done
```

IF Statement Guide for mshell

Overview

This guide explains how to write IF statements in mshell, what works, best practices, and common use cases.

Basic Syntax

Simple IF statement

```
mshell:
if [ condition ]
then
    command1
    command2
fi
```

IF-ELSE statement

```
mshell:
if [ condition ]
then
    command1
else
    command2
```

```
fi
```

IF-ELIF-ELSE statement

mshell:

```
if [ condition1 ]
then
    command1
elif [ condition2 ]
then
    command2
else
    command3
fi
```

Components:

- `if` - keyword to start the statement
- `[ condition ]` - test condition (must be in square brackets)
- `then` - marks start of true branch
- `else` - optional, marks start of false branch
- `elif` - optional, additional condition
- `fi` - marks end of IF statement

Condition Operators

Numeric Comparison Operators

-lt (less than)

mshell:

```
num=3
if [ $num -lt 5 ]; then
    echo "$num is less than 5"
fi
```

-le (less than or equal)

mshell:

```
num=5
if [ $num -le 5 ]; then
    echo "$num is less than or equal to 5"
fi
```

-gt (greater than)

mshell:

```
num=7
if [ $num -gt 5 ]; then
    echo "$num is greater than 5"
fi
```

-ge (greater than or equal)

mshell:

```
num=5
if [ $num -ge 5 ]; then
    echo "$num is greater than or equal to 5"
fi
```

-eq (equal)

mshell:

```
num=5
if [ $num -eq 5 ]; then
    echo "$num equals 5"
fi
```

-ne (not equal)

mshell:

```
num=3
if [ $num -ne 5 ]; then
    echo "$num does not equal 5"
fi
```

Symbol Operators

< (less than)

mshell:

```
num=3
if [ $num < 5 ]; then
```

```
    echo "$num is less than 5"
fi
```

<= (less than or equal)

mshell:

```
num=5
if [ $num <= 5 ]; then
    echo "$num is less than or equal to 5"
fi
```

> (greater than)

mshell:

```
num=7
if [ $num > 5 ]; then
    echo "$num is greater than 5"
fi
```

>= (greater than or equal)

mshell:

```
num=5
if [ $num >= 5 ]; then
    echo "$num is greater than or equal to 5"
fi
```

String Comparison Operators

= (string equality)

mshell:

```
name="alice"
if [ $name = alice ]; then
    echo "Hello Alice!"
fi
```

!= (string inequality)

mshell:

```
status="active"
if [ $status != inactive ]; then
    echo "Status is not inactive"
fi
```

-z (string is empty)

mshell:

```
text=""
if [ -z $text ]; then
    echo "Text is empty"
fi
```

-n (string is not empty)

mshell:

```
text="hello"
if [ -n $text ]; then
    echo "Text is not empty"
fi
```

File Test Operators

-f (file exists and is regular file)

mshell:

```
if [ -f /etc/hosts ]; then
    echo "File exists"
fi
```

-e (path exists)

mshell:

```
if [ -e /etc ]; then
    echo "Path exists"
fi
```

Comparing Variables

Two Numeric Variables

mshell:

```
a=5
b=3
if [ $a > $b ]; then
    echo "$a is greater than $b"
```

```
fi
```

Two String Variables

```
mshell:
name1="alice"
name2="bob"
if [ $name1 = $name2 ]; then
    echo "Names are equal"
else
    echo "Names are different"
fi
```

Variable with Literal

```
mshell:
status="active"
if [ $status = active ]; then
    echo "System is active"
fi
```

Multiple Commands

Multiple Commands in THEN

```
mshell:
if [ $num > 5 ]; then
    echo "Number is big"
    echo "Processing..."
    echo "Done"
fi
```

Multiple Commands in ELSE

```
mshell:
if [ $num > 5 ]; then
    echo "Big number"
```

```
else
    echo "Small number"
    echo "Need more"
fi
```

IF-ELSE Patterns

Simple IF-ELSE

mshell:

```
age=20
if [ $age >= 18 ]; then
    echo "Adult"
else
    echo "Minor"
fi
```

IF-ELIF-ELSE

mshell:

```
score=85
if [ $score >= 90 ]; then
    echo "Grade: A"
elif [ $score >= 80 ]; then
    echo "Grade: B"
elif [ $score >= 70 ]; then
    echo "Grade: C"
else
    echo "Grade: F"
fi
```

Multiple ELIF

mshell:

```
temp=25
```

```
if [ $temp < 0 ]; then
    echo "Freezing"
elif [ $temp < 10 ]; then
    echo "Cold"
elif [ $temp < 20 ]; then
    echo "Cool"
elif [ $temp < 30 ]; then
    echo "Warm"
else
    echo "Hot"
fi
```

Edge Cases

Empty String

mshell:

```
text=""
if [ -z $text ]; then
    echo "Empty"
fi
```

Zero Value

mshell:

```
num=0
if [ $num -eq 0 ]; then
    echo "Zero"
fi
```

Negative Numbers

mshell:

```
num=-5
if [ $num < 0 ]; then
```

```
    echo "Negative"
fi
```

What Works

✅ WORKS:

- Numeric comparisons: `-lt`, `-le`, `-gt`, `-ge`, `-eq`, `-ne`
- Symbol comparisons: `<`, `<=`, `>`, `>=`
- String comparisons: `=`, `!=`, `-z`, `-n`
- File tests: `-f`, `-d`, `-e`, `-r`, `-w`, `-x`
- Variable comparisons
- IF-ELSE statements
- IF-ELIF-ELSE statements
- Multiple ELIF branches
- Multiple commands in THEN/ELSE
- Nested IF (with limitations)

Common Patterns

Pattern 1: Range Check

mshell:

```
age=25
lowerage=18
highage=65
range_ok=1
if [ $age < $lowerage ]; then
      range_ok=0
    fi
 if [ $age > $highage ]; then
       range_ok=0
    fi
if [ $range_ok -eq 1 ]; then
    echo "Working age"
  fi
```

Pattern 2: Status Check

mshell:

```
status="active"
if [ $status = active ]; then
    echo "System running"
else
    echo "System stopped"
fi
```

Pattern 3: File Processing

mshell:

```
file="/etc/hosts"
if [ -f $file ]; then
    echo "Processing $file"
    # process file
else
    echo "File not found: $file"
fi
```

Pattern 4: Grade Calculator

mshell:

```
score=85
grade=F
lock=0


# A: score >= 90  (i.e., NOT(score < 90))
gate=1
if [ $lock -ne 0 ]; then
    gate=0
```

```
fi
if [ $score < 90 ]; then
    gate=0
fi
if [ $gate -eq 1 ]; then
    grade=A
    lock=1
fi

# B: score >= 80
gate=1
if [ $lock -ne 0 ]; then
    gate=0
fi
if [ $score < 80 ]; then
    gate=0
fi
if [ $gate -eq 1 ]; then
    grade=B
    lock=1
fi

# C: score >= 70
gate=1
if [ $lock -ne 0 ]; then
```

```
    gate=0
fi
if [ $score < 70 ]; then
    gate=0
fi
if [ $gate -eq 1 ]; then
    grade=C
    lock=1
fi
echo Grade: $grade
# Answer: Set score = 85 (numeric)
Grade: B
```

Best Practices

1. Use Clear Conditions

mshell:

```
# GOOD - clear and readable
if [ $age >= 18 ]; then
    echo "Adult"
fi

# BAD - confusing
if [ $x -ne 0 ]; then
    if [ $y -eq 1 ]; then
        # what does this mean?
    fi
fi
```

2. Always Use Spaces Around Brackets

mshell:

```
# CORRECT
if [ $num > 5 ]; then
    echo "Yes"
fi

# WRONG - may not parse
if [$num > 5]; then
    echo "Yes"
fi
```

3. Use Descriptive Variable Names

mshell:

```
# GOOD
if [ $is_valid = true ]; then
    process_data
fi

# BAD
if [ $x = 1 ]; then
    do_something
fi
```

4. Handle Both Cases

mshell:

```
# GOOD - explicit handling
if [ $status = success ]; then
    echo "Success"
else
    echo "Failed"
fi
```

```
# ACCEPTABLE but less clear
if [ $status = success ]; then
    echo "Success"
fi
# What happens if not success?
```

5. Use ELIF Instead of Nested IF When Possible

mshell:

```
# BETTER - flat structure
if [ $score >= 90 ]; then
    echo "A"
elif [ $score >= 80 ]; then
    echo "B"
elif [ $score >= 70 ]; then
    echo "C"
fi

# WORSE - nested
if [ $score >= 90 ]; then
    echo "A"
else
    if [ $score >= 80 ]; then
        echo "B"
    else
        if [ $score >= 70 ]; then
            echo "C"
        fi
    fi
fi
```

Common Mistakes

1. Forgetting $ When Accessing Variables

mshell:

```
# WRONG
if [ num > 5 ]; then
    echo "Big"
fi

# CORRECT
if [ $num > 5 ]; then
    echo "Big"
fi
```

2. Missing Spaces Around Brackets

mshell:

```
# WRONG
if [$num > 5]; then
    echo "Big"
fi

# CORRECT
if [ $num > 5 ]; then
    echo "Big"
fi
```

3. Forgetting THEN

mshell:

```
# WRONG
if [ $num > 5 ]
    echo "Big"
fi

# CORRECT
```

```
if [ $num > 5 ]; then
    echo "Big"
fi
```

4. Forgetting FI

mshell:

```
# WRONG
if [ $num > 5 ]; then
    echo "Big"
# Missing fi

# CORRECT
if [ $num > 5 ]; then
    echo "Big"
fi
```

Performance Tips

1. Order Conditions by Likelihood

mshell:

```
# BETTER - check most likely first
if [ $status = success ]; then
    # most common case
elif [ $status = pending ]; then
    # less common
elif [ $status = error ]; then
    # rare
fi
```

2. Use Early Returns When Possible

mshell:

```
# BETTER - early exit
```

```
if [ ! -f $file ]; then
    echo "File not found"
    # exit early
fi
# Continue processing...
```

Comparison: IF vs Other Constructs

IF vs WHILE

- Use IF for: single condition check, decision making
- Use WHILE for: repeated actions, loops

IF vs FOR

- IF can be inside FOR
- FOR cannot be inside IF
- Use IF for conditions, FOR for iterations

IF-ELIF vs Multiple IF

mshell:

```
# ELIF - only one branch executes
if [ $x = 1 ]; then
    echo "One"
elif [ $x = 2 ]; then
    echo "Two"
fi

# Multiple IF - all are checked
if [ $x = 1 ]; then
    echo "One"
fi
if [ $x = 2 ]; then
    echo "Two"
fi
```

Testing Your IF Statements

Test all branches:

mshell:

```
# Test TRUE branch
num=10
if [ $num > 5 ]; then
    echo "PASS: True branch"
fi

# Test FALSE branch
num=3
if [ $num > 5 ]; then
    echo "FAIL"
else
    echo "PASS: False branch"
fi

# Test ELIF
num=7
if [ $num > 10 ]; then
    echo "FAIL"
elif [ $num > 5 ]; then
    echo "PASS: ELIF branch"
fi
```

Summary

Key Points:

1. Use `if [ condition ]; then` format

2. Always use `$` when accessing variables
3. Use `then`, `else`, `elif`, and `fi` keywords
4. Always include spaces around `[ ]`
5. Use ELIF for multiple conditions

Most Common Format:

mshell:

```
if [ condition ]; then
    command
fi
```

With ELSE:

mshell:

```
if [ condition ]; then
    command1
else
    command2
fi
```

With ELIF:

mshell:

```
if [ condition1 ]; then
    command1
elif [ condition2 ]; then
    command2
else
    command3
fi
```

Comparison Examples:

mshell:

```
# Numeric
if [ $num -lt 5 ]; then echo "Small"; fi
if [ $num > 5 ]; then echo "Big"; fi
```

```
# String
if [ $name = alice ]; then echo "Alice"; fi
if [ -z $text ]; then echo "Empty"; fi

# File
if [ -f $file ]; then echo "Exists"; fi
if [ -d $dir ]; then echo "Directory"; fi
```

Prepared: Igor Lukyanov, Art2Dec SoftLab, October 31st 2025.