Daksh Cookbook

Practical Recipes for Phase 0

What this book is

A practical collection of reusable Daksh programming patterns. Each recipe focuses on a small, real programming task and combines Phase 0 concepts in a readable way.

Scope

All recipes are intentionally based on the Phase 0 web-interpreter feature set and implemented standard library surface. Future or unfinished syntax is not required.

1. Basic Output

Pattern: Use **tell** whenever you need simple console-style output in the web interpreter.

```
tell "Hello, Daksh!"
tell "Programming language"
tell 10
```

2. Multiple Variables

Pattern: Useful when several related values need to be initialized together.

```
set name = "Manideep", age = 21, score = 95.5

tell name
tell age
tell score
```

3. Safe Constant

Pattern: Use **const** for values that should not be reassigned.

```
const pi = 3.14

tell pi
```

4. Dynamic Variable

Pattern: Daksh Phase 0 supports dynamic reassignment of a variable to different value types.

```
set value = 10
tell value

set value = "Daksh"
tell value

set value = true
tell value
```

5. Convert and Inspect

Pattern: Use conversion functions and **type()** when values arrive in a different representation.

```
set textNumber = "25"
set number = integer(textNumber)

tell number
tell type(number)
tell text(number)
```

6. Arithmetic Summary

Pattern: A compact arithmetic pattern for calculations and testing.

```
set a = 20
set b = 6

tell a + b
tell a - b
tell a * b
tell a / b
tell a % b
```

7. Power Calculation

Pattern: Use ****** for exponentiation.

```
set base = 2
set exponent = 5

set result = base ** exponent

tell result
```

8. Even or Odd

Pattern: The remainder operator provides a simple parity check.

```
set number = 17

if number % 2 == 0 {
tell "Even"
} else {
tell "Odd"
}
```

9. Grade Decision

Pattern: A reusable pattern for ordered condition ranges.

```
set average = 88

if average >= 90 {
tell "A"
} else if average >= 75 {
tell "B"
} else {
tell "C"
}
```

10. Logical Eligibility

Pattern: Combine comparisons with **&&** when multiple requirements must be true.

```
set age = 21
set hasId = true

if age >= 18 && hasId {
tell "Eligible"
} else {
tell "Not eligible"
}
```

11. Range Loop

Pattern: Use an inclusive ascending range for straightforward counting.

```
for i in 1..5 {
tell i
}
```

12. While Counter

Pattern: A basic condition-controlled loop using increment.

```
set i = 1

while i <= 5 {
tell i
i++
}
```

13. Break Search

Pattern: Use **break** when the loop has completed its purpose.

```
set target = 7

for i in 1..10 {
if i == target {
tell "Found"
break
}
}
```

14. Continue Filter

Pattern: Use **continue** to skip the current iteration.

```
for i in 1..10 {
if i % 2 == 0 {
continue
}

tell i
}
```

15. Nested Loops

Pattern: Nested loops are useful for grid-like or multi-level processing.

```
for row in 1..3 {
for column in 1..3 {
tell row
tell column
}
}
```

16. List Traversal

Pattern: Use **foreach** when you want values directly rather than indexes.

```
set numbers = [10, 20, 30, 40]

foreach number in numbers {
tell number
}
```

17. List Sum

Pattern: A standard accumulation pattern.

```
set numbers = [10, 20, 30, 40]
set total = 0

foreach number in numbers {
set total += number
}

tell total
```

18. List Search

Pattern: The list module provides a direct membership helper.

```
set numbers = [10, 20, 30, 40]

if list.contains(numbers, 30) {
tell "Found"
} else {
tell "Not found"
}
```

19. List Mutation

Pattern: Use list operations when a collection must change during a program.

```
set numbers = [10, 20, 30]

list.add(numbers, 40)
list.remove(numbers, 20)

tell numbers
```

20. Nested Object

Pattern: Objects can contain lists and other structured values.

```
set student = {
name: "Manideep",
marks: [85, 90, 95],
active: true
}

tell student.name
tell student.marks[1]
tell student.active
```

21. Nested Assignment

Pattern: Nested assignment lets you update a property or indexed value directly.

```
set student = {
name: "Manideep",
marks: [85, 90, 95]
}

set student.name = "Daksh"
set student.marks[1] = 99

tell student.name
tell student.marks
```

22. Function with Return

Pattern: Keep repeated calculations inside reusable functions.

```
function add(a, b) {
return a + b
}

set result = add(10, 20)
tell result
```

23. Function over a List

Pattern: Functions can receive collections and return computed values.

```
function totalMarks(marks) {
set total = 0

foreach mark in marks {
set total += mark
}

return total
}

set marks = [80, 90, 95]
tell totalMarks(marks)
```

24. Recursive Factorial

Pattern: Recursion works when each call moves toward a clear stopping condition.

```
function factorial(n) {
if n <= 1 {
return 1
}

return n * factorial(n - 1)
}

tell factorial(5)
```

25. Student Result

Pattern: A complete pattern combining functions, lists, loops, conditions, and objects.

```
function processStudent(name, marks) {
set total = 0

foreach mark in marks {
```

```
set total += mark
}

set average = total / list.length(marks)

if average >= 90 {
return {
name: name,
average: average,
grade: "A"
}
} else if average >= 75 {
return {
name: name,
average: average,
grade: "B"
}
} else {
return {
name: name,
average: average,
grade: "C"
}
}
}

set student = processStudent(
"Manideep",
[85, 90, 95]
)

tell student.name
tell student.average
tell student.grade
```

26. String Cleaning

Pattern: Trim unwanted surrounding whitespace before applying formatting.

```
import string

set name = " manideep "
set cleanName = string.upper(string.trim(name))

tell cleanName
```

27. String Validation

Pattern: Use string helpers for simple validation rules.

```
import string

set email = "user@example.com"

if string.contains(email, "@") {
tell "Looks valid"
} else {
tell "Check value"
}
```

28. Math Helpers

Pattern: Use the math module instead of manually implementing common numeric operations.

```
import math

tell math.sqrt(144)
tell math.abs(-25)
tell math.round(12.6)
tell math.max(10, 20)
tell math.min(10, 20)
```

29. Random Choice

Pattern: Choose one item from a list with the random module.

```
import random

set colors = ["red", "blue", "green"]

tell random.choice(colors)
```

30. Random Number

Pattern: Generate a random integer inside a chosen range.

```
import random

set value = random.integer(1, 100)

tell value
```

31. Date Report

Pattern: Use the time module for date/time information.

```
import time

tell time.today()
tell time.now()
tell time.dayName(time.today())
tell time.monthName(time.today())
```

32. Date Arithmetic

Pattern: Create future dates without manually calculating calendar changes.

```
import time

set today = time.today()
set future = time.addDays(today, 7)

tell today
tell future
```

33. Date Comparison

Pattern: Use explicit date comparison helpers.

```
import time

set first = time.today()
set second = time.addDays(first, 7)

tell time.isBefore(first, second)
tell time.isAfter(first, second)
```

34. Formatted Date

Pattern: Use a formatting pattern when a readable date/time representation is required.

```
import time

set current = time.nowDateTime()

tell time.format(current, "YYYY-MM-DD HH:mm:ss")
```

35. System Report

Pattern: The system module exposes browser-available system/runtime information.

```
import system

tell system.osName()
tell system.osVersion()
tell system.architecture()
tell system.language()
tell system.dakshVersion()
```

36. Online Check

Pattern: Use the runtime's online-status helper before network-dependent workflows.

```
import system

if system.isOnline() {
tell "Online"
```

```
} else {
tell "Offline"
}
```

37. Delay

Pattern: Use **system.sleep()** for an asynchronous delay in supported execution contexts.

```
import system

tell "Starting"
system.sleep(1)
tell "Finished
```

38. Input and Conversion

Pattern: A common pattern when input arrives as text and must be converted before arithmetic.

```
set value = input("Enter a number: ")
set number = integer(value)

tell number * 2
```

39. Number Analyzer

Pattern: Combine independent condition checks to build a small analyzer.

```
set number = 42

if number > 0 {
tell "Positive"
} else if number < 0 {
tell "Negative"
} else {
tell "Zero"
}

if number % 2 == 0 {
tell "Even"
} else {
tell "Odd"
}
```

40. Shopping Total

Pattern: A compact shopping-bill pattern using collection traversal and a condition.

```
set prices = [100, 250, 75, 50]
set total = 0

foreach price in prices {
set total += price
}

if total >= 400 {
set total -= 20
}

tell total
```

41. Expense Summary

Pattern: Accumulate values and derive a summary statistic.

```
set expenses = [250, 120, 500, 80]
set total = 0

foreach expense in expenses {
set total += expense
}

set average = total / list.length(expenses)

tell total
tell average
```

42. Inventory Update

Pattern: Nested assignment is useful for updating structured application data.

```
set product = {
name: "Keyboard",
quantity: 5
}

set product.quantity = 8

tell product.name
tell product.quantity
```

43. Student Collection

Pattern: A list of objects is a useful pattern for record-oriented programs.

```
set students = [
{ name: "A", score: 90 },
{ name: "B", score: 76 },
{ name: "C", score: 88 }
]

foreach student in students {
tell student.name
tell student.score
}
```

44. Utility Function

Pattern: Functions can wrap standard-library operations into application-specific helpers.

```
import math

function circleArea(radius) {
return math.pi * radius ** 2
}

tell circleArea(5)
```

45. Combined Text + List

Pattern: Combine collection traversal with string processing.

```
import string
import list

set names = [" alice", "bob ", " carol "]
```

```
foreach name in names {
set clean = string.upper(string.trim(name))
tell clean
}

tell list.length(names)
```

46. Random Quiz Score

Pattern: This is the starting pattern for larger quiz/game programs.

```
import random

set answers = ["A", "B", "C", "D"]
set chosen = random.choice(answers)

tell "Selected:"
tell chosen
```

47. Mini Student Report

Pattern: A reusable reporting pattern that combines multiple standard-library modules.

```
import string
import list
import math

function report(name, marks) {
set cleanName = string.upper(string.trim(name))
set total = 0

foreach mark in marks {
set total += mark
}

set average = total / list.length(marks)

return {
name: cleanName,
total: total,
average: math.round(average)
}
}

set result = report(" manideep ", [85, 90, 95])

tell result.name
tell result.total
tell result.average
```

48. Error-Aware Indexing Test

Pattern: During development, test valid indexes first and separately test invalid indexes to observe Daksh's runtime error behavior.

```
set numbers = [10, 20, 30]

tell numbers[0]
tell numbers[2]
```

49. Module Integration

Pattern: A compact example of several imported modules working together.

```
import math
import string
import list

set values = [4, 9, 16]

foreach value in values {
set root = math.sqrt(value)
tell root
}

set name = string.upper(string.trim(" Daksh "))
tell name
tell list.length(values)
```

50. Phase 0 Application Pattern

Pattern: Use this as a base architecture for larger Phase 0 programs.

```
import math
import string
import list

function process(name, values) {
set cleanName = string.upper(string.trim(name))
set total = 0

foreach value in values {
set total += value
}

set average = total / list.length(values)

return {
name: cleanName,
total: total,
average: math.round(average)
}
}

set result = process(" daksh ", [80, 90, 95])

tell result.name
tell result.total
tell result.average
```

Cookbook Design Patterns

Accumulate: start with a neutral value such as 0, then update it inside a loop.

Filter: combine a condition with continue when unwanted values should be skipped.

Search: stop early with break when the target is found.

Transform: process each list item and create a cleaned or derived value.

Record: use objects for structured values such as students, products, or expenses.

Reuse: move repeated logic into functions.

Compose: combine functions with standard-library helpers rather than placing everything in one block.

Validate: test normal values, boundaries, and invalid cases before considering a program complete.

Phase 0 Recipe Boundaries

This cookbook does not depend on unfinished/future features such as ternary syntax, pattern matching, bitwise operators, lambdas, finalized custom-error syntax, or a native compiler/runtime.

File recipes respect browser security: arbitrary filesystem inspection is not assumed, and file operations use browser-mediated APIs where supported.

Final Practice Challenge

Choose one of the following and build it without copying a complete recipe: Student Management, Shopping Cart, Expense Manager, Quiz Game, Number Game, Inventory Tracker, or a Personal Utility Toolkit.

Aim to combine at least five Phase 0 areas: variables, conditions, loops, functions, lists/objects, input/output, and standard-library modules.

End of Daksh Phase 0 Cookbook