The Language

1. Introduction

Suny is a lightweight scripting language designed to be **small but powerful**, combining the minimalism and efficiency of **Lua** with the clarity and readability of **Python**.

Suny is intended to be a language that is:

- **Small** in implementation, so it is easy to understand and port.
- **Powerful** in expressiveness, so developers can solve real problems without feeling restricted.
- **Beginner-friendly**, encouraging experimentation and rapid learning.

1.1 Origins and Motivation

Suny was created as an experiment: *Can one person mind design and build a language from scratch that is both minimal and useful?*

Many modern languages are large, with complex standard libraries, heavy runtime systems, and thousands of pages of documentation. While this makes them powerful, it also makes them intimidating to beginners and difficult to fully understand.

Lua inspired Suny by showing that a small, elegant core can be extremely flexible. Python inspired Suny with its philosophy of readability and simplicity. Suny combines both ideas: a minimal implementation with a syntax that feels natural and beginner-friendly.

1.2 Philosophy of Suny

The design of Suny is guided by three principles:

1. **Simplicity**
    - The syntax is minimal, with few keywords.
    - Code should be as close to “pseudocode” as possible.
2. **Clarity**
    - Programs should be easy to read and write.
    - Indentation and structure should make logic clear at first glance.
3. **Power from smallness**
    - Instead of a large standard library, Suny focuses on a small but flexible core.
    - Advanced features can be built from simple building blocks.
    - The VM and bytecode are simple, so the language can be embedded or extended easily.

1.3 Typical Uses

Suny is not meant to replace large general-purpose languages like C++ or Java. Instead, it is designed for:

- **Learning programming concepts**: because the syntax is clean and the language is small, learners can quickly see how programming works.
- **Rapid experimentation**: with its interactive REPL, Suny encourages trial and error.
- **Scripting and automation**: Suny scripts can be written quickly to automate repetitive tasks.
- **Language research**: Suny itself is a good case study for building interpreters, compilers, and VMs.

1.4 A First Taste of Suny

Here is a simple Suny program:

```
print("Hello, Suny!")

i = 1
while i <= 5 do
    print("Count: %s" % string(i))
    i = i + 1
end
```

This small example demonstrates Suny’s philosophy:

- Clean syntax (`while ... do ... end` is intuitive).
- No unnecessary boilerplate (no `main()` required).
- Immediate feedback in the REPL or script mode.

1.5 Implementation and Portability

Suny is written in **C**, which makes it:

- **Portable**: it can run on Windows, Linux, macOS, or even embedded devices.
- **Efficient**: compiled C code executes quickly with minimal overhead.
- **Compact**: the entire language implementation is small compared to larger interpreters.

The virtual machine (VM) of Suny executes bytecode instructions, similar to Lua or Python, but with a simplified design so it is easy to understand.

1.6 Vision for Suny

Suny will continue to evolve, but its vision will remain the same:

- Stay **small**: the core should remain lightweight and easy to understand.
- Stay **powerful**: expressive enough to build real applications.
- Stay **friendly**: a tool for both learners and curious developers who want to explore language design.

Like Lua, Suny will always value **elegance over complexity**. Like Python, it will always value **readability over clever tricks**.

In this way, Suny aims to be a language that is both a learning tool and a practical scripting solution: **a small language with big possibilities.**

2. Getting Started

Every programming language revolves around two fundamental concepts:

- **Input** – data provided to the program.
- **Output** – results produced by the program.

Suny follows the philosophy of being *small but powerful*: you can start writing programs immediately with very little setup, yet the language remains expressive enough for more complex projects.

2.1. Your First Program

The very first program most people write is a greeting message:

```
print("Hello, Suny!")
```

When executed, it produces:

```
Hello, Suny!
```

Explanation

- `print` is a **built-in function** that sends output to the screen.
- `"Hello, Suny!"` is a **string literal**, which means a fixed sequence of characters enclosed in double quotes.
- In Suny, strings can contain letters, numbers, punctuation, or even Unicode characters.

This simple example already shows Suny's core design principle: **clarity and simplicity**. With a single line of code, you can produce visible output.

2.2. More Printing Examples

The `print` function is versatile. Here are a few variations:

```
print(123)                          # prints a number
print("Suny %s" % "Language")       # sub string with %
print(10 + 20)                      # prints the result of an expression
print(null)                         # prints "null"
```

Output:

```
123
Suny Language
30
null
```

Notes:

- Unlike some languages, `print` in Suny automatically converts values to a string representation when displaying them.

- Multiple arguments are separated by a space.
- The special value `null` represents “nothing” or “no value”.

2.3. Running Suny Programs

There are two main ways to run Suny code:

(a) Interactive Prompt (REPL)

The **REPL** (Read–Eval–Print Loop) lets you type commands one at a time and immediately see results.

To start the prompt, open a terminal and run:

```
suny -p
```

Example session:

```
PS C:\Suny> suny -p
Suny 1.0  Copyright (C) 2025-present, by dinhsonhai132
>> print("Suny is fun!")
Suny is fun!
>> 5 * (2 + 3)
25
>> exit(1)
PS C:\Suny>
```

The REPL is ideal for **learning**, quick experiments, or testing small pieces of code.

(b) Running from a File

For larger programs, it is more convenient to save your code into a file with the extension `.suny`.

Example: create a file called `main.suny` with the contents:

```
print("Welcome to Suny!")
```

Run it with:

```
PS C:\Suny> suny main.suny
Welcome to Suny!
PS C:\Suny>
```

This way, you can build reusable scripts and share them with others.

2.4. Command-Line Options

The `suny` command supports several options. To see them, type:

```
PS C:\Suny> suny -h
```

You will see:

```
Suny 1.0 Copyright (C) 2025-present, by dinhsonhai132
Usage: suny [options] [file]
Options:
  -c [file] Compile the file
  -p Run the prompt
  -h Show this help
```

Explanation:

- `suny file.suny` → Runs the given program file.

- `suny -p` → Starts the interactive prompt (REPL).
- `suny -c file.suny` → Compiles the file (future versions may produce bytecode or executables).
- `suny -h` → Displays the help message.

2.5. Summary

At this point, you know how to:

- Write your first Suny program.
- Display text, numbers, and expressions using `print`.
- Run code interactively in the REPL.
- Execute code stored in `.suny` files.
- Use the command-line options for Suny.

Suny programs begin simple, but the language is designed to scale as you grow. In the following sections, we will cover **basic syntax, variables, data types, and control structures**, which together form the foundation of programming in Suny.

3. Simple Math and Operators

Suny supports a rich set of operators for performing **arithmetic calculations**, **comparisons**, and other common tasks. These operators form the foundation for writing expressions, which are evaluated to produce values.

3.1. Arithmetic Operators

Arithmetic operators allow you to perform basic mathematical calculations.

```
print(2 + 3)       # Addition: 5
print(5 - 2)       # Subtraction: 3
print(4 * 2)       # Multiplication: 8
print(10 / 2)      # Division: 5.0
print(10 % 2)      # modulo: 1
print((1 + 2) * 3) # Parentheses control order: 9
```

Explanation:

- `+` adds numbers.
- `-` subtracts numbers.
- `*` multiplies numbers.
- `%` returns the remainder after division.
- `/` divides numbers and always produces a floating-point result (e.g., `5 / 2` → `2.5`).
- Parentheses `()` can be used to control precedence, just like in mathematics.

3.2. Comparison Operators

Comparison operators compare two values and return a **Boolean result** (`true` or `false`).

```
print(3 < 5)    # true
print(5 > 3)    # true
print(2 == 2)   # true
print(2 <= 3)   # true
print(5 >= 5)   # true
print(10 != 5)  # true
```

Explanation:

- `<` → less than
- `>` → greater than
- `==` → equal to
- `<=` → less than or equal to
- `>=` → greater than or equal to
- `!=` → not equal to

Comparing Different Types

In Suny, comparisons generally make sense when values are of the same type. For example:

```
print("apple" == "apple")   # true
print("apple" == "banana")  # false
```

Comparing numbers and strings directly is not allowed and will raise an error:

```
print(5 == "5")  # Error: cannot compare number and string
```

This design keeps Suny simple and predictable.

3.3. Boolean Results

The results of comparisons are Boolean values: `true` or `false`.

```
a = (5 > 3)
print(a)    # true

b = (10 == 2)
print(b)    # false
```

These Boolean results are often used in **if statements** and **loops** (explained in later chapters).

3.4. Operator Precedence

When multiple operators appear in the same expression, Suny follows a **precedence order**:

1. Parentheses `()`
2. Multiplication, Division, Modulo `* / %`
3. Addition and Subtraction `+ -`
4. Comparisons `< > <= >= == !=`

Example:

```
print(2 + 3 * 4)    # 14, because * has higher precedence
print((2 + 3) * 4)  # 20, because () overrides precedence
```

3.5. Summary

In this section, you learned:

- Suny supports **basic arithmetic operators**: `+ - * / %`.
- The modulo operator `%` returns the remainder.
- **Comparison operators** return Booleans (`true` / `false`) and include `<, >, ==, !=, <=, >=`.
- Operator precedence follows standard mathematical rules.

These operators are the building blocks for expressions. Combined with variables and control structures, they allow you to perform calculations, make decisions, and write meaningful programs.

---

4. Variables

In programming, **variables** are like containers that hold values. Instead of writing the same number or string over and over, you can store it in a variable and use the variable's name to refer to it. This makes your code shorter, clearer, and easier to change later.

In **Suny**, variables are simple and flexible. You don't have to declare their type (like `int` or `float` in C). Instead, Suny figures it out automatically when you assign a value.

4.1 Global Variables

A **global variable** is a variable created outside of any function. It can be accessed from anywhere in the program: inside functions, loops, or just at the top level.

This makes globals powerful, but also dangerous—if too many parts of your code can change the same variable, it's easy to introduce bugs.

Example in Suny:

```
a = 1
b = 2

print(a)  # prints 1
print(b)  # prints 2

b = b + 5
print(b)  # prints 7
```

Key Points:

- A global is “visible” everywhere in the program.
- Changing it inside a function changes it for everyone else too.
- This makes programs easier to write at first, but harder to maintain in the long run.

Comparison:

- **C/C++:** Globals must be declared outside of `main()` or any function, often with a type like `int x = 5;` .
- **Python:** Any variable defined at the top level of a file is global, but inside functions you must use `global x` if you want to modify it.
- **Lua:** All variables are global by default unless marked `local` .

In Suny, like Lua, variables are global unless you put them inside a function, where they become local.

4.2 Local Variables

A **local variable** exists only inside the function where you create it. Once the function finishes running, the variable disappears, and you cannot access it anymore.

This is useful because it prevents different parts of the program from interfering with each other.

Example in Suny:

```
function test() do
    x = 10
    print(x)  # prints 10
end

test()
print(x)  # error: x is not defined
```

Here, `x` is local to `test()` . Trying to use it outside gives an error.

Why Locals Are Better:

- **Safety:** No other part of your program can accidentally change them.
- **Clarity:** When reading the function, you know that the variable belongs only there.
- **Memory:** Locals only live while the function is running, so they free up memory afterward.

Good Practice:

Always prefer **local variables** unless you have a strong reason to use globals. Globals are best for values that truly belong to the whole program, such as configuration settings or constants.

4.3 Assignment

An **assignment** means giving a variable a new value.

In Suny, this is done with the `=` operator:

```
a = 5
b = "hello"
c = true
```

Once assigned, you can use the variable in calculations or other expressions.

Updating Variables

You can change a variable by reassigning it:

```
a = 0
a = a + 1   # now a is 1
a = a * 2   # now a is 2
```

4.4 Example Program

Here's a full program that mixes globals, locals, and assignments:

```
score = 0   # global variable

function add_points(points) do
    local_bonus = 2      # local variable
    score = score + points + local_bonus
    print("Added: %s, Current score: %s" % string(points) % string(score))
end

add_points(5)   # Added 5 points. Current score: 7
add_points(3)   # Added 3 points. Current score: 12

print(score)    # 12
print(local_bonus)  # error: local_bonus is not defined
```

4.5 Summary

- **Variables** hold values like numbers, strings, or booleans.
- **Globals** can be used anywhere but may cause conflicts if overused.
- **Locals** are safer and should be preferred.
- **Assignments** let you change a variable's value, and compound assignments make it shorter.

Think of globals as "public" values and locals as "private" values. If you want your code to be clean, predictable, and bug-free—use locals as much as possible.

5. Data Types

Suny is **dynamically typed**, meaning you don't need to declare the type of a variable explicitly. The type of a variable is determined automatically at runtime based on the value you assign to it.

This makes Suny flexible and expressive, while still providing a consistent set of **core data types**.

The main built-in types in Suny are:

- **Boolean** (`true`, `false`)
- **Numbers** (integers and floats)
- **Strings** (text enclosed in quotes)
- **Lists** (ordered collections of items)
- **Functions** (first-class values, both named and anonymous)

5.1 Boolean

Booleans represent **truth values**: `true` or `false`.

```
is_sunny = true
is_raining = false

print(is_sunny)   # true
print(is_raining) # false
```

Boolean in Conditions

```
weather = "sunny"

if weather == "sunny" then
    print("Go outside!")
else
    print("Stay inside!")
end
```

Booleans are especially important in **control flow** (if/else, loops, etc.). They can also result from **comparison operators**:

```
print(5 > 3)    # true
print(5 < 3)    # false
print(5 == 5)   # true
print(5 != 5)   # false
```

Boolean Operators

Operator	Meaning	Example	Result
`and`	Logical AND	`true and false`	false
`or`	Logical OR	`true or false`	true
`not`	Logical NOT	`not true`	false

5.2 Numbers

Numbers are used for mathematics and calculations. Suny supports **integers** (whole numbers) and **floats** (decimal numbers).

```
a = 10       # integer
b = -5       # integer
c = 3.14     # float
d = -0.5     # float
```

5.3 Strings

Strings represent **text**. They are sequences of characters enclosed in **double quotes** `" "` or **single quotes** `' '`.

```
name = "Dinh Son Hai"
greeting = 'Hello, world!'

print(name)     # Dinh Son Hai
print(greeting) # Hello, world!
```

String Operations

```
first = "Hello"
second = "World"
combined = first + " " + second

print(combined)      # Hello World
print(size(combined)) # 11
```

Strings can be compared:

```
print("abc" == "abc")  # true
print("abc" != "def")  # true
```

Strings in Conditions

```
password = "1234"

if password == "1234" do
    print("Access granted")
else
    print("Access denied")
end
```

Escape Characters

Escape	Meaning	Example	Output
`\n`	Newline	`"Hello\nWorld"`	Hello World
`\t`	Tab	`"Col1\tCol2"`	Col1 Col2
`\\`	Backslash	`"C:\\Path\\File"`	C:\Path\File
`\"`	Double quote	`"He said: \"Hi\""`	He said: "Hi"
`\'`	Single quote	`'It\'s sunny'`	It's sunny

String Formatting (Formatting with `%`)

Strings can be formatted using the `%` operator.

```
name = "Hai"
print("Hello my name is %s" % name)
```

Output:

```
Hello my name is Hai
```

Notes:

- %s is a placeholder for a string value
- string(value) converts numbers to strings

5.4 Lists

Lists are **ordered collections** that can hold multiple items, even of different types.

```
numbers = [1, 2, 3, 4, 5]
names = ["Alice", "Bob", "Charlie"]
mixed = [1, "Two", true, 4.5]
```

Accessing and Modifying Items

```
print(numbers[0])  # 1
numbers[0] = 10
print(numbers[0])  # 10
```

Adding and Removing Items

```
push(numbers, 6)   # append
pop(numbers)       # remove last item
```

Length of a List

```
print(size(numbers))  # 5
```

Looping Over Lists

```
fruits = ["apple", "banana", "cherry"]

# Using index
for i in range(size(fruits)) do
    print(fruits[i])
end

# Using element directly
for fruit in fruits do
    print(fruit)
end
```

5.5 Functions

Functions in Suny are **first-class values**: They can be assigned to variables, passed as arguments, returned, and even created anonymously.

5.5.1 Basic Function

```
function add(a, b) do
    return a + b
end

print(add(1, 2))  # 3
```

5.5.2 Higher-Order Functions

Functions can take other functions as arguments:

```
function apply(func, x, y) do
    return func(x, y)
end

print(apply(add, 5, 7))  # 12
```

5.5.3 Inner Functions

Functions can be nested:

```
function foo() do
    function bar() do
        print("This is bar")
    end
    return bar()
end

foo()  # Output: This is bar
```

5.5.4 Anonymous Functions

```
a = 10

getA = function() do
    return a
end

print(getA())  # 10
```

You can also call them immediately:

```
print(function() do
    return 42
end())   # 42
```

5.5.5 Lambda Functions

A **lambda** is a short form of anonymous function:

```
let f(x) = x + 1
print(f(2))  # 3
```

5.5.6 Closure

A **closure** is a function that can *remember and access* variables from the scope in which it was created, even after that outer scope has finished executing.

In other words:

> **Closure = Function + Its captured environment (the variables it keeps).**

This allows functions to “carry” state and behave like objects with private data.

Detailed Example

```
function foo() do
    count = 0
    return function() do
        count = count + 1
        return count
    end
end

a = foo()
for i in range(10) do
    print(a())
end
```

Here’s what happens:

1. `foo()` is called. Inside it, the variable `count` is created.
2. `foo()` returns an inner function that **uses `count`**.
3. Even though `foo()` has finished, `count` does **not** disappear.
4. The returned function `a()` keeps a reference to `count`. This saved environment is the **closure**.
5. Each time `a()` is called, it updates the same `count` value.

So the function “remembers” its state:

```
1
2
3
...
10
```

5.6 HashMap

HashMaps (also called dictionaries or objects) are **key-value collections** that allow you to store and retrieve data using keys instead of numeric indices.

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "Hanoi"
}

print(person["name"])  # Alice
print(person["age"])   # 25
```

Creating HashMaps

HashMaps are created using curly braces `{}` with key-value pairs:

```
# Empty HashMap
empty = {}

# HashMap with initial values
user = {
    "username": "hai123",
    "email": "hai@example.com",
    "active": true
}
```

Accessing Values

Use square brackets `[]` with the key to access values:

```
config = {
    "theme": "dark",
    "language": "vi"
}

print(config["theme"])     # dark
print(config["language"])  # vi
```

Modifying and Adding Values

You can change existing values or add new key-value pairs:

```
settings = {
    "volume": 50,
    "muted": false
}

# Modify existing value
settings["volume"] = 75

# Add new key-value pair
settings["brightness"] = 80

print(settings)
# { "volume": 75, "muted": false, "brightness": 80 }
```

Nested HashMaps

HashMaps can contain other HashMaps, creating nested structures:

```
game = {
    "player": {
        "hp": 100,
        "items": ["sword", "shield"]
    },
    "fps": 60
}

# Access nested values
print(game["player"]["hp"])        # 100
print(game["player"]["items"][0])  # sword

# Modify nested values
game["player"]["hp"] = 200
game["fps"] = 60
game["name"] = "RPG"

print(game)
# {
#     "player": { "hp": 200, "items": ["sword", "shield"] },
#     "fps": 60,
#     "name": "RPG"
# }
```

Common HashMap Operations

```
data = {
    "x": 10,
    "y": 20
}

# Get number of key-value pairs
print(size(data))  # 2

# Check if key exists (using conditional)
if data["z"] == null then
    print("Key 'z' does not exist")
end
```

Use Cases

HashMaps are perfect for:

- Configuration settings
- Game state management
- User profiles and data
- Structured data representation
- Mapping relationships between values

5.7 Summary

- Booleans → true/false
- Numbers → integers & floats with `+ - * / // %`
- Strings → text with escape sequences
- Lists → ordered collections with indexing and iteration
- Functions → first-class values, support closures, anonymous functions, and lambdas

---

6. Class and OOP

Classes allow you to define **custom data types** with **properties** and **methods**. They are the foundation of **Object-Oriented Programming (OOP)**.

6.1 Defining a Class

A class is defined using the `class` keyword.

```
class Vector do
    x = null
    y = null

    function __init__(self, x, y) do
        self.x = x
        self.y = y
        return self
    end
end
```

6.2 Operator Overloading

Suny allows operator overloading using special methods:

Method	Operator
`__add__`	`+`
`__sub__`	`-`
`__mul__`	`*`
`__div__`	`/`
`__tostring__`	string conversion

6.3 Full Vector Example

```
class Vector do
    x = null
    y = null

    function __init__(self, x, y) do
        self.x = x
        self.y = y
        return self
    end

    function __add__(a, b) do
        return Vector(a.x + b.x, a.y + b.y)
    end

    function __sub__(a, b) do
        return Vector(a.x - b.x, a.y - b.y)
    end

    function __mul__(a, b) do
        return Vector(a.x * b.x, a.y * b.y)
    end

    function __div__(a, b) do
        return Vector(a.x / b.x, a.y / b.y)
    end

    function __tostring__(self) do
        return "Vector(%s, %s)" % string(self.x) % string(self.y)
    end
end
```

6.4 Using the Class

```
a = Vector(10, 20)
b = Vector(2, 4)

print(a + b)  # Vector(12, 24)
print(a - b)  # Vector(8, 16)
print(a * b)  # Vector(20, 80)
print(a / b)  # Vector(5, 5)
```

6.5 Example Loop

```
a = Vector(0, 0)

while true do
    a = a + Vector(1, 1)
    print(a)
end
```

6.6 Notes

- `self` refers to the current object.
- Operator methods always receive **two operands**.
- `__tostring__` controls how an object is printed.
- Class names usually start with a capital letter.

6.7 Inheritance

Suny supports **single** and **multiple inheritance** using the `extends` keyword.

- A subclass automatically inherits **fields** and **methods** from its parent classes.
- If a field or method is redefined, the **child overrides** the parent version.
- With multiple inheritance, resolution follows **left-to-right order**.

6.7.1 Base Classes

```
class Animal do
    leg = 0
end

class Fly do
    wings = 0
end
```

6.7.2 Single Inheritance

```
class Dino extends Animal do
    leg = 2
end
```

`Dino` inherits from `Animal` and overrides `leg`.

```
d = Dino()
print(d.leg)  # 2
```

6.7.3 Multiple Inheritance

```
class Chicken extends Animal, Fly do
    leg = 2
    wings = 2
end
```

`Chicken` inherits from both `Animal` and `Fly`.

```
c = Chicken()
print(c.leg)    # 2
print(c.wings) # 2
```

6.8 Method Inheritance Example

```
class Animal do
    function speak(self) do
        return "..."
    end
end

class Dog extends Animal do
    function speak(self) do
        return "Woof"
    end
end

print(Dog().speak())  # Woof
```

6.9 Summary

- `extends` enables inheritance
- Child classes can override fields and methods
- Multiple inheritance is supported
- Method lookup is **child → parent (left to right)**

Inheritance allows Suny programs to model real-world relationships cleanly and reuse code effectively.

---

7. Control Structures

Control structures determine the **flow of execution** in a program. They allow you to make decisions, repeat actions, and handle complex logic.

Suny provides these main control structures:

- **Conditional Statements** (`if`, `else`)
- **Loops** (`while`, `for`)
- **Special Controls** (`break`, `continue`)

7.1 Conditional Statements

Conditional statements let your program choose between different paths.

7.1.1 Basic `if`

```
score = 75

if score >= 50 then
    print("You passed!")
end
```

-> If the condition is `true`, the code inside the block runs. -> If it is `false`, the block is skipped.

7.1.2 `if ... else`

```
score = 40

if score >= 50 then
    print("You passed!")
else
    print("Try again")
end
```

If the first condition fails, the `else` block executes.

7.1.3 `if ... else`

```
score = 85

if score >= 90 then
    print("Excellent")
else
    print("Failed")
end
```

 Conditions are checked from top to bottom.

 Only the **first matching block** is executed.

7.1.4 Nested Conditions

Conditions can be placed inside each other:

```
age = 20
has_id = true

if age >= 18 then
    if has_id then
        print("Access granted")
    else
        print("ID required")
    end
else
    print("Too young")
end
```

7.1.5 `if ... elif ... else`

`elif` (else-if) is used when multiple conditions must be checked in sequence, but **only one branch should be executed**.

```
score = 75

if score >= 90 then
    print("Excellent")
elif score >= 75 then
    print("Good")
elif score >= 50 then
    print("Pass")
else
    print("Failed")
end
```

 Conditions are evaluated **from top to bottom** **Only the first matching block** is executed `else` is **optional**

7.1.6 Multiple `elif` branches

Any number of `elif` branches may be used:

```
x = 0

if x > 0 then
    print("Positive")
elif x == 0 then
    print("Zero")
elif x < 0 then
    print("Negative")
end
```

7.1.7 `elif` vs Nested `if`

Nested `if` (less readable):

```
if score >= 90 then
    print("Excellent")
else
    if score >= 75 then
        print("Good")
    else
        print("Failed")
    end
end
```

Using `elif` (recommended):

```
if score >= 90 then
    print("Excellent")
elif score >= 75 then
    print("Good")
else
    print("Failed")
end
```

Recommendation

- Use `elif` when conditions are **mutually exclusive**
- Use nested `if` when logic is **hierarchical or dependent**

7.1.5 Comparison Operators Recap

Operator	Meaning	Example	Result
`==`	Equal	`5 == 5`	true
`!=`	Not equal	`5 != 3`	true
`>`	Greater than	`5 > 3`	true
`<`	Less than	`3 < 5`	true
`>=`	Greater than or equal to	`5 >= 5`	true
`<=`	Less than or equal to	`4 <= 5`	true

7.1.6 Boolean Logic in Conditions

```
temperature = 30
sunny = true

if temperature > 25 and sunny do
    print("Perfect beach day")
end

if temperature < 10 or not sunny do
    print("Maybe stay home")
end
```

7.2 Loops

Loops let you **repeat code** multiple times.

7.2.1 `while` Loop

Repeats while the condition is `true`.

```
count = 0

while count < 5 do
    print(count)
    count = count + 1
end
```

Output:

```
0
1
2
3
4
```

7.2.2 Infinite `while`

```
while true do
    print("Running forever...")
end
```

-> You usually combine this with `break` to stop.

7.2.3 `for ... in range()`

Suny provides `range(n)` to generate numbers from `0` to `n-1`.

```
for i in range(5) do
    print(i)
end
```

Output:

```
0
1
2
3
4
```

You can also use `range(start, end)`:

```
for i in range(3, 7) do
    print(i)
end
# 3, 4, 5, 6
```

7.2.4 `for ... in list`

Iterating over items directly:

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits do
    print(fruit)
end
```

Output:

```
apple
banana
cherry
```

7.2.5 Loop Structures: `loop`, `times`, and `do`

In this section, we explore how to execute blocks of code repeatedly using basic looping constructs. These tools are essential for handling tasks that require iteration, such as incrementing counters or processing data sets.

---

1. The Basic `loop`

The `loop` keyword creates an **infinite loop**. It will continue to execute the code inside the `do...end` block until it is manually interrupted (e.g., by a `break` statement or a system exit).

Syntax Example:

```
i = 0

loop do
    print(i)
    i = i + 1

    if i >= 5 then
        break
    end
end
```

---

2. The `times` Iterator

When you know exactly how many times a block of code should run, use the `times` method. This is a cleaner, more readable way to handle fixed iterations compared to manual counters.

Syntax Example:

```
i = 0

loop 5 times do
    print(i)
    i = i + 1
end
```

- **How it works:** The code inside the `do...end` block will execute exactly 5 times.
- **Alternative Syntax:** You can also use curly braces for single-line loops: `5.times { |n| print(n) }`.

---

3. Comparison Table

Feature	`loop`	`times`
Duration	Infinite (until broken)	Fixed number of iterations
Use Case	Waiting for an external signal or user input	Repeating a task a specific number of times
Control	Requires an internal `break`	Ends automatically

Key Takeaways

- **The `do...end` Block:** Both constructs use `do` and `end` to encapsulate the logic that should be repeated.
- **Counter Management:** In a `loop`, you are responsible for incrementing your variables and defining the exit condition. In `times`, the iteration limit is handled

for you.

> **Warning:** Always ensure that an infinite `loop` has a way to terminate, otherwise your program will hang or crash due to memory exhaustion.

7.2.6 Nested Loops

```
for i in range(3) do
    for j in range(2) do
        print("i = %, j = %" % string(i) % string(j))
    end
end
```

7.3 Loop Control: `break` and `continue`

7.3.1 `break`

Stops the loop immediately.

```
for i in range(10) do
    if i == 5 do
        break
    end
    print(i)
end
```

Output:

```
0
1
2
3
4
```

7.3.2 `continue`

Skips to the next iteration.

```
for i in range(5) do
    if i == 2 do
        continue
    end
    print(i)
end
```

Output: `0 1 2 3 4`

7.4 Combining Control Structures

Complex programs often use **if statements inside loops**:

```
for i in range(1, 11) then
    if i % 2 == 0 then
        print("%s is even" % string(i))
    else
        print("%s is odd" % string(i))
    end
end
```

7.5 Summary

- `if`, `elif`, `else` → decision making
- `while` → repeat while condition is true
- `for` → iterate over ranges or collections
- `break` → exit loop early
- `continue` → skip current iteration

Control structures are the **backbone of logic** in Suny. They allow you to model real-world decisions, repeat tasks, and build dynamic programs.

---

8. Include

The **`include`** statement in Suny allows you to organize your program across multiple files or folders. Instead of writing everything in a single file, you can split your code into smaller parts (modules, configs, helpers) and bring them together when needed.

When Suny sees an `include`, it **inserts the code from that file or folder directly into the current file** before running the program. This makes it behave almost the same as if you had copy-pasted the contents manually, but in a more organized way.

8.1 Including a File

If the target is a **file**, Suny copies all its contents.

```
# config.suny
pi = 3.14
```

```
# main.suny
include "config.suny"

print(pi)   # 3.14
```

Here, the variable `pi` becomes part of `main.suny`'s scope, as if it was defined inside it.

8.2 Including a Folder

If the target is a **folder**, Suny automatically looks for a file named `main.suny` inside that folder.

```
math/
 ├── main.suny
 └── extra.suny
```

```
# math/main.suny
square(x) = x * x
```

```
# main.suny
include "math"

print(square(5))   # 25
```

Suny only loads `math/main.suny` by default. If you need extra files (`extra.suny` in this example), you must include them explicitly:

```
include "math/extra.suny"
```

8.3 Error Cases

- **Missing file or folder**

```
include "not_found"
```

```
Error: include target 'not_found' not found
```

- **Folder without main.suny**

```
mylib/
└── helper.suny
```

```
include "mylib"
```

```
Error: no main.suny in folder 'mylib'
```

- **Name conflicts**

  ```
  # a.suny
  x = 10

  # b.suny
  x = 20
  ```

  ```
  include "a"
  include "b"

  print(x)   # which one? result depends on last include
  ```

  Best practice: avoid re-using the same global variable names across includes.

8.5 Summary

- Use `include` for **constants, small configs, or utility functions**.
- Keep each folder/module self-contained with its own `main.suny`.
- Avoid circular includes (`A` includes `B`, and `B` includes `A`).
- Use unique variable/function names to prevent conflicts.