Programming in Suny

1. INTRODUCTION

- Suny is a lightweight scripting language designed to be simple, clear, and easy to learn, even for people who have never programmed before.
- It was written in C, making it fast and efficient, while keeping syntax clean and beginner-friendly.

Key Features:

- Simplicity – easy-to-remember syntax
- Clarity – clean and readable code
- Flexibility – simple yet strong enough to solve complex problems

Uses:

- Learn programming concepts easily
- Write small scripts and automate tasks
- Experiment with building applications in a fun, lightweight way

---

2. GETTING STARTED

Every programming language has input and output functions:

- Input function: allows users to enter data. The program stores this input, often as a string.
- Output function: displays information on the screen.

Example:

```
print("hello")
```

Output:

```
hello
```

Explanation:

- print displays text or values on the screen.
- Text in double quotes " is a string, storing letters, numbers, or symbols.

Running Suny Programs

- Save your program as a .suny file.
- Run it using the interpreter:

```
prompt> suny your_file.suny
```

Example:

```
prompt> suny main.suny
hello
prompt>
```

Using REPL

Suny also provides a REPL (Read-Eval-Print-Loop) to run code directly in the terminal:

```
prompt> suny
Suny 1.0 Copyright (C) 2025-present, by dinhsonhai132
>> print("hello")
hello
>>
```

REPL Breakdown:

- Read: reads the code
- Eval: evaluates (runs) the code
- Print: displays results
- Loop: repeats for the next input

Benefits:

- Instant feedback
- Practice and learning tool
- Test small code snippets without a file

---

3. SIMPLE MATH

Suny provides mathematical symbols for calculations:

```
>> print(2 + 2)
4
>> print((1 + 1) * 2)
4
>> print(4 / 2 * 2)
4
```

Basic Operators

- `+` → addition
- `-` → subtraction
- `*` → multiplication
- `/` → division
- `(`, `)` → parentheses, control calculation order

Comparison Operators

- `<` → less than
- `>` → greater than
- `==` → equal to
- `<=` → less than or equal to
- `>=` → greater than or equal to

Example:

```
>> print(5 > 3)
true
>> print(2 == 2)
true
>> print(1 <= 0)
false
```

---

4. GLOBAL VARIABLE

Global variables can be accessed and modified anywhere in the program.

Define global variables:

```
a = 1
b = 2
print(a) # 1
print(b) # 2
```

Notes:

- Global Scope: variables outside functions
- Modifying: in Suny, globals can be used directly inside functions
- Use: share information across functions, but avoid overuse

---

5. DATA TYPE

- Suny is a dynamically typed language. There are no type definitions in the language; each value carries its own type.
- There are 5 basic types in Suny, boolen, list, float, function, string

5.1 BOOLEN

A Boolean is a type of data that can only have two values:

- true → represents truth
- false → represents falsehood

Booleans are often used in conditions and comparisons.

Example:

```
is_sunny = true
is_raining = false

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

Using Boolean in conditions:

```
weather = "sunny"

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

Explanation:

- `==` checks if two values are equal.
- The condition inside if must evaluate to a Boolean (`true` or `false`).
- If the condition is `true`, the code inside the if block runs; otherwise, the else block runs.
- Booleans are fundamental for controlling program flow and making decisions in your code.

5.2 NUMBER

- In Suny, numbers are a basic data type used to store numeric values. They can be either integers (whole numbers) or floating-point numbers (numbers with decimals).

Example:

```
# Integer numbers
a = 10
b = -5

# Floating-point numbers
c = 3.14
d = -0.5

print(a)  # 10
print(b)  # -5
print(c)  # 3.14
print(d)  # -0.5
```

Arithmetic with Numbers:

Suny supports basic arithmetic operations:

- `+` → addition
- `-` → subtraction
- `*` → multiplication

- `/` → division

```
x = 10
y = 3

print(x + y)  # 13
print(x - y)  # 7
print(x * y)  # 30
print(x / y)  # 3.3333...
```

5.3 STRINGS

In Suny, a string is a sequence of characters used to represent text. Strings are enclosed in double quotes: `"`.

Example:

```
name = "Đinh Son Hai"
greeting = "Hello, world!"

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

String Operations:

1. Concatenation (joining strings)

```
first = "Hello"
second = "World"
combined = first + " " + second
print(combined)  # Hello World
```

2. String Length

```
text = "Suny"
print(size(text))  # 4
```

3. Strings in Conditions

```
password = "1234"

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

Explanation:

- Strings store text data.
- You can join them, measure their length, access individual characters, and compare them in conditions.

Escape Characters in Strings

In Suny, you can use escape characters to represent special characters inside strings. Each escape sequence starts with a backslash `\`.

Escape	Meaning	Example	Output
`\n`	Newline (move to next line)	`"Hello\nWorld"`	`Hello` `World`
`\t`	Tab (adds horizontal space)	`"Col1\tCol2"`	`Col1 Col2`
`\r`	Carriage return (moves cursor to line start)	`"12345\rAB"`	`AB345`
`\\`	Backslash	`"C:\\Path\\File"`	`C:\Path\File`
`\"`	Double quote inside string	`"He said: \"Hi\""`	`He said: "Hi"`
`\'`	Single quote inside string	`'It\'s sunny'`	`It's sunny`

Example Usage:

```
print("Hello\nWorld")     # prints on two lines
print("Column1\tColumn2") # adds a tab space
print("Backslash: \\")     # prints a single backslash
print("Quote: \"Hi\"")  # prints double quotes inside string
```

5.4 LISTS

In Suny, a list is a collection of items stored in a single variable. Lists can store numbers, strings, Booleans, or even other lists.

Creating a List:

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

Accessing Items:

- Lists are zero-indexed (the first item has index 0).

```
print(numbers[0])  # 1
print(names[2])    # Charlie
```

Modifying Items:

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

Adding Items:

```
push(numbers, 4)
print(numbers)  # [1, 2, 3, 4]
```

Pop Items:

```
numbers = [1, 2, 3, 4]
pop(numbers)
print(numbers)  # [1, 2, 3]
```

List Length:

```
print(size(numbers))  # 3
Lists in Loops:
fruits = ["apple", "banana", "cherry"]
for i in range(size(fruits)) do
    print(fruits[i])
end
```

Or you can:

```
fruits = ["apple", "banana", "cherry"]
for i in fruits do
    print(i)
end
```

5.4 FUNCTIONS

Expressions denote values. Expressions in Suny include:

- Numeric constants
- String literals
- Variables
- Unary and binary operations
- Function calls

Expressions can also include:

- Unconventional function definitions
- Table constructors

Examples

5.4.1. Basic Function

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

print(foo(1, 2))  # Output: 3
```

5.4.2. Function Returning Another Function

```
func bar() do
    return bar
end

c = bar()
print(c())  # Output: function itself
```

- `bar` returns itself.
- `c` now holds the function `bar`.
- Calling `c()` effectively calls `bar()` again.

- This is an example of a higher-order function, where a function can return another function (or itself).

5.4.3. Functions as Values

Functions in SunyLang are first-class values:

- Can be assigned to variables
- Can be passed as arguments to other functions
- Can be returned from functions

```
function add(x, y) do
    return x + y
end

function apply(func, a, b) do
    return func(a, b)
end

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

- `apply` takes a function `func` and two values `a`, `b`.
- Calling `apply(add, 5, 7)` executes `add(5, 7)` and prints `12`.

4. LOGICAL EXPRESSIONS

In Suny, logical expressions allow you to combine or invert Boolean values using and, or, and not operators.

Operators:

- `and` → returns `true` if both values are `true`
- `or` → returns `true` if at least one value is `true`
- `not` → returns the opposite Boolean value

Examples:

```
x = true
y = false

print(x and y)  # false
print(x or y)   # true
print(not x)    # false
print(not y)    # true
```

Using Logical Expressions in Conditions:

```
is_sunny = True
has_umbrella = False

if is_sunny or has_umbrella do
    print("Go outside")
```

```
else
    print("Stay inside")
end

if not is_sunny do
    print("It is cloudy")
end
```

Explanation:

- `and` requires both conditions to be True to return True.
- `or` requires at least one condition to be True to return True.
- `not` inverts the Boolean value.
- Logical expressions are very useful for making decisions in your programs.

5. CONTROL STRUCTURES

Control Structures

- Suny provides a small and conventional set of control structures:
- `if`, `else` for conditional statements
- `while` for loops
- `for` for iteration

All control structures have an explicit terminator: `end` terminates the `if`, `for`, and `while` structures.

5.1 CONDITIONAL

Conditional statements let your program make decisions based on Boolean conditions.

Syntax:

```
if condition do
    # code to run if condition is True
else
    # code to run if condition is False
end
```

Example:

```
score = 75

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

Explanation:

- `if` checks the condition.

- `else` runs if the if condition is False.
- `end` marks the end of the conditional block.

5.2 WHILE

The while loop allows your program to repeat a block of code as long as a condition is True.

Syntax:

```
while condition do
    # code to repeat while condition is True
end
```

Example:

```
count = 1

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

Explanation:

- The loop will continue as long as the condition is True.
- Update variables inside the loop to avoid an infinite loop.
- end marks the end of the while loop.

5.3 For

The for loop is used for iterating over a range of values or items in a collection.

For with a range:

```
for i in range(0, 5)
    print(i)
end
```

- Iterates from 1 to 5, printing each value.
- Using range(a, b) function to generate a list from a to b if (b > a) or b to a if (b < a)

For-in loop (iterating over a collection):

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

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

- Iterates over each item in the fruits list.
- fruit represents the current item in each iteration.
- `end` marks the end of the loop.

Explanation:

- Standard for is for numeric ranges.
- `for`, `in` is for iterating over lists or collections.
- Both loops require end to close the block.

The End

Thank you for reading!