Programming in Suny (version 1.0)

---

1. Introduction

Suny is a lightweight scripting language designed to be **simple, readable, and beginner-friendly**. It is ideal for people who are just starting programming, but also powerful enough to solve real problems.

Suny is written in **C**, which makes it **fast, efficient, and portable** across different platforms. Its syntax is clean, minimizing boilerplate code and focusing on **clarity and simplicity**.

Key Features

- **Simplicity:** Minimal and easy-to-remember syntax.
- **Clarity:** Readable code that is easy to understand.
- **Flexibility:** Supports a wide range of programming tasks without complexity.

Typical Uses

- Learn programming concepts quickly.
- Write small scripts or automation tasks.
- Experiment with application development in a lightweight way.

Suny encourages **learning by doing**, allowing users to experiment interactively and receive instant feedback through its REPL.

---

2. Getting Started

All programs involve **input** (getting data from the user) and **output** (displaying results):

- **Input:** Receives information from the user (usually as a string).
- **Output:** Displays text, numbers, or other information on the screen.

Example: Printing a Message

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

Output:

```
Hello, Suny!
```

Explanation:

- `print()` is used to display text or values.
- Text inside double quotes `" "` is a **string**, representing letters, numbers, or symbols.

Running Suny Programs

1. Save your code in a file with the extension `.suny`.
2. Run it using the Suny interpreter:

```
prompt> suny my_program.suny
```

Example:

```
prompt> suny main.suny
Hello, Suny!
prompt>
```

Using the REPL

Suny includes a **REPL** (Read-Eval-Print-Loop) for interactive programming:

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

How the REPL Works:

1. **Read:** Reads your input code.
2. **Eval:** Executes the code.
3. **Print:** Displays the result.
4. **Loop:** Repeats the process until you exit.

Advantages of REPL:

- Instant feedback while learning.
- Test small code snippets without creating files.
- Experiment safely without affecting saved code.

---

3. Simple Math and Operators

Suny supports **basic arithmetic** and **comparison operations**, allowing you to perform calculations and make decisions.

Arithmetic Operations

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

Comparison Operators

Comparison operators are used to compare values. They return **Boolean values** (`true` or `false`):

```
# Comparison examples
print(3 < 5)    # true
print(5 > 3)    # true
print(2 == 2)   # true
print(2 <= 3)   # true
print(5 >= 5)   # true
```

Explanation:

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

---

4. Variables

Global Variables

Variables store values your program can use.
A global variable is defined outside of functions and can be used anywhere in the program.

Example in Suny:

```
a = 1
b = 2

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

Notes:

- Global variables can be read and modified from any part of the program.
- Too many globals can make programs hard to manage. Prefer local variables inside functions to avoid name conflicts and unexpected changes.

Local Variables

A local variable is defined inside a function and only exists while the function is running.
It cannot be accessed outside the function.

Example in Suny:

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

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

Notes:

- Locals are safer than globals because they do not affect the rest of the program.
- Use locals whenever possible.

Assignment

Assignments update the value stored in a variable.

Example in Suny:

```
a = 0   # set variable
a += 1  # increase by 1
a -= 1  # decrease by 1
a *= 2  # multiply by 2
a /= 2  # divide by 2
```

These compound assignments are shorthand for longer forms:

- `a += 1` is the same as `a = a + 1`
- `a -= 1` is the same as `a = a - 1`
- `a *= 2` is the same as `a = a * 2`
- `a /= 2` is the same as `a = a / 2`

---

5. Data Types

Suny is **dynamically typed**, which means variables do not need an explicit type declaration. Suny automatically determines the type based on the value.

5.1 Boolean

Booleans represent truth values:

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

Use in conditions:

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

5.2 Numbers

Suny supports **integers** and **floating-point numbers**:

```
# Integers
a = 10
b = -5

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

Arithmetic Examples:

```
x = 10
y = 3
print(x + y)  # 13
print(x - y)  # 7
print(x * y)  # 30
print(x / y)  # 3.3333
```

5.3 Strings

Strings are sequences of characters enclosed in double quotes `" "`:

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

Operations with Strings:

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

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

Using 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

5.4 Lists

Lists store multiple items:

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

Accessing items:

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

Modifying items:

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

Adding and removing items:

```
push(numbers, 6)
pop(numbers)
```

Length of a list:

```
print(size(numbers))
```

Looping over lists:

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

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

# Using item 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 to other functions, returned from functions, and even created anonymously.

Basic Function

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

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

Explanation:

- `function name(parameters) do ... end` defines a named function.
- `return` specifies the value that the function gives back.
- You can call the function using its name, passing required arguments.

Higher-Order Functions

Functions can accept other functions as arguments, or return functions:

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

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

- `apply` takes a function `func` and two numbers, and calls `func(x, y)` .
- This demonstrates **higher-order functions**, useful for functional programming patterns.

Inner Functions

Functions can be **defined inside another function**. The inner function is local to the outer function:

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

    return bar()
end

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

- `bar` exists only within `foo` .
- This allows **encapsulation** and avoiding global namespace pollution.

Anonymous Functions

Anonymous functions are **functions without a name**. They can be used inline or stored in variables:

```
a = 10

print(function() do
    return a
end)()  # Output: 10
```

Explanation:

- `function() do ... end` defines an anonymous function.
- The function can be called immediately by adding `()` at the end.
- You can also assign it to a variable:

```
getA = function() do
    return a
end

print(getA())  # 10
```

Use Cases:

- Passing functions to other functions (callbacks).
- Returning functions from functions.
- Writing concise code without naming every function.

Lambda

A **lambda** is a short, anonymous function you can define quickly without using the full `function ... do ... end` syntax.

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

Notes:

- `let` defines a lambda function.
- Lambdas are useful for small, one-line functions.
- They can be assigned to variables, passed as arguments, or returned from functions.

Example of what function can does

```
function foo() do
    count = 0

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

```
a = foo()
n = 0
while n < 10 do
    n = a()
    print(n)
end
```

Output:

```
prompt> suny main.mer
1
2
3
4
5
6
7
8
9
10
prompt>
```

---

6. Logical Expressions

Logical operators:

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

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

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
```

---

7. Control Structures

Conditional Statements

```
score = 75
if score >= 50 do
    print("You passed!")
else
    print("Try again")
end
```

While Loops

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

For Loops

Range-based:

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

Collection-based:

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

---

8. Include

Suny lets you split your program into multiple files and reuse code with **include**.
When you use `include` , the contents of the other file are inserted **directly into the current file**.

Example

```
# config.suny
pi = 3.14


# main.suny
include "config.suny"

print(pi)  # 3.14
```

Notes:

- `include` is like copy-pasting the file's code into the current file.
- All variables and functions from the included file become part of the current scope.
- Useful for small shared files such as constants or configuration.
- Be careful with naming conflicts (two includes defining the same variable).

---

9. Import

Unlike `include` , the `import` statement works specifically with `.dll` files. `import` locates and loads the function:

```
SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler comp
```

marked with `SUNY_API` in the `.dll` file and executes it. If the function is not found, an error will occur.

Working with Import

You can create custom datatypes using `import` or develop your own **functions** by compiling them into `.dll` files. Simply use `import your_lib` in your `suny` file, and you'll be able to access these functions directly without needing `call()` to load `.dll` functions. Your `.dll` file must contain the `Smain` function.

Example: Create a `mathlib.c` file

```
#include <Suny.h>

// Create your own built-in function in C

struct Sobj* Scos(struct Sframe* frame) {
    struct Sobj* angle = Sframe_pop(frame); // Take the first argument
    float result = cosf(angle->value->value);
    return Sobj_set_int(result);
}

SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler comp
    SunyInitialize_c_api_func(frame, compiler, 33, "cos", 1, Scos);
    //                                          ^       ^      ^        ^
    //                                     address  function  arg     poir
    //                                              name      count    func
    return frame;
}
```

Compile it into `mathlib.dll`, then create and run a Suny file that imports it:

```
import "mathlib"

print(cos(90))
```

---

10. Userdata

In `Suny`, you can create custom datatypes beyond the built-in types like lists and functions using `userdata`.

Create your `.c` file:

```c
#include <Suny.h>

struct Svector { // Your custom datatype
    int x;
    int y;
};

struct Sobj* Svector_new(int x, int y) {
    struct Svector* vector = (struct Svector*)malloc(sizeof(struct Svecto

    vector->x = x;
    vector->y = y;

    struct Sobj* userdata = Sobj_make_userdata(vector);
    return userdata;
}

struct Sobj* Sobj_make_vector(struct Sframe* frame) {
    struct Sobj* y = Sframe_pop(frame);
    struct Sobj* x = Sframe_pop(frame);

    int xv = Sobj_get_value(x);
    int yv = Sobj_get_value(y);

    struct Sobj* vo = Svector_new(xv, yv);
    return vo;
}

SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler* com
    SunyInitialize_c_api_func(frame, compiler, 33, "vector", 2, Sobj_make
    return frame;
}
```

Compile it into `vector.dll` and run it:

```
import "vector"

a = vector(1, 2)
b = vector(2, 3)
print(a + b)
```

Command prompt output:

```
0
prompt>
```

The output shows that the vector doesn't support the `+` operation by default. To enable this functionality, we need to implement meta methods:

```
#include <Suny.h>

#define VEC_X(v) (((struct Svector*)(v->f_type->f_userdata->data))->x)
#define VEC_Y(v) (((struct Svector*)(v->f_type->f_userdata->data))->y)

struct Svector {
    int x;
    int y;
};

struct Sobj* Svector_new(int x, int y) {
    struct Svector* vector = (struct Svector*)malloc(sizeof(struct Svecto
    vector->x = x;
    vector->y = y;

    struct Sobj* userdata = Sobj_make_userdata(vector);
    return userdata;
}

struct Sobj* Sobj_vec_add(struct Sobj* v1, struct Sobj* v2) {
    int tx = VEC_X(v1) + VEC_X(v2);
    int ty = VEC_Y(v1) + VEC_Y(v2);

    struct Sobj* userdata = Svector_new(tx, ty);
    userdata->meta = v1->meta; // Assign the meta from v1

    return userdata;
}

struct Sobj* Sobj_vec_print(struct Sobj* v) {
    int x = VEC_X(v);
    int y = VEC_Y(v);

    printf("vector(%d, %d)", x, y);
    return v;
}

struct Sobj* Sobj_make_vector(struct Sframe* frame) {
```

```
    struct Sobj* y = Sframe_pop(frame);
    struct Sobj* x = Sframe_pop(frame);

    int xv = Sobj_get_value(x);
    int yv = Sobj_get_value(y);

    struct Sobj* vo = Svector_new(xv, yv);

    // Initialize meta methods
    vo->meta = malloc(sizeof(struct Smeta));
    vo->meta->mm_add = Sobj_vec_add;
    vo->meta->mm_tostring = Sobj_vec_print;

    return vo;
}

SUNY_API struct Sframe* Smain(struct Sframe* frame, struct Scompiler* com
    SunyInitialize_c_api_func(frame, compiler, 33, "vector", 2, Sobj_make
    return frame;
}
```

Now let's run it:

```
import "vector"

a = vector(1, 2)
b = vector(2, 3)
print(a + b)
```

Command prompt output:

```
vector(3, 5)
prompt>
```

Using meta methods, we can now support operations like `+`, `-`, `*`, `/`, `to_string`, and more for custom datatypes.

---

8. Summary

Suny provides a **clear and easy-to-learn syntax**, making it suitable for beginners and lightweight scripting tasks. With support for:

- Variables and data types

- Functions and higher-order functions
- Lists and loops
- Logical expressions and conditional statements

You can quickly and efficiently build simple to moderately complex programs.

9. Suny Standard library

`Suny` has one standard lib called `stdlib`, `stdlib` is the only library in `Suny`.
It supports multiple datatypes such as `vector`, `bigint`, `complex`, ...
and `stdlib` also provides a lot of math constants such as `pi`, `euler`, ...
and math functions like `sin`, `cos`, `tan`, ...

To use `stdlib`, import it with:

```
import "stdlib"

a = vector([1, 2, 3])
b = vector([4, 5, 6])

print(a + b)    # output: vector(5, 7, 9)
```

9.1 Vector

`vector` is a datatype in library `stdlib` of Suny.
Vector works like a mathematical vector and supports element-wise operations.

Creation

```
a = vector([1, 2, 3])
b = vector([4, 5, 6])
```

Operations

```
print(a + b)    # vector(5, 7, 9)
print(a - b)    # vector(-3, -3, -3)
print(a * b)    # vector(4, 10, 18)    (element-wise multiplication)
print(a / b)    # vector(0.25, 0.4, 0.5)
```

Scalar operations

```
print(a * 2)   # vector(2, 4, 6)
print(a / 2)   # vector(0.5, 1, 1.5)
```

9.2 Bigint

`bigint` is a datatype for working with very large integers that cannot fit in normal 64-bit integers.

Creation

```
a = bigint("12345678901234567890")
b = bigint("98765432109876543210")
```

Operations

```
print(a + b)
print(a * b)
print(a - b)
print(a / b)
print(a + 1) # also work for float
```

Bigint supports the same arithmetic operators as normal integers, but with arbitrary precision.

9.3 Complex

`complex` is a datatype for complex numbers in the form `a + bi`.

Creation

```
z1 = complex(1, 2)   # 1 + 2i
z2 = complex(3, 4)   # 3 + 4i
```

Operations

```
print(z1 + z2)   # complex(4, 6)
print(z1 * z2)   # complex(-5, 10)
```

9.4 Math constants

`stdlib` provides common mathematical constants:

- `pi` = 3.14159...
- `euler` = 2.71828...
- `tau` = 6.28318...

Example

```
print(pi)
print(euler)
```

9.5 Math functions

`stdlib` also provides common math functions:

- `sin(x)`
- `cos(x)`
- `tan(x)`
- `sqrt(x)`
- `log(x)`

Example

```
print(sin(pi / 2))    # 1
print(cos(0))         # 1
print(sqrt(16))       # 4
```

The End

Thank you for reading!
by **dinhsonhai132**