Chapter 4. Programming Basics

The way Scilab has been used in the previous chapters may give the impression that it is simply an "enhanced calculator" capable only of executing commands entered via the keyboard and displaying the result. In reality, in Scilab, programs can be written either as *scripts* or as *functions*, and this mode of operation is preferred as soon as the number of command lines becomes sufficiently large.

1. Scripts:

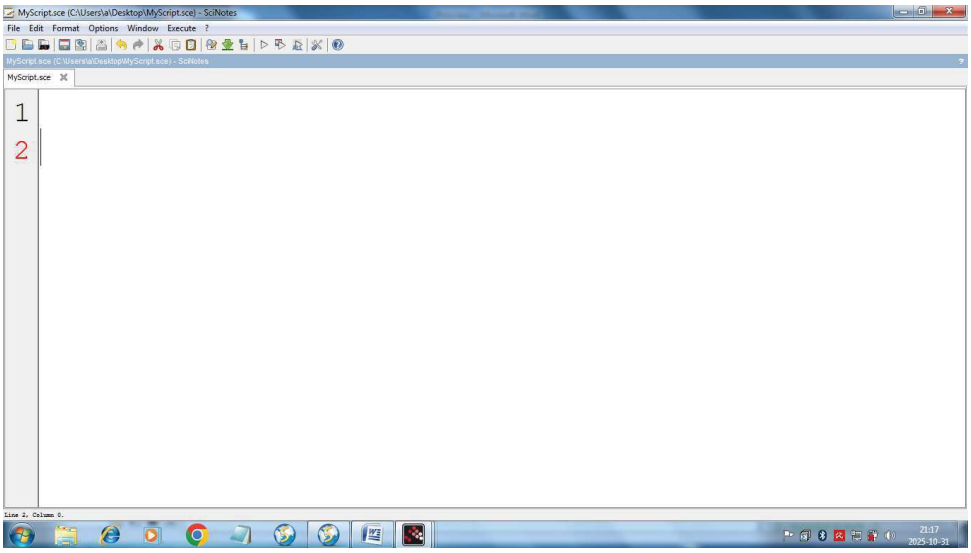
A *script* in Scilab is a text file that contains a list of commands written just as they would be entered in the console, and when executed, Scilab runs all these commands automatically in sequence. It is usually saved with the extension **.sce** and created using **SciNotes**, Scilab's integrated text editor. A script shares the same workspace as the main environment, meaning that all variables defined in it remain available after execution. Scripts are especially useful for repeating a set of commands or performing long calculations without having to retype them.

To accomplish this, you can launch the script editor directly from the console using the command:

```
--> scinotes
```

Alternatively, you can select **SciNotes** from the Scilab menu.

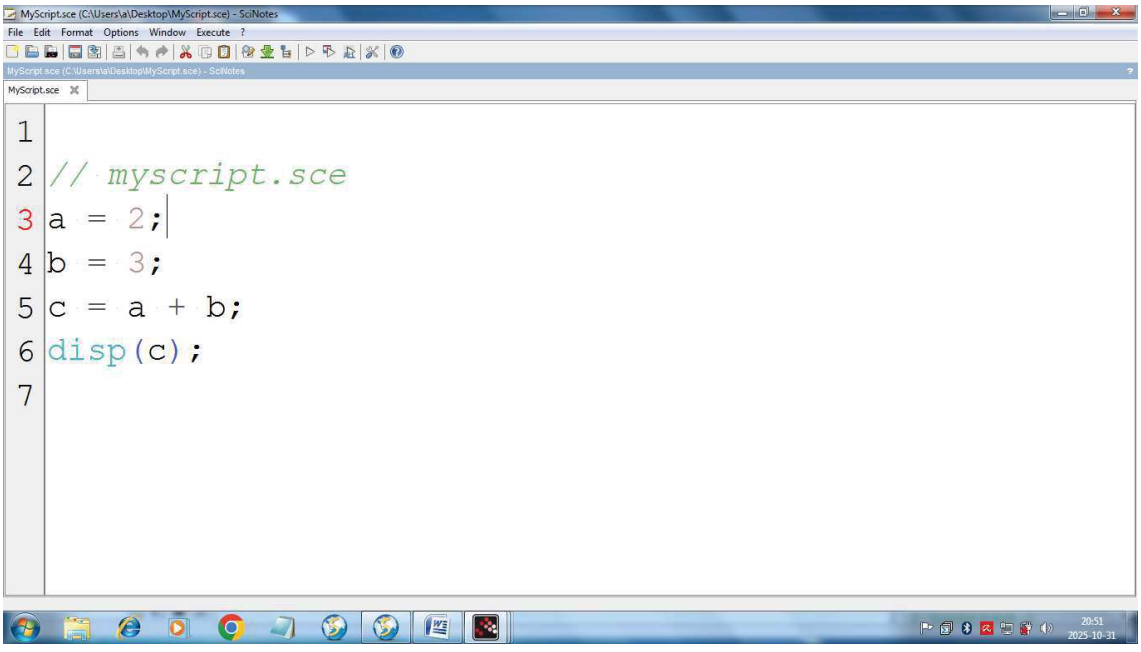
The file created is named **Untitled** by default. To change this name, simply select **Save As** from the **File** menu and choose the name **MyScript.sce**. The created window should look like this:



You can now enter the following lines in the editor window:

```
// MyScript.sce
a = 2;
b = 3;
c = a + b;
disp(c);
```

The created window should look like this:



Once the text has been entered in the window, save the file (Use the save button or by selecting **Save** from the **File** menu).

In a script, parentheses, loop endings, function closures, and test commands are automatically generated.

To execute the script "MyScript.sce", simply click on the button  , which should display the following lines in the command window:

```
--> exec('C:\Users\a\Desktop\MyScript.sce', -1)
   5.
```

You can also execute the file by selecting **…file with echo** from the **Execute** menu or by entering the following command in the console:

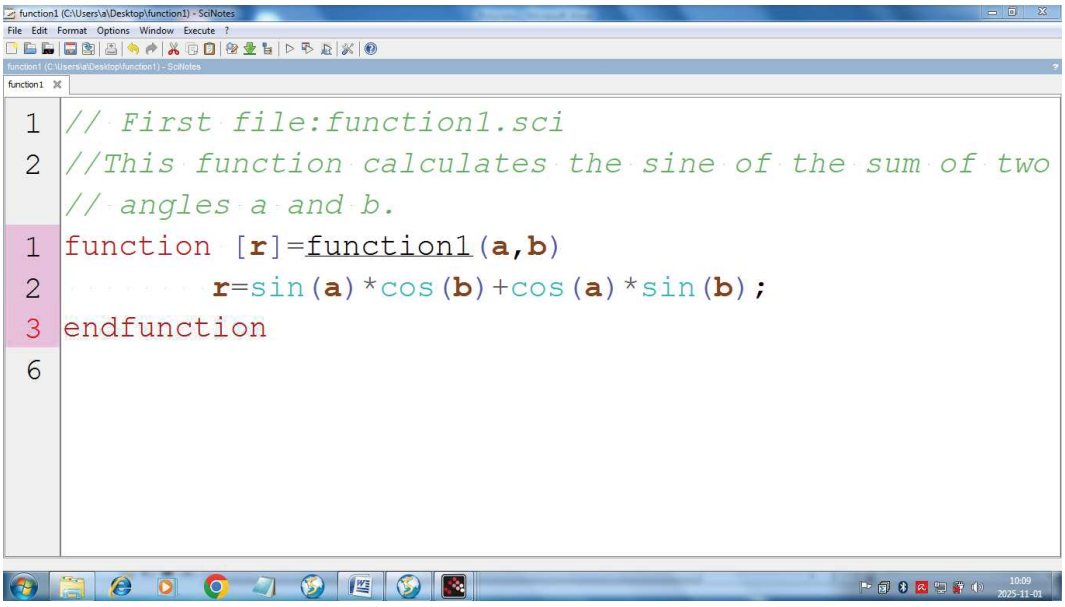
```
--> exec('C:\Users\a\Desktop\MyScript.sce')
```

2. Functions:

A *function* is a block of instructions stored in a **.sci** file that can take input arguments and return one or more results. Functions help structure and organize code, making it easier to manage complex or repeated computations. The general syntax for defining a function is as follows:

```
function [r1, r2, ..., rm] = function_name(arg1, arg2,..., argn)
    // Function body
    r1 = ...  // Value returned by r1
    ...
    rm = ...  // Value returned by rm
endfunction
```

As an example, let's create a function to calculate the **`sine`** of the sum of two angles **`a`** and **`b`**.



The file name can be different from the name of the function it defines, and a single file may contain several functions. The function is then loaded into the environment and can be executed.

```
--> exec('C:\Users\a\Desktop\function1')

--> function1(2,3)

ans =

  -0.9589243
```

It is also possible to define an "inline" function, i.e., directly from the Scilab command line. This is practical when the function is very short to write. For example:

```
--> deff("c=plus(a,b)","c=a+b");
--> plus(1,2)
 ans =
       3.
```

In Scilab, most of the time, we work with vectors and matrices. The operators and basic functions are designed to support this kind of manipulation and, more generally, to allow program vectorization. Of course, the Scilab language includes conditional operations (**`if`**–**`then`**–**`else`**, **`elseif`**), loops (**`while`**, **`for`**), and recursive programming. However, vectorization helps to reduce the use of these features, which are not very efficient in an interpreted language. The interpretation overhead can be dramatically penalizing compared to what a compiled **C** or **Fortran** program can achieve when performing mainly scalar calculations. Efficiency losses by a factor of **10** or even **100** can occur. Therefore, it is important to minimize the interpreter's workload by vectorizing programs as much as possible.

Control structures are instructions that define and manage the order of execution in a program. They allow the program to make decisions based on conditions or to repeat actions through loops for specific processes.

3. Control loops:

"for" Loop:

The **`for`** loop is commonly used for repetition, executing a block of instructions a predetermined number of times. Syntax:

```
for var = start : step : end
   instructions
end
//"var" takes values from "start" to "end", increasing
by "step".
```

For example:

```
--> for i = 1 : 5 : 20
  >     i*i
  > end
 ans =
   1.
 ans =
   36.
 ans =
   121.
 ans =
   256.
```

Alternatively, you can replace **start : step : end** with a vector. For example:

```
--> v=[1 3 7 8]
 v = [1x4 double]
   1.   3.   7.   8.
--> for i=v
  >     i/2
  > end
 ans =
   0.5
 ans =
   1.5
 ans =
   3.5
```

```
   ans =

     4.
```

Loops can also be used to define matrices:

```
--> n=4; a=zeros(n,n);

--> for i=1 : n-1

  >   a(i,i)=2;  a(i,i+1)=1; a(i+1,i)=-1;

  > end

--> a(n,n)=2;

--> a

 a = [4x4 double]

   2.   1.   0.   0.

  -1.   2.   1.   0.

   0.  -1.   2.   1.

   0.   0.  -1.   2.
```

Use **break** to exit a loop:

```
--> for i=0:0.01:1
  > test=i-exp(-i);
  > if (test>0) then
  > i
  > break
  > end
  > end
 i =
   0.5700000
```

"While" Loop:

The **while** loop executes instructions an indeterminate number of times, based on a logical condition. The syntax is:

```
while condition do
            instructions
    // instructions to execute while the condition is
true
End
```

For example**:**

```
--> // Example: Display numbers from 1 to 5
--> i = 1; // initialization
--> while i < 3 do
  >     disp(i);   // display the current value of i
  >     i = i + 1; // increment i
  > end
   1.
   2.
```

4. <u>Conditional statements:</u>

The "**if**" statement is the simplest and most commonly used control structure. It executes different actions based on whether a condition is true or false. The syntax of the if control structure is:

```
if condition then
  instructions
// statements executed when the condition is true
End
```

or:

```
if condition then
instructions //statements executed when the condition is true
else
Instructions //statements executed when the conditions is false
End
```

For example:

```
--> a=3.5;
--> if a>1 then
  >    c=log(a-1)
  > end
--> c
 c =
   0.9162907
```

Using `if`, `then`, and `else`: The "instructions1" are executed if the logical expression "condition" is true; otherwise, the "instructions2" are executed.

```
  -> a=0.9;
  --> if a>0 then
    > if a>1 then
    > f=1;
    > else
    > f=a*a;
    > end
    > else
    > f=0;
    > end
  --> f
   f =
     0.81
```

If **`else`** is followed by another **`if`**, you can use:

```
if condition then

      instructions
    // statements executed when the condition is true
elseif another_condition then
      instructions
    // statements executed when the second condition
    //is true
```

```
else
      instructions
    // statements executed when all conditions are
    // false
End
```

5. Reading and Displaying Variables (Input and Output):

The **input** command is used for user input. The syntax is:

```
variable = input('prompt text')
```

Example:

```
-> A=input('Enter the matrix A:  ')
Enter the matrix A:  [1,2;3,4]
 A = [2x2 double]
   1.   2.
   3.   4.
```

Second example:

```
--> name = input('Enter your name:  ', 'string')
Enter your name:  Scilab
 name =
  "Scilab"
```

The "string" option allows the input of character strings.

The **disp** command is used to display variables. Its syntax is:

```
disp(var)
```

Here, "var" can be a number, vector, matrix, string, or expression. Example:

```
--> t = 5;
--> disp("The solution is = ",(1 + sqrt(t)) / 2 )
  "The solution is = "
   1.6180340
```