CHAPTER 4

Loops, Functions, and Files

In Chapter 3, you learned about arrays, matrices, and vectors in GNU Octave in detail. You will use these concepts in the remaining chapters to demonstrate the functionality offered by GNU Octave.

In this chapter, you will learn concepts such as if statements, loops, functions, and file operations in detail. The following is the list of the topics you will learn and demonstrate:

- Decision making with `if` statements
- Loops in GNU Octave
- User-defined functions
- Global variables
- Working with files

Decision Making with If Statements

If you have worked with any programming language before, you will find this section and the next couple sections easy to comprehend. However, we don't recommend skipping anything because you must familiarize yourself with the GNU Octave syntax for the decision-making operations.

A. Pajankar and S. Chandu, *GNU Octave by Example*,
https://doi.org/10.1007/978-1-4842-6086-9_4

The `if` statement is the simplest and the most basic decision-making statement. It has been around since the days of machine-language and assembly-language programming, way before modern programming languages like C and BASIC came into existence.

It's time to learn how to use the `if` statement for decision making. You will use a Jupyter notebook for the demonstrations in this chapter.

The following statement creates a variable named as `x` and assigns the value `34` to it:

```
x = 34
```

Here's the syntax of the `if` statement:

```
if (rem(x, 2) == 0)
    printf("Even Number!\n")
endif
```

In this statement, you are comparing the remainder of x / 2 with the number 0. If they are equal, the expression returns true and the lines in the `if` – `endif` block are executed. Otherwise, GNU Octave just skips those lines. Run this code block and see the output.

Now let's add the `else` block. When the condition in the `if` statement is not satisfied, GNU Octave runs the `else` block. The following is an example:

```
if (rem(x, 2) == 0)
    printf("Even Number!\n")
else
    printf("Odd Number!\n")
endif
```

The function `rem()` in the code snippet above computes the remainder. Note that the `endif` statement comes after the `else` block and not after the `if` conditions alone.

Run this code and see the output. Try assigning different values for x and run the code to see both code blocks (`if` and `else`) in action.

If you want to evaluate multiple conditions, you can use an `elseif` clause in the `if` code block as follows:

```
x=25
if (rem(x, 2) == 0)
    printf("Divisible by 2!\n")
elseif(rem(x, 3) == 0)
    printf("Divisible by 3!\n")
else
    printf("Not Divisible by 2 or 3!\n")
endif
```

First, the `if` statement is checked. If it returns true, then the code under the `if` block is executed and the rest is skipped. If the statement in the `if` clause returns false, then the statement in the `elseif` is checked. If it returns true, then it runs the code block for the `elseif` and rest of the code is skipped. If the statement in the `elseif` returns false, then the code block in the `else` clause is executed. Run this code. The output is as follows:

```
Not Divisible by 2 or 3!
```

You can have multiple `elseif` clauses in the decision-making code. GNU Octave also has a `switch` statement for this kind of situation, and you can find out more about it at https://octave.org/doc/v4.2.1/The-switch-Statement.html.

Loops in GNU Octave

Let's see how to create loops in GNU Octave. Before modern programming languages, loops in assembly and machine languages were written using GOTO and IF statements. However, modern programming languages like

GNU Octave provide far more sophisticated and cleaner constructs for loops. Let's see the many ways of writing loops one by one.

Let's start with the `while` loop in Octave. The following is an example of the `while` loop:

```
x = 1;
while x <= 5
    printf("x ^ 2 is %d:\n", x**2)
    x = x + 1;
endwhile
```

The `while` statement always checks for the condition mentioned in it at the beginning of every iteration. If the condition is true, it runs all of the following statements in order until the statement `endwhile`, so this code prints the squares of the integer numbers from 1 to 5 as follows:

```
x ^ 2 is 1:
x ^ 2 is 4:
x ^ 2 is 9:
x ^ 2 is 16:
x ^ 2 is 25:
```

You must make sure that the code block in the `while` block has statements that will render the condition false at some time if you do not want the loop to run indefinitely. You can manually terminate the loop with a `break` statement. Here is the same code in a slightly different style:

```
x = 1;
while 1
    printf("x ^ 2 is %d:\n", x**2)
    x = x + 1;
    if x == 6
        break
    endif
endwhile
```

We mentioned the number 1 as the condition of the `while` loop. It always returns true. So, the `while` loop runs perpetually unless you explicitly break in the code block. In the code block, an `if` condition that checks equality of x with 6. When it is true, the `break` statement is executed and the while loop ends. We programmed it in this way to demonstrate the functionality of `break;` it is not usually done this way.

You can write the same program with the `do-until` construct. The following is an example:

```
x = 1;
do
    printf("x ^ 2 is %d:\n", x**2)
    x = x + 1;
until x > 5
```

In this code, the statements between `do` and `until` are executed in each iteration until the condition in the `until` statement is false. As soon as the condition is true, the loop is terminated. This loop also prints the squares of the integer numbers from 1 to 5. As an exercise, try adding the `break` statement in the loop above.

You can also write a `for` loop for the same output as follows:

```
for i = 1:5
    printf("i ^ 2 is %d:\n", i**2)
endfor
```

The statements between `for` and `endfor` are executed if the loop counter denoted by variable `i` is between 1 to 5. In the beginning of the `for` loop, `i` is set to 1, and in every iteration, it is incremented by 1 automatically until it is 5 and then the loop is terminated. Run the program and see the output yourself. The loop by default increments by 1. If you want to increment by any other value, say 2, you modify the `for` statement as follows:

```
for i = 1:2:10
```

The `i` in this case will be 1, 3, 5, 7, 9.

This is wraps up loops in GNU Octave. In the next section, you will see how to create user-defined functions in detail.

User-Defined Functions

Functions are nothing but subroutines. If you want to use a piece of code frequently in your program, you write it as subroutine. GNU Octave offers many built-in functions and packages to perform operations. You have seen quite a lot of built-in functions already, such as `rem()` and `printf()`. Now you will learn how to write custom functions. This is very handy when you want to write your own reusable code.

The input to any function is known an argument, and the output of a function is known as a return value. Here's an example a simple function that does not accept any input (arguments) and does not return any output:

```
function []= f0 ()
    printf("Test") ;
end
```

In this code example, the words `function` and `end` are keywords. This function prints the string `Test` when called. `f0()` is the name of the function. You can call it as follows:

```
f0()
```

This will run the function and print the string. You can create a function that returns value(s) too. When the function returns only a single value, the square brackets around the return value are not needed. The following is a function that returns the value of the pi with two decimal precision:

```
function y = f1 ()
    y = 3.14;
end
```

You can call it as follows:

```
f1()
```

You can also call it as follows by assigning the returned value to a variable:

```
a = f1()
```

Now, let's see an example of a function that accepts a couple of arguments and returns a single value. We added the square brackets around the return value to demonstrate how it is written this way. As mentioned, you can write it both ways (with or without square brackets) if the function returns single value.

```
function [y] = f2 (a, b)
    y = a + b;
end
```

The function `f2()` accepts two arguments and returns the addition of both. You can call this as follows:

```
f2(1, 2)
```

The other way to call this is

```
m = f2(1, 2)
```

You can have a function that returns multiple values as follows:

```
function [y1, y2] = f3 (a, b)
    y1 = a + b;
    y2 = a - b;
end
```

You can call this as follows:

```
[m, n] = f3(2, 4)
```

Then you can use the values of the variable `a` and `b` separately hereafter.

Another type of function is known as an inline function. An inline function has the keyword `inline`. The interpreter replaces the function call with the function code directly in an inline function. Here's an example of an inline function:

```
f0 = inline ("sqrt(x^2+y^2)") ;
```

You can call this function as follows:

```
f0(4, 3)
```

Inline functions are good for relatively simple functions that will not be used often in the program and that can be written in a single-line expression. Inline functions can only have one expression and can only return a single variable. The returned variable can be a multidimensional matrix.

Note Inline functions cannot access variables (including global variables) in the current session at any time.

Global Variables

Now that you have learned about functions, you can revisit the global variables from Chapter 2 to better understand their behavior.

A global variable may be accessed inside a function without passing it as a parameter. Passing a global variable to a function will make a local copy of the variable and not modify the global value.

```
global x = 0
function f(x)
    x = 1;
end
```

Notice that when you call

```
f(x)
```

it is

```
x = 1
```

But, when you print the value of x,

```
x
x = 0
```

As explained above, the local copy of the variable `x` is modified in the function `f(x)` but the global value remains the same.

Working with Files

Let's see how to work with files. You can read the data from files on the disk and store the data in files. Before you start with file-related programming demonstrations, you will learn how to run a few OS commands with a Jupyter notebook for Octave. You can even run Linux commands on the Jupyter notebook for Octave or the Octave Interactive console. This is because the GNU Octave interpreter can also interpret Linux commands. Let's see a few examples. You can see the present working directory with the following command:

```
pwd
```

The output is as follows:

```
ans = C:\Users\Ashwin\OneDrive\GNU Octave Book\First_Drafts\
Chapter04\programs
```

You can see the list of files in the current directory as follows:

```
dir
```

The output is as follows:

```
.                    .ipynb_checkpoints  test.xlsx
..                   Chapter04.ipynb
```

The files or folders that start with a . are hidden and not usually visible in the file explorer.

You can even use the Linux command `ls` to get the detailed output as follows:

```
Volume in drive C has no label.
Volume Serial Number is 9C4B-9156

Directory of C:\Users\Ashwin\OneDrive\GNU Octave Book\First_
Drafts\Chapter04\programs

[.]                  [.ipynb_checkpoints]
[..]                 Chapter04.ipynb
               1 File(s)          6,570 bytes
               3 Dir(s)  120,328,843,264 bytes free
```

Let's see a few file operations. First, create a matrix of size 5x5 as follows:

```
mat01 = rand (5, 5);
```

You can save this to a file as follows:

```
save file1.mat mat01
```

This command creates a file named `file1.mat` and saves it to that file. The .mat file extension is short for matrix, a data container format that is compatible with MATLAB and Octave. mat01 has values as follows:

```
 0.81598396769278381 0.92855422110525021 0.75365606653988848
0.50191794722525962 0.49488955306890497
 0.13756053717337141 0.91373377756306917 0.21944809091873169
0.86626249762210572 0.49854345466053068
 0.48677848511935479 0.90558318580210329 0.73794364985973382
0.37583995095818151 0.39386225963682803
 0.21045562411897317 0.32938941997464716 0.64352812535181725
0.69685526187959523 0.15707829430546633
 0.49126417869029831 0.21355975998368698 0.20118076472616681
0.047443232382045439 0.31718894583130069
```

Note that you are assigning random values to the matrix while creating it, so the contents of this file will be different for you. You can load this file as follows:

```
load file1.mat
```

This will load the data from the file in the variable name mentioned in the file. Since you have saved a matrix with `mat01` as the variable name, you can see the same variable name after loading. You can use this statement to load and use the data in a different notebook and program too. This is one of the best ways to save your working data like matrices, arrays, and vectors. Also, you can write custom programming APIs in other high-level languages to work with this data because it is formatted data.

You can store the values of the multiple variables to a file as follows:

```
m1 = rand(2, 2); m2 = rand(3, 3); m3 = rand(4, 4);
save ( "file2.mat" , "m1", "m2", "m3" )
```

Note that this is the plaintext format and you can assign any extension of your choice to these files.

You can load the variables into memory with the usual command:

```
load file2.mat
```

You will be able to access variables `m1`, `m2`, and m3 after this command. They will have the values stored in the file for the respective variables. You can save an array into a file in binary format with the following command:

```
save -binary binfile.bin m1
```

The contents of the file are binary, so opening this file in a text editor will show you incomprehensible ASCII characters. The best way to use it is to load it into memory as follows:

```
load binfile.bin
```

You can even save the data into a CSV (comma-separated value) file format. This format is a universal file format for saving tabular data. Here's how to save it in a CSV file:

```
a = [0 1 2; 3 4 5; 6 7 8]
csvwrite('test.csv', a)
```

This will create a CSV file and save the array there. The following are the contents of the file on the disc:

```
0,1,2
3,4,5
6,7,8
```

You can load it into a variable with the following statement:

```
a1 = csvread('test.csv')
```

You can even read a CSV file hosted online into a matrix with the following command:

```
a = urlread('http://samplecsvs.s3.amazonaws.com/
Sacramentorealestatetransactions.csv')
```

If you wish to store this online file in a local file on the disc, it can be done with the following command:

```
urlwrite('http://samplecsvs.s3.amazonaws.com/
Sacramentorealestatetransactions.csv', 'local_copy.csv')
```

You can also load and save Excel files (`.xlsx`). For this to work, you need to download the `io` package from https://octave.sourceforge.io/packages.php.

Before you proceed, make sure you have the paths set correctly. In Windows, add the following two paths to your `Path` variable:

```
Path_to_Octave_Installation\usr\bin
Path_to_Octave_Installation\mingw64\bin
```

In Ubuntu, in the terminal before launching the Jupyter notebook, run the following command:

```
sudo apt-get install liboctave-dev
```

Now, you must install and load the `io` package via the following commands:

```
pkg install io.tar.gz
```

Ignore any warnings after this step.

```
pkg load io
```

Continuing with a similar example as when you experimented with CSV files, type the following commands to see for yourself how working with Excel works in Octave:

```
a = [0 1 2; 3 4 5; 6 7 8]
xlswrite('test.xlsx', a)
```

This will create an Excel file and save the array there. The following are the contents of the file on the disc:

```
0,1,2
3,4,5
6,7,8
```

You can load it into a variable with the following statement:

```
a1 = xlsread('test.xlsx')
```

This is how to work with Excel.

Summary

In this chapter, you learned how to write decision-making programs with `if` statements. You also learned how to write loops. You learned how to write user-defined functions and briefly explored global variables. At the end, you learned the important concept of working with various file formats like CSV and Excel.

In the next chapter, you will see how to visualize data with GNU Octave.