CHAPTER 3

Data Types and Variables in Detail

In Chapter 2, you learned basic concepts like naming conventions for variables, mathematical operations, getting help, and clearing the command prompt. You also saw an overview of global variables.

In this chapter, you will explore the concepts of data types and variables in detail. The following is the list of the topics you will learn and demonstrate:

Data Types in GNU Octave

Let's create a new Jupyter notebook file for GNU Octave. You will save all the demonstrations for this chapter in this notebook.

Convert the first cell to markdown, as you did in Chapter 2, and type in and run the following code to create a heading and a sub-heading:

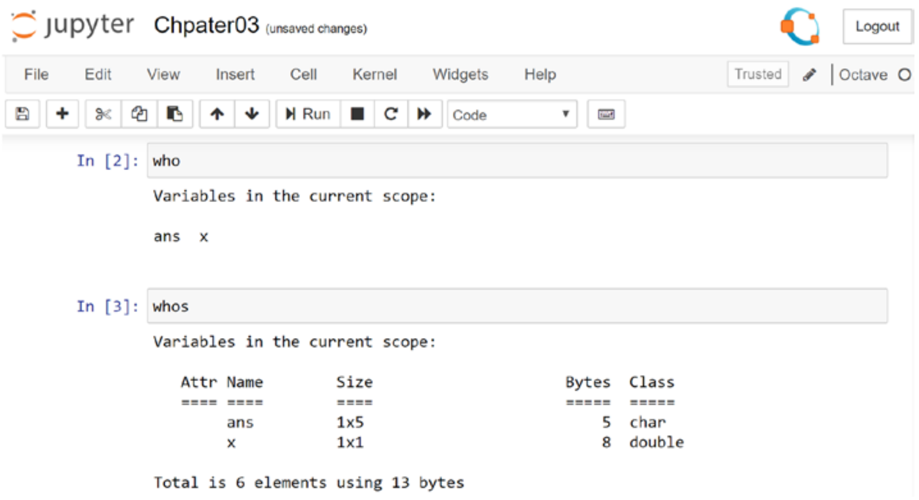
```
# Data Types
## Basics
```

After this, create a simple variable as follows:

```
x = 10
```

https://doi.org/10.1007/978-1-4842-6086-9_3

You know that all variables in the current scope can be seen by the command who and you can learn the details of the variables with the command whos, as shown in Figure 3-1.


jupyter Chpater03 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Octave
Run
Code

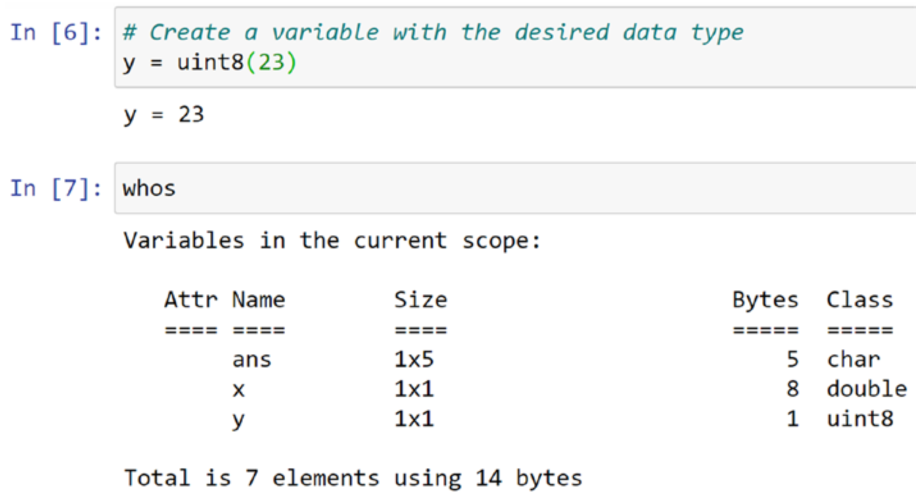
```
In [2]: who
        Variables in the current scope:

        ans  x


In [3]: whos
        Variables in the current scope:

          Attr Name        Size                     Bytes  Class
          ==== ====        ====                     =====  =====
               ans         1x5                          5  char
               x           1x1                          8  double

        Total is 6 elements using 13 bytes
```

Figure 3-1. *Output of the commands who and whos*

If you notice in Figure 3-1, all of the variables are matrices by default and all of the numeric variables are double by default (you'll look at doubles a little later in this section). However, the type double requires a lot of memory. There are four signed integer types of data: int8, int16, int32, and int64. They require 1, 2, 4, and 8 bytes, respectively. Similarly, uint8, uint16, uint32, and uint64 are unsigned integer data types and they also require 1, 2, 4, and 8 bytes, respectively. You can create a variable with the desired data type as follows:

```
# Create a variable with the desired data type
y = uint8(23)
```

In this code, the code comment starts with #. Whenever the GNU Octave interpreter encounters #, it treats it as a code comment and ignores the text after it on that line. The output after running the code above and the command whos is shown in Figure 3-2.



```
In [6]: # Create a variable with the desired data type
        y = uint8(23)

        y = 23

In [7]: whos

        Variables in the current scope:

          Attr Name        Size                     Bytes  Class
          ==== ====        ====                     =====  =====
               ans         1x5                          5  char
               x           1x1                          8  double
               y           1x1                          1  uint8

        Total is 7 elements using 14 bytes
```

Figure 3-2. *A variable of the type uint8*

As an exercise, create variables of the other integer data types discussed above.

Floating numbers are represented by double and single precision formats. The single precision occupies 4 bytes. Out of these 4 bytes (32 bits), 23 bits are used to store the fraction, 8 bits are used for the exponent, and 1 bit is used for the sign. The double precision occupies 8 bytes. Out of these eight bytes (64 bits), 52 bits are used for the fraction, 11 bits are used for the exponent, and 1 bit is used for the sign. The following is an example of the same:

```
# Single and double precision floats
a = single(3.14)
b = double(3.14)
```

Run this code and then the command whos to see the details of the variables you created.

In the last chapter, you learned that the letters i, j, I, and J are used to represent $\sqrt{(-1)}$, which is an imaginary number. Using this number, you can define complex numbers. Run the following code:

```
# defining and understanding complex numbers
z = 2 + 3I
real(z)
imag(z)
```

The first line defines a complex number. The next two lines return the real and the imaginary part of the complex number. By default, the complex number is a double. You can explicitly define a single or a double precision complex number as follows:

```
z1 = single(2 + 3I)
z2 = double(2 + 3I)
```

Run the command `whos` after this to see the datatype of these complex numbers.

You can also create character strings as follows:

```
s1 = "Hello World!"
```

These are the basic data types in GNU Octave. In the next section, you will learn how to work with the multidimensional data structures known as arrays.

Arrays, Vectors, and Matrices

Just like in the programming languages C and C++, you can create and use arrays in GNU Octave. An array is collection of variables of the same datatype that are stored in continuous memory locations. Arrays can have one or more dimensions. Multi-dimensional arrays are usually called matrices. Let's see examples of arrays. Convert a cell to markdown and type the following code to create a heading:

```
# Arrays and Vectors
```

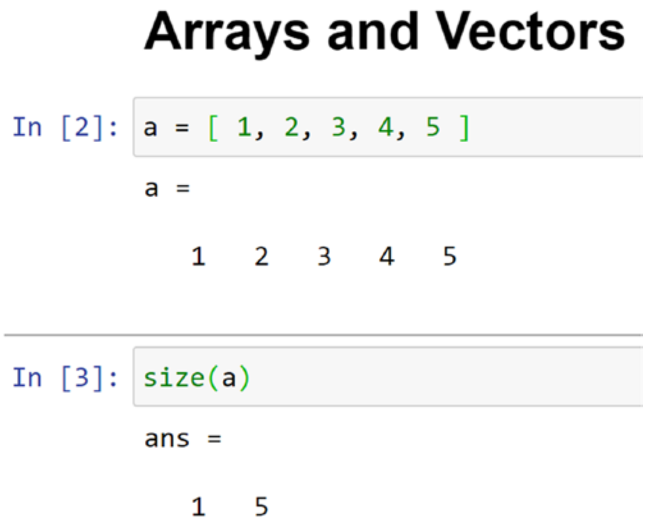
Then type and execute the following code in two different cells:

```
a = [ 1, 2, 3, 4, 5 ]
size(a)
```

The output is shown in Figure 3-3. The

```
ans = 1 5
```

means that the matrix a has one row and five columns.



Arrays and Vectors

```
In [2]: a = [ 1, 2, 3, 4, 5 ]

a =

   1   2   3   4   5

In [3]: size(a)

ans =

   1   5
```

Figure 3-3. *A simple array*

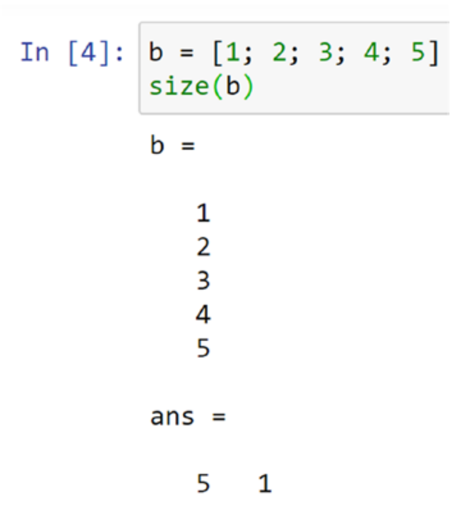
You can also declare the above array as

```
a = [ 1 2 3 4 5 ]
```

The above array has only one row. You can similarly create an array with a single column as follows:

```
b = [1; 2; 3; 4; 5]
size(b)
```

The semicolon (;) is the delimiter for rows. The output is shown in Figure 3-4.



```
In [4]: b = [1; 2; 3; 4; 5]
        size(b)

        b =

           1
           2
           3
           4
           5

        ans =

           5   1
```

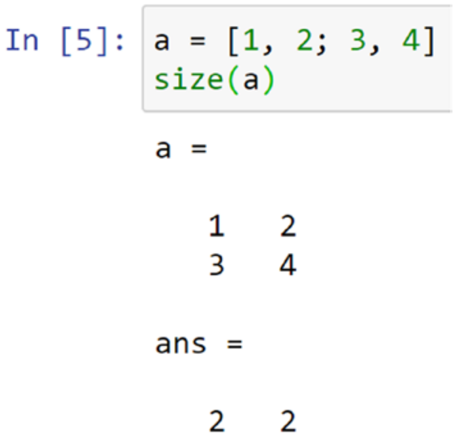
Figure 3-4. *A simple array with a single column*

In GNU Octave, a vector is a matrix with either one row or one column. The above are examples of vectors.

You can even create 2D matrices as follows:

```
a = [1, 2; 3, 4]
size(a)
```

The output is shown in Figure 3-5.



```
In [5]: a = [1, 2; 3, 4]
        size(a)

        a =

           1   2
           3   4

        ans =

           2   2
```

Figure 3-5. *A 2x2 2D matrix*

In the case of 2D or multi-dimensional matrices, the number of elements in every row must be equal. Otherwise the GNU Octave interpreter throws an error as follows:

```
a = [1, 2; 3, 4, 5]
error: vertical dimensions mismatch (1x2 vs 1x3)
```

If you run the command `whos`, you can see that the default datatype of all the arrays is double. You can create the arrays, vectors, and matrices of any custom datatype as follows:

```
a = int16([1, 2, 3])
b = int8([1; 2; 3])
c = int32([1, 2; 3, 4])
```

Indexing in Arrays

Let's use the above examples to understand indexing. Indexing starts from 1 in Octave. In C and C++, it starts from 0. So, if you have a lot of experience with C and C++ programming, be careful. You can retrieve the first element in the array a in the following ways:

```
a(1)
a(1, 1)
```

The second element can be retrieved in the following ways:

```
a(2)
a(2, 1)
```

You can retrieve the elements of a column vector as follows:

```
b(1)
b(1, 1)
b(2)
b(2, 1)
```

For the 2D matrix c, you can retrieve the elements as follows:

```
c(1, 1)
c(1, 2)
c(2, 1)
c(2, 2)
```

Operations on Arrays

You can perform mathematical operations on numerical arrays. Let's see a few operations. Create two arrays as follows:

```
a = [0, 1; 2, 3]
b = [4, 5; 6, 7]
```

Let's perform a few operations with an array as one operand and a numerical constant as the other operand:

```
a + 5
a - 3
7 - a
a * 2
a ** 2
a ^ 2
a / 2
a % 2
```

Let's perform a few operations with arrays as both operands:

```
a + b
a - b
b - a
a * b
a / b
a % b
```

Array Creation Routines

There are many array creation routines in GNU Octave. The function `ones()` creates a matrix of ones. The following is an example:

```
ones ( 3, 3 )
```

The function `zeros()` creates a matrix of zeros. The following is an example:

```
zeros ( 3, 3 )
```

The function `eye()` creates an identity matrix (a matrix with all of the diagonal elements as 1s and the rest as 0s). The following is an example:

```
eye ( 3, 3 )
```

The function `rand()` creates a matrix of random numbers. The following is an example:

```
rand ( 5, 5 )
```

Run the above examples and see the output.

Let's see two more functions and their respective output. The function `linspace(base, limit, n)` accepts the lower and upper limits, and creates an array with `n` linearly spaced elements. The following is an example:

```
linspace( 1, 10, 4 )
```

The output is as follows:

```
ans = 1    4    7   10
```

The function `logspace (base, limit, n)` accepts the lower and upper limits and creates an array with `n` logarithmically spaced elements. The following is an example:

```
logspace( 1, 5, 5 )
```

The output is as follows:

```
ans = 10        100       1000      10000     100000
```

You can assign them to variables or you can directly display their values.

As an exercise, try passing different values to these array creation functions.

Matrix Manipulation Function

Let's see a few matrix manipulation functions. Create a 2D matrix as follows:

```
a = [ 1 2 3; 4 5 6; 7 8 9 ]
```

You will use this matrix for the demonstrations of all of the matrix manipulation functions that we are going to discuss in this section. Using `'` after a variable name computes the transpose of the matrix:

```
a'
```

The output is as follows:

```
   1   4   7
   2   5   8
   3   6   9
```

You can compute the determinant of the matrix with `det(a)`.

You can flip matrices in the various ways with the following functions:

```
flip(a)
fliplr(a)
flipud(a)
```

The function `fliplr()` flips the matrix left to right and the function `flipud()` flips up to down. You can convert a matrix into an upper and lower triangular matrix with the following functions:

```
triu(a)
tril(a)
```

Run these function calls and see the output.

Summary

In this chapter, you examined the data types in GNU Octave. You studied and demonstrated arrays and operations on them. You learned about operations on arrays and matrices. You will use many of these operations on matrices when working with images. Images are represented as multi-dimensional arrays or matrices in GNU Octave. You will also use these concepts when you study data visualization.

In the next chapter, you will explore loops, conditional statements, and functions in GNU Octave in detail.