**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **07-07-2020** | **Name:** | **Karthik J** |
| **Course:** | MATLAB Onramp from MathWorks | **USN:** | **4AL16EC030** |
| **Topic:** | Indexing into and  modifying arrays,  array calculations,  calling functions, | **Semester & Section:** | **8TH A** |
| **GitHub Repository:** | Karthik-J |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image Section**   MATLAB - Arrays All variables of all data types in MATLAB are multidimensional arrays. A vector is a one-dimensional array and a matrix is a two-dimensional array.  We have already discussed vectors and matrices. In this chapter, we will discuss multidimensional arrays. However, before that, let us discuss some special types of arrays. Special Arrays in MATLAB In this section, we will discuss some functions that create some special arrays. For all these functions, a single argument creates a square array, double arguments create rectangular array.  The **zeros()** function creates an array of all zeros −  For example −  zeros(5)  MATLAB will execute the above statement and return the following result −  ans =  0 0 0 0 0  0 0 0 0 0  0 0 0 0 0  0 0 0 0 0  0 0 0 0 0  The **ones()** function creates an array of all ones −  For example −  ones(4,3)  MATLAB will execute the above statement and return the following result −  ans =  1 1 1  1 1 1  1 1 1  1 1 1  The **eye()** function creates an identity matrix.  For example −  eye(4)  MATLAB will execute the above statement and return the following result −  ans =  1 0 0 0  0 1 0 0  0 0 1 0  0 0 0 1  The **rand()** function creates an array of uniformly distributed random numbers on (0,1) −  For example −  rand(3, 5)  MATLAB will execute the above statement and return the following result −  ans =  0.8147 0.9134 0.2785 0.9649 0.9572  0.9058 0.6324 0.5469 0.1576 0.4854  0.1270 0.0975 0.9575 0.9706 0.8003 A Magic Square A **magic square** is a square that produces the same sum, when its elements are added row-wise, column-wise or diagonally.  The **magic()** function creates a magic square array. It takes a singular argument that gives the size of the square. The argument must be a scalar greater than or equal to 3.  magic(4)  MATLAB will execute the above statement and return the following result −  ans =  16 2 3 13  5 11 10 8  9 7 6 12  4 14 15 1 Multidimensional Arrays An array having more than two dimensions is called a multidimensional array in MATLAB. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix.  Generally to generate a multidimensional array, we first create a two-dimensional array and extend it.  For example, let's create a two-dimensional array a.  a = [7 9 5; 6 1 9; 4 3 2]  MATLAB will execute the above statement and return the following result −  a =  7 9 5  6 1 9  4 3 2  The array a is a 3-by-3 array; we can add a third dimension to a, by providing the values like −  a(:, :, 2)= [ 1 2 3; 4 5 6; 7 8 9]  MATLAB will execute the above statement and return the following result −  a =  ans(:,:,1) =  0 0 0  0 0 0  0 0 0  ans(:,:,2) =  1 2 3  4 5 6  7 8 9  We can also create multidimensional arrays using the ones(), zeros() or the rand() functions.  For example,  b = rand(4,3,2)  MATLAB will execute the above statement and return the following result −  b(:,:,1) =  0.0344 0.7952 0.6463  0.4387 0.1869 0.7094  0.3816 0.4898 0.7547  0.7655 0.4456 0.2760  b(:,:,2) =  0.6797 0.4984 0.2238  0.6551 0.9597 0.7513  0.1626 0.3404 0.2551  0.1190 0.5853 0.5060  We can also use the **cat()** function to build multidimensional arrays. It concatenates a list of arrays along a specified dimension −  Syntax for the cat() function is −  B = cat(dim, A1, A2...)  Where,   * B is the new array created * A1, A2, ... are the arrays to be concatenated * dim is the dimension along which to concatenate the arrays  Example Create a script file and type the following code into it −  a = [9 8 7; 6 5 4; 3 2 1];  b = [1 2 3; 4 5 6; 7 8 9];  c = cat(3, a, b, [ 2 3 1; 4 7 8; 3 9 0])   |  |  |  |  | | --- | --- | --- | --- | | **Date:** | **07-07-2020** | **Name:** | **Karthik J** | | **Course:** | Cisco Networking Academy | **USN:** | **4AL16EC030** | | **Topic:** | Introduction to IoT | **Semester & Section:** | **8TH A** | | **GitHub Repository:** | Karthik-J |  |  |       [**Everything Becomes Programmable**](https://static-course-assets.s3.amazonaws.com/I2IoT20/en/index.html#2)  [**Basic Programming Concepts**](https://static-course-assets.s3.amazonaws.com/I2IoT20/en/index.html#2.1.1) Programming Variables Programming languages utilize variables as dynamic buckets to hold phrases, numbers, or other important information that can be used in coding. Instead of repeating specific values in numerous places throughout the code, a variable can be used. Variables can hold the result of a calculation, the result of a database query, or some other value. This means that the same code will function using different pieces of data without having to be rewritten.  For instance “x + y = z” is an example of a programming expression. In this expression, x, y and z are variables which can represent characters, character strings, numeric values or memory addresses.  A variable can refer to a value. For instance the expression “a = 10” associates the value 10 to variable a.  A variable can also represent a memory location. The expression “a = 10” represents that the value 10 is stored in some location of the computer memory, which is referred to as ‘a’.  Variables can be classified into two categories:   * **Local Variables** - These are variables that are within the scope of a program / function / procedure. * **Global Variables** - These are variables that are in the scope for the time of the program’s execution. They can be retrieved by any part of the program.   Variables allow programmers to quickly create a wide range of simple or complex programs which tell the computer to behave in a pre-defined fashion. Basic Program Structures People impart logic to computers through programs. Using specific logic structures, a programmer can prepare a computer to make decisions. The most common logic structures are:   * **IF – THEN** - This logic structure allows the computer to make a decision based on the result of an expression. An example of an expression is myVar > 0. This expression is true if the value stored in the myVar variable is greater than zero. When an IF-THEN structure is encountered, it evaluates the provided expression. If the expression is false, the computer moves on to the next structure, ignoring the contents of the IF-THEN block. If the expression is true, the computer executes the associated action before moving on to the next instruction in the program. * **FOR Loops** – These are used to execute a specific set of instructions a specific number of times, based on an expression. The term loop comes from the fact that the set of instructions is executed repeatedly. While the syntax of FOR loops varies from language to language, the concept remains the same. A variable acts as a counter inside a range of values identified by a minimum and a maximum. Every time the loop is executed, the counter variable is incremented. When the counter is equal to the defined maximum value, the loop is abandoned and the execution moves on to the next instruction. * **WHILE Loops** – These are used to execute a specific set of instructions while an expression is true. Notice that often the instructions inside the loop will eventually make the expression evaluate as false. |