```
>> mat(:,4) = [9 2]'
mat =
     2    11     4     9
     5     6     7     2
```

Just as we saw with vectors, if there is a gap between the current matrix and the row or column being added, MATLAB will fill in with zeros.

```
>> mat(4,:) = 2:2:8
mat =
     2    11     4     9
     5     6     7     2
     0     0     0     0
     2     4     6     8
```

1.5.4 Dimensions

The **length** and **size** functions in MATLAB are used to find array dimensions. The **length** function returns the number of elements in a vector. The **size** function returns the number of rows and columns in a matrix. For a matrix, the **length** function will return either the number of rows or the number of columns, whichever is largest. For example, the following vector, *vec*, has four elements so its length is 4. It is a row vector, so the size is 1 × 4.

```
>> vec = -2:1
vec =
   -2  -1   0   1

>> length(vec)
ans =
   4

>> size(vec)
ans =
   1   4
```

For the matrix *mat* shown next, it has three rows and two columns, so the size is 3 × 2. The length is the larger dimension, 3.

```
>> mat = [1:3; 5:7]'
mat =
   1     5
   2     6
   3     7
>> size(mat)
ans =
   3   2
```

```
>> length(mat)
ans =
   3

>> [r c] = size(mat)
r =
   3
c =
   2
```

Note: The last example demonstrates a very important and unique concept in MATLAB: the ability to have a vector of variables on the left-hand side of an assignment.

The **size** function returns two values, so in order to capture these values in separate variables we put a vector of two variables on the left of the assignment. The variable *r* stores the first value returned, which is the number of rows, and *c* stores the number of columns.

QUICK QUESTION!

How could you create a matrix of zeros with the same size as another matrix?

Answer:

For a matrix variable *mat*, the following expression would accomplish this:

```
zeros(size(mat))
```

The **size** function returns the size of the matrix, which is then passed to the **zeros** function, which then returns a matrix of zeros with the same size as *mat*. It is not necessary in this case to store the values returned from the **size** function.

MATLAB also has a function, **numel**, which returns the total number of elements in any array (vector or matrix):

```
>> vec = 9:-2:1
vec =
     9     7     5     3     1

>> numel(vec)
ans =
     5

>> mat = randint(2,3,[1,10])
mat =
     7          9          8
     4          6          5

>> numel(mat)
ans =
     6
```

For vectors, this is equivalent to the length of the vector. For matrices, it is the product of the number of rows and columns.

MATLAB also has a built-in expression **end** that can be used to refer to the last element in a vector; for example, v(end) is equivalent to v(length(v)). For matrices, it can refer to the last row or column. So, using **end** for the row index would refer to the last row. In this case, the element referred to is in the first column of the last row:

```
>> mat = [1:3; 4:6]'
mat =
     1     4
     2     5
     3     6
>> mat(end,1)
ans =
     3
```

Using **end** for the column index would refer to the last column (e.g., the last column of the second row):

```
>> mat(2,end)
ans =
     5
```

This can be used only as an index.

1.5.4.1 Changing Dimensions

In addition to the transpose operator, MATLAB has several built-in functions that change the dimensions or configuration of matrices, including **reshape**, **fliplr**, **flipud**, and **rot90**.

The **reshape** function changes the dimensions of a matrix. The following matrix variable *mat* is 3×4, or in other words it has 12 elements.

```
>> mat = randint(3,4,[1 100])
mat =
    14    61     2    94
    21    28    75    47
    20    20    45    42
```

These 12 values instead could be arranged as a 2×6 matrix, 6×2, 4×3, 1×12, or 12×1. The **reshape** function iterates through the matrix columnwise. For example, when reshaping *mat* into a 2×6 matrix, the values from the first column in the original matrix (14, 21, and 20) are used first, then the values from the second column (61, 28, 20), and so forth.

```
>> reshape(mat,2,6)
ans =
    14    20    28     2    45    47
    21    61    20    75    94    42
```

The **fliplr** function "flips" the matrix from left to right (in other words the left-most column, the first column, becomes the last column and so forth), and the **flipud** functions flips up to down. Note that in these examples *mat* is unchanged; instead, the results are stored in the default variable *ans* each time.

```
>> mat = randint(3,4,[1 100])
mat =
    14    61     2    94
    21    28    75    47
    20    20    45    42

>> fliplr(mat)
ans =
    94     2    61    14
    47    75    28    21
    42    45    20    20

>> mat
mat =
    14    61     2    94
    21    28    75    47
    20    20    45    42

>> flipud(mat)
ans =
    20    20    45    42
    21    28    75    47
    14    61     2    94
```

The **rot90** function rotates the matrix counterclockwise 90 degrees, so for example the value in the top-right corner becomes instead the top-left corner and the last column becomes the first row:

```
>> mat
mat =
    14    61     2    94
    21    28    75    47
    20    20    45    42

>> rot90(mat)
```

```
ans =
     94     47     42
      2     75     45
     61     28     20
     14     21     20
```

QUICK QUESTION!

Is there a **rot180** function? Is there a **rot-90** function (to rotate clockwise)?

Answer:

Not exactly, but a second argument can be passed to the **rot90** function, which is an integer *n*; the function will rotate 90**n* degrees. The integer can be positive or negative. For example, if 2 is passed, the function will rotate the matrix 180 degrees (so, it would be the same as rotating the value of *ans* another 90 degrees).

```
>> mat
mat =
     14     61      2     94
     21     28     75     47
     20     20     45     42
>> rot90(mat,2)
ans =
     42     45     20     20
     47     75     28     21
     94      2     61     14
```

If a negative number is passed for *n*, the rotation would be in the opposite direction, in other words, clockwise.

```
>> mat
mat =
     14     61      2     94
     21     28     75     47
     20     20     45     42
>> rot90(mat,-1)
ans =
     20     21     14
     20     28     61
     45     75      2
     42     47     94
```

The function **repmat** can also be used to create a matrix; **repmat(mat,m,n)** creates a larger matrix, which consists of an $m \times n$ matrix of copies of mat. For example, here is a 2 × 2 random matrix:

```
>> intmat = randint(2,2,[0 100])
intmat =
    100     77
     28     14
```

The function **repmat** can be used to replicate this matrix six times as a 3 × 2 matrix of the variable *intmat*.

```
>> repmat(intmat,3,2)
ans =
    100     77    100     77
     28     14     28     14
    100     77    100     77
```

```
 28          14          28          14
100          77         100          77
 28          14          28          14
```

1.5.5 Using Functions with Vectors and Matrices

Since MATLAB is written to work with vectors and matrices, an entire vector or matrix can be passed as an argument to a function. MATLAB will evaluate the function on every element, and return as a result a vector or matrix with the same dimensions as the original. For example, we could pass the following vector, *vec*, to the **abs** function in order to get the absolute value of every element.

```
>> vec = -3:4
vec =
    -3     -2     -1      0      1      2      3      4

>> abs(vec)
ans =
     3      2      1      0      1      2      3      4
```

The original vector *vec* has eight elements, and since the **abs** function is evaluated for every element, the resulting vector also has eight elements.

This also would be the case for matrices:

```
>> mat = randint(2,3,[-5,5])
mat =
     -5     -1      0
      3      5     -1

>> abs(mat)
ans =
     5      1      0
     3      5      1
```

We will see much more on operations and functions of arrays (vectors and matrices) in Chapters 4 and 11.

1.5.6 Empty Vectors

An ***empty vector***, or, in other words, a vector that stores no values, can be created using empty square brackets:

```
>> evec = []
evec =
     []

>> length(evec)
ans =
  0
```

Then, values can be added to the vector by ***concatenating***, or adding values to the existing vector. The following statement takes what is currently in *evec*, which is nothing, and adds a 4 to it.

```
>> evec = [evec 4]
evec =
     4
```

The following statement takes what is currently in *evec*, which is 4, and adds an 11 to it.

```
>> evec = [evec 11]
evec =
     4  11
```

This can be continued as many times as desired, in order to build a vector up from nothing.

Empty vectors can also be used to ***delete elements from arrays***. For example, to remove the third element from an array, the empty vector is assigned to it:

```
>> vec = 1:5
vec =
    1     2     3     4     5

>> vec(3) = []
vec =
    1     2     4     5
```

The elements in this vector are now numbered 1 through 4.

Subsets of a vector could also be removed; for example:

```
>> vec = 1:8
vec =
    1     2     3     4     5     6     7     8

>> vec(2:4) = []
vec =
    1     5     6     7     8
```

Individual elements cannot be removed from matrices, since matrices always have to have the same number of elements in every row.

```
>> mat = [7 9 8; 4 6 5]
mat =
     7     9     8
     4     6     5
>> mat(1,2) = [];
??? Indexed empty matrix assignment is not allowed.
```

However, entire rows or columns could be removed from a matrix. For example, to remove the second column:

```
>> mat(:,2) = []
mat =
     7     8
     4     5
```

SUMMARY

Common Pitfalls

It is common when learning to program to make simple spelling mistakes and to confuse the necessary punctuation. Following are examples of very common errors:

- Putting a space in a variable name
- Confusing the format of an assignment statement as

  ```
  expression = variablename
  ```

  rather than

  ```
  variablename = expression
  ```

  The variable name must always be on the left.
- Using a built-in function name as a variable name, and then trying to use the function
- Confusing the two division operators / and \
- Forgetting the operator precedence rules
- Confusing the order of arguments passed to functions, for example, to find the remainder of dividing 3 into 10 using **rem(3,10)** instead of **rem(10,3)**
- Not using different types of arguments when testing functions
- Attempting to create a matrix that does not have the same number of values on each row
- Forgetting to use parentheses to pass an argument to a function; for example, **fix 2.3** instead of **fix(2.3)**. MATLAB returns the ASCII equivalent for each character when this mistake is made. (What happens is that is that it is interpreted as the function of a string; for example, **fix('2.3')**).

Programming Style Guidelines

Following these guidelines will make your code much easier to read and understand, and therefore easier to work with and modify.

- Use mnemonic variable names (names that make sense; for example, *radius* instead of *xyz*).

PRACTICE 1.7

Think about what would be produced by the following sequence of statements and expressions, and then type them to verify your answers.

```
m = [1:4; 3 11 7 2]
m(2,3)
m(:,3)
m(4)
size(m)
numel(m)
reshape(m,1,numel(m))
vec = m(1,:)
vec(2) = 5
vec(3) = []
vec(5) = 8
vec = [vec 11]
```

- Do not use names of built-in functions as variable names.
- If different sets of random numbers are desired, set the seed for the **rand** function.
- Do not use just a single index when referring to elements in a matrix; instead, use both the row and column indices.
- To be general, never assume that the dimensions of any array (vector or matrix) are known. Instead, use the function **length** to determine the number of elements in a vector, and the function **size** for a matrix, for example:

```
len = length(vec);
[r c] = size(mat);
```

MATLAB Functions and Commands			
info	floor	double	linspace
demo	ceil	int8	zeros
help	round	int16	length
lookfor	rem	int32	size
namelengthmax	sign	int64	numel
who	pi	intmin	end
whos	i	intmax	reshape
clear	j	char	fliplr
format	inf	logical	flipud
sin	exp	rand	rot90
abs	NaN	clock	repmat
fix	single	randint	

MATLAB Operators			
assignment =	multiplication *	divided into \	colon:
addition +	exponentiation ^	parentheses ()	transpose ‘
subtraction –	divided by /	negation -	

Exercises

1. Create a variable, *myage*, and store your age in it. Subtract one from the value of the variable. Add two to the value of the variable.
2. Use the built-in function **namelengthmax** to find out the maximum number of characters that you can have in an identifier name under your version of MATLAB.
3. Explore the **format** command in more detail. Use **help format** to find options. Experiment with **format bank** to display dollar values.
4. Find a **format** option that would result in the following output format:

```
>> 5/16 + 2/7
ans =
      67/112
```

5. Think about what the results would be for the following expressions, and then type them to verify your answers.

```
25 / 4 * 4
```

```
3 + 4 ^ 2
4 \ 12 + 4
   3 ^ 2
(5 - 2) * 3
```

6. Create a variable, *pounds*, to store a weight in pounds. Convert this to kilograms and assign the result to a variable *kilos*. The conversion factor is 1 kilogram = 2.2 pounds.
7. The combined resistance RT of three resistors R_1, R_2, and R_3 in parallel is given by

$$RT = \frac{1}{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}$$

   Create variables for the three resistors and store values in each, and then calculate the combined resistance.
8. Create a variable *ftemp* to store a temperature in degrees Fahrenheit (F). Convert this to degrees Celsius and store the result in a variable *ctemp*. The conversion factor is C = (F – 32) * 5/9.
9. The function **sin** calculates and returns the sine of an angle in radians. Use **help elfun** to find the name of the function that returns the sine of an angle in degrees. Verify that calling this function and passing 90 degrees to it results in 1.
10. A vector can be represented by its rectangular coordinates x and y or by its polar coordinates r and θ. The relationship between them is given by the equations:

```
x = r * cos(θ)
y = r * sin(θ)
```

    Assign values for the polar coordinates to variables *r* and *theta*. Then, using these values, assign the corresponding rectangular coordinates to variables *x* and *y*.
11. Wind often makes the air feel even colder than it is. The Wind Chill Factor (WCF) measures how cold it feels with a given air temperature T (in degrees Fahrenheit) and wind speed (V, in miles per hour). One formula for the WCF is:

```
WCF=35.7+0.6T-35.7(V0.16)+0.43T(V0.16)
```

    Create variables for the temperature T and wind speed V, and then using this formula calculate the WCF.
12. Use **help elfun** or experiment to answer the following questions:

    - Is **fix(3.5)** the same as **floor(3.5)**?
    - Is **fix(3.4)** the same as **fix(-3.4)**?
    - Is **fix(3.2)** the same as **floor(3.2)**?
    - Is **fix(-3.2)** the same as **floor(-3.2)**?
    - Is **fix(-3.2)** the same as **ceil(-3.2)**?

13. Find MATLAB expressions for the following:
    $\sqrt{19}$
    $3^{1.2}$
    `tan(π)`
14. Use **intmin** and **intmax** to determine the range of values that can be stored in the types **int32** and **int64**.
15. Are there equivalents to **intmin** and **intmax** for real number types? Use **help** to find out.
16. Store a number with a decimal place in a **double** variable (the default). Convert the value of that variable to the type **int32** and store the result in a new variable.
17. Generate a random:
    - Real number in the range from 0 to 1
    - Real number in the range from 0 to 20
    - Real number in the range from 20 to 50
    - Integer in the range from 0 to 10
    - Integer in the range from 0 to 11
    - Integer in the range from 50 to 100
18. Get into a new Command Window, and type **rand** to get a random real number. Make a note of the number. Then, exit MATLAB and repeat this, again making note of the random number; it should be the same as before. Finally, exit MATLAB and again get into a new Command Window. This time, change the seed before generating a random number; it should be different.
19. In the ASCII character encoding, the letters of the alphabet are in order: 'a' comes before 'b' and also 'A' comes before 'B', for example. However, which comes first: lower- or uppercase letters?
20. Shift the string 'xyz' up in the character encoding by two characters.
21. Using the colon operator, create the following vectors

```
3          4          5          6
1.0000     1.5000     2.0000     2.5000     3.0000
5          4          3          2
```

22. Using the **linspace** function, create the following vectors:

```
 4          6          8
-3         -6         -9        -12        -15
 9          7          5
```

23. Create the following vectors twice, using **linspace** and using the colon operator:

```
1     2     3     4     5     6     7     8     9     10
2     7     12
```

24. Create a variable, *myend*, which stores a random integer in the range from 8 to 12. Using the colon operator, create a vector that iterates from 1 to *myend* in steps of 3.

25. Using the colon operator and the transpose operator, create a column vector that has the values –1 to 1 in steps of 0.2.
26. Write an expression that refers to only the odd numbered elements in a vector, regardless of the length of the vector. Test your expression on vectors that have both an odd and even number of elements.
27. Create a vector variable, *vec*; it can have any length. Then, write assignment statements that would store the first half of the vector in one variable and the second half in another. Make sure that your assignment statements are general, and work whether *vec* has an even or odd number of elements (hint: use a rounding function such as **fix**).
28. Using colon operators for the rows, create the matrix:

```
7     6     5
3     5     7
```

29. Generate a 2 × 3 matrix of random

- Real numbers, each in the range from 0 to 1
- Real numbers, each in the range from 0 to 10
- Integers, each in the range from 5 to 20

30. Create a variable, *rows*, which is a random integer in the range from 1 to 5. Create a variable, *cols*, which is a random integer in the range from 1 to 5. Create a matrix of all zeros with the dimensions given by the values of *rows* and *cols*.
31. Find an efficient way to generate the following matrix:
mat =

```
 7     8     9    10
12    10     8     6
```

- Then, give expressions that will, for the matrix *mat*,
  - Refer to the element in the first row, third column
  - Refer to the entire second row
  - Refer to the first two columns

32. Create a matrix variable, *mymat*, which stores the following:
mymat =

```
2     5     8
7     5     3
```

Using this matrix, find a simple expression that will transform the matrix into each of the following:

```
2     7
5     5
8     3
```

```
2     5
7     8
5     3

8     5     2
3     5     7

8     3
5     5
2     7

2     5     8     2     5     8
7     5     3     7     5     3
```

33. Create a 4 × 2 matrix of all zeros and store it in a variable. Then, replace the second row in the matrix with a 3 and a 6.
34. Create a vector, x, which consists of 20 equally spaced points in the range from $-\pi$ to $+\pi$. Create a y vector that is **sin(x)**.
35. Create a 3 × 5 matrix of random integers, each in the range from –5 to 5. Get the **sign** of every element.
36. Create a 4 × 6 matrix of random integers, each in the range from –5 to 5; store it in a variable. Create another matrix that stores for each element the absolute value of the corresponding element in the original matrix.
37. Create a 3 × 5 matrix of random real numbers. Delete the third row.
38. The built-in function **clock** returns a vector that contains six elements: the first three are the current date (year, month, day) and the last three represent the current time in hours, minutes, and seconds. The seconds is a real number, but all others are integers. Store the result from **clock** in a variable called *myc*. Then, store the first three elements from this variable in a variable called *today*, and the last three elements in a variable called *now*. Use the **fix** function on the vector variable called *now* to get just the integer part of the current time.