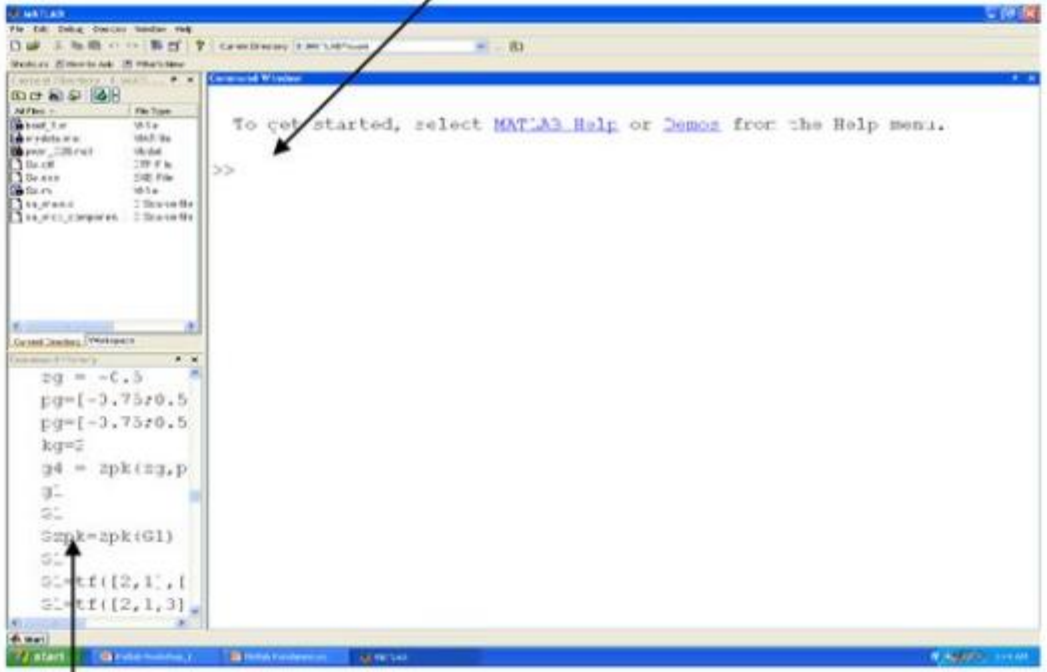
Introduction to MATLAB Programming

Outline

- MATLAB Environment
  - MATLAB layout
  - Data types
- MATLAB as a Calculator
- Variables
  - Built-in variables
  - Logical variables
  - String
- Matrices
  - Creating
  - Operation
  - Indexing
  - Matrices operation
- Build-in function
- Script file

MATLAB Desktop

On starting MATLAB:
* Matlab Command Prompt



```
To get started, select MATLAB Help or Demos from the Help menu.

>>
```

```
zg = -0.5
pg=[-3.75;0.5
pg=[-3.75;0.5
kg=2
g4 = zpk(zg,p
g1
G1
Gzpk=zpk(G1)
G1
G1=tf([2,1],[
G1=tf([2,1,3]
```

The Command history maintains a record
of Matlab functions you ran

Matlab Desktop

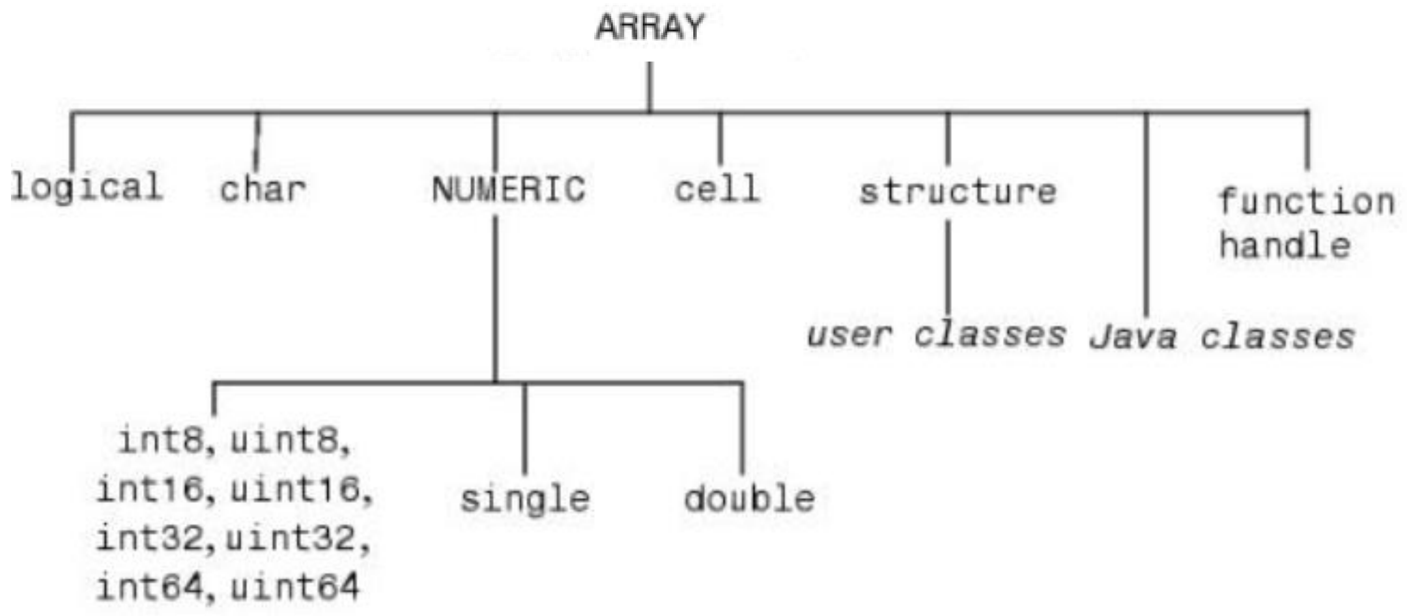
- The desktop provides different ways of interacting with Matlab
- ⊲ Entering commands in the command window
- ⊲ Viewing values stored in variables
- ⊲ Editing statements in Matlab functions and scripts
- ⊲ Creating and annotating plots

Data types supported by MATLAB

- Name Description
- -------- ----------------
- double - double precision, floating point numbers (8 bytes per element)
- uint8 -unsigned 8-bit integers in the range [0-255] (1 byte per element)
- uint16 - unsigned 16-bit integers in the range [0,65535]
- uint32 - unsigned 32-bit integers in the range [0,4294967295]
- int8 - singed 8-bit integers in the range [-128,127]
- int16 - singed 16-bit integers in the range [-32768,32767]
- int32 - singed 32-bit integers in the range [-2147483648,2147483647]
- single- single precision floating-point numbers (4 bytes per element)
- char - characters (2 bytes per element using Unicode rep.)
- logical - values are 0 or 1 (1 byte per element)
- All numeric computations (first 8 entries in the above table) in MATLAB are done using double quantities.
- uint8 is the frequently used 8-bit image.

Data Types

- There are 14 fundamental data types in MATLAB



Default Numeric Data Type – double

Variables in MATLAB

- MATLAB stores all variables as matrices/arrays
  - A single value is just a 1 x 1 matrix
- You can store numerical values, characters, and strings into variables
- Meaningful variable names will help you make a good program (detailed naming rules are covered later)
  - Variable names are case sensitive – be careful
  - Variable, VARIABLE, variable, and vaRIAble are all different
- The data stored in a variable is stored in the memory assigned for MATLAB in your computer and when you exit MATLAB the data is erased
- Variables created are shown in the workspace window
- **clear all** will erase all variables (and other stuff) from the workspace

General Purpose Commands

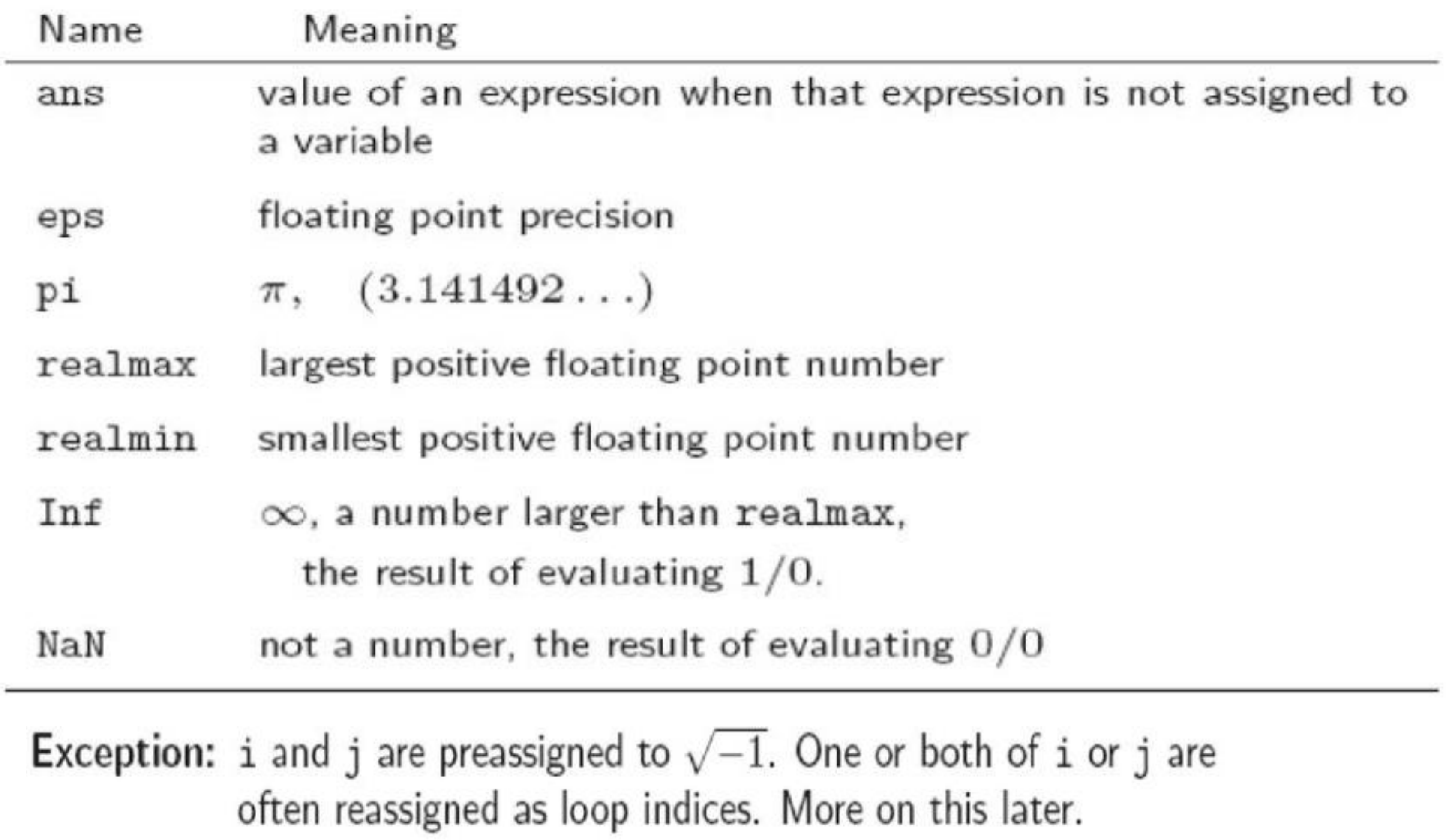
who	lists all variables in the current workspace
whos	lists all variables in the workspace including array sizes
clear	clears all variables and functions from memory
clc	to clear the Command Window

Variables in MATLAB

- You can store numerical values, characters, and strings into variables
- MATLAB stores all variables as matrices/arrays
  - A single value is just a 1 x 1 matrix
- Meaningful variable names will help you make a good program

  (detailed naming rules are covered later)
  - Variable names are case sensitive – be careful
  - Variable, VARIABLE, variable, and vaRIAble are all different
- The data stored in a variable is stored in the memory assigned for MATLAB in your computer and when you exit MATLAB the data is erased
- Variables created are shown in the workspace window
- **clear all** will erase all variables (and other stuff) from the

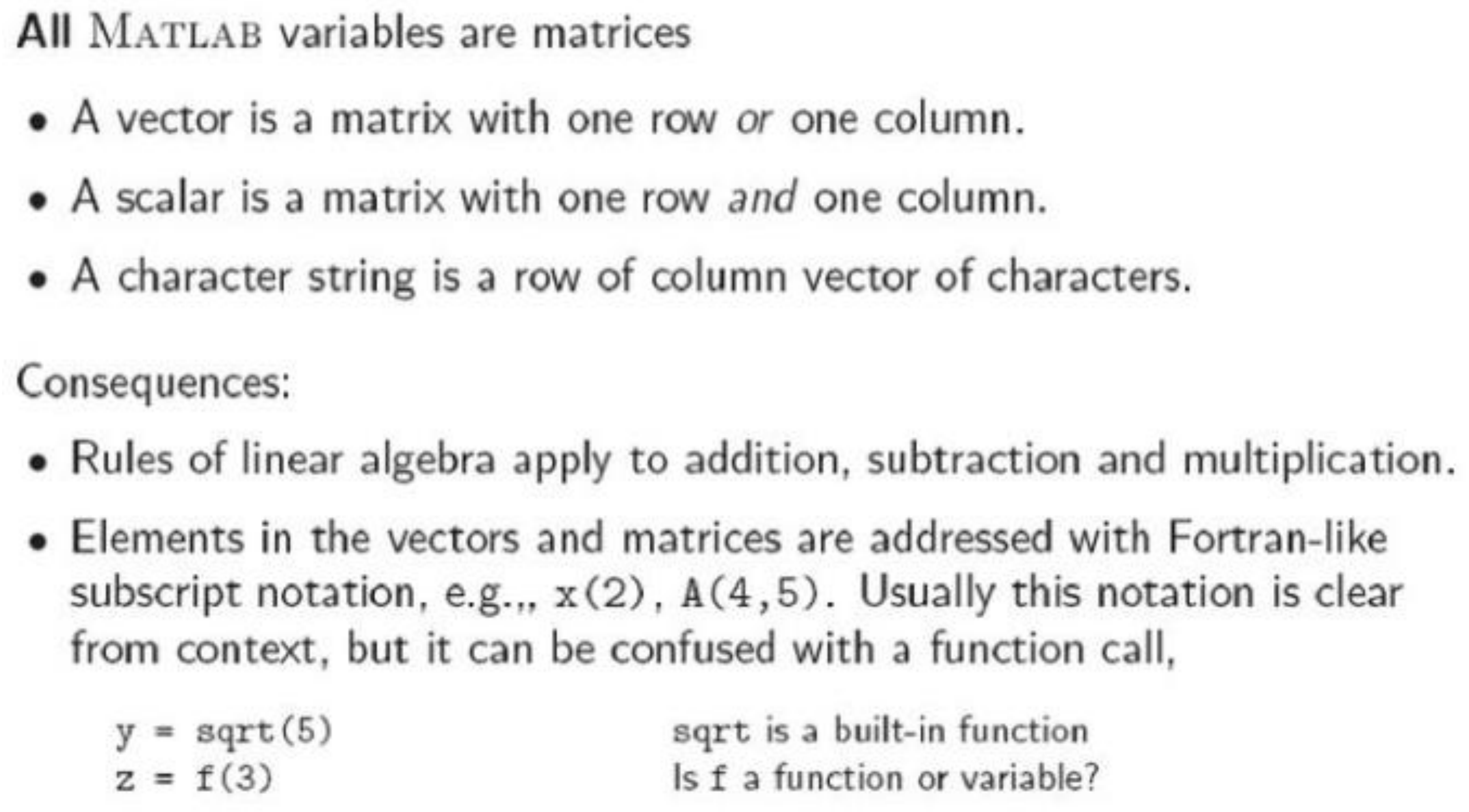
Built-in MATLAB Variables



Name	Meaning
`ans`	value of an expression when that expression is not assigned to a variable
`eps`	floating point precision
`pi`	π, $(3.141492\ldots)$
`realmax`	largest positive floating point number
`realmin`	smallest positive floating point number
`Inf`	∞, a number larger than `realmax`, the result of evaluating $1/0$.
`NaN`	not a number, the result of evaluating $0/0$

Exception: `i` and `j` are preassigned to $\sqrt{-1}$. One or both of `i` or `j` are often reassigned as loop indices. More on this later.

Matrices and Vectors



All MATLAB variables are matrices

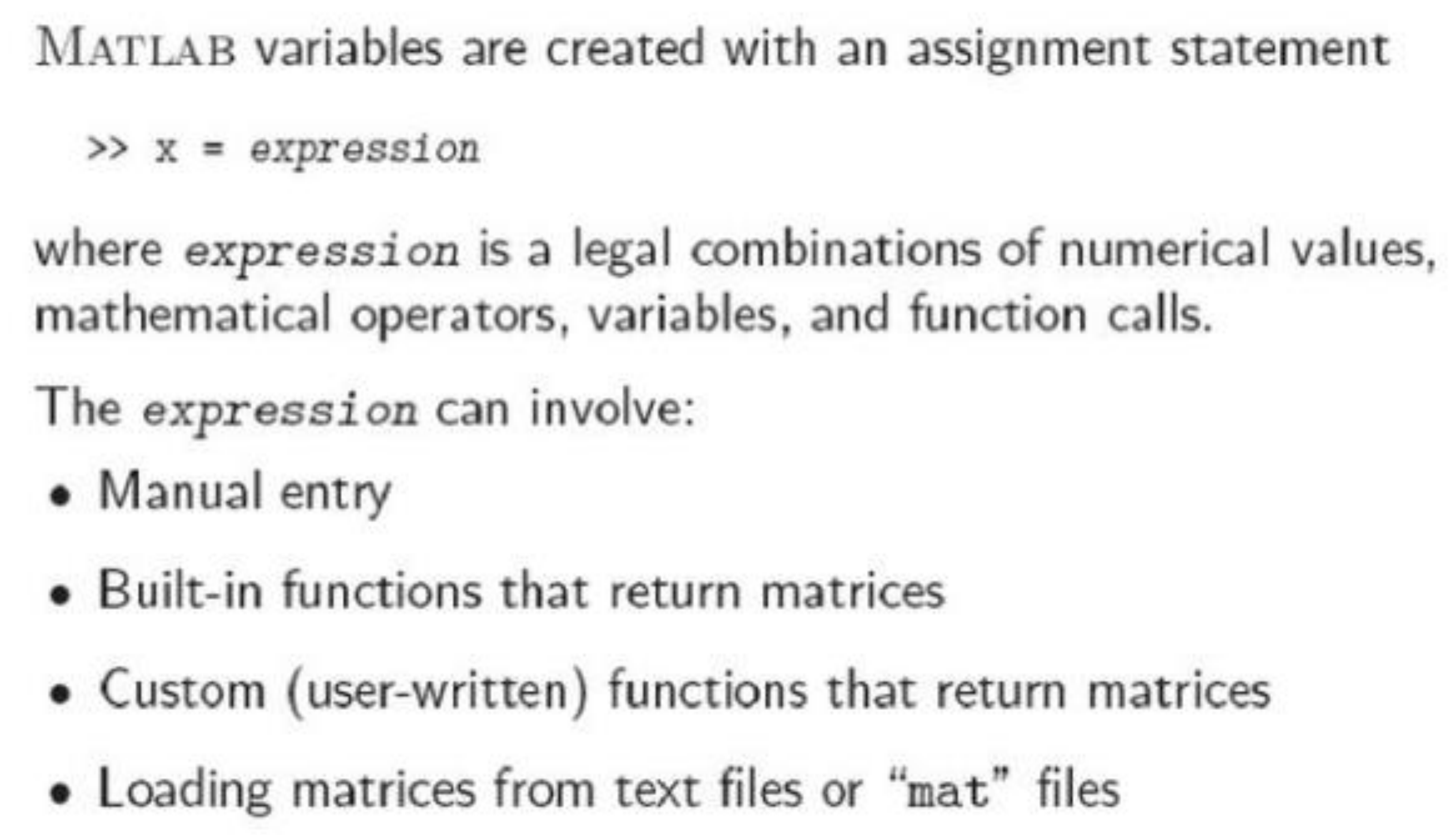
- A vector is a matrix with one row *or* one column.
- A scalar is a matrix with one row *and* one column.
- A character string is a row of column vector of characters.

Consequences:

- Rules of linear algebra apply to addition, subtraction and multiplication.
- Elements in the vectors and matrices are addressed with Fortran-like subscript notation, e.g.,, `x(2)`, `A(4,5)`. Usually this notation is clear from context, but it can be confused with a function call,

```
y = sqrt(5)                    sqrt is a built-in function
z = f(3)                       Is f a function or variable?
```

Creating MATLAB Variables



MATLAB variables are created with an assignment statement

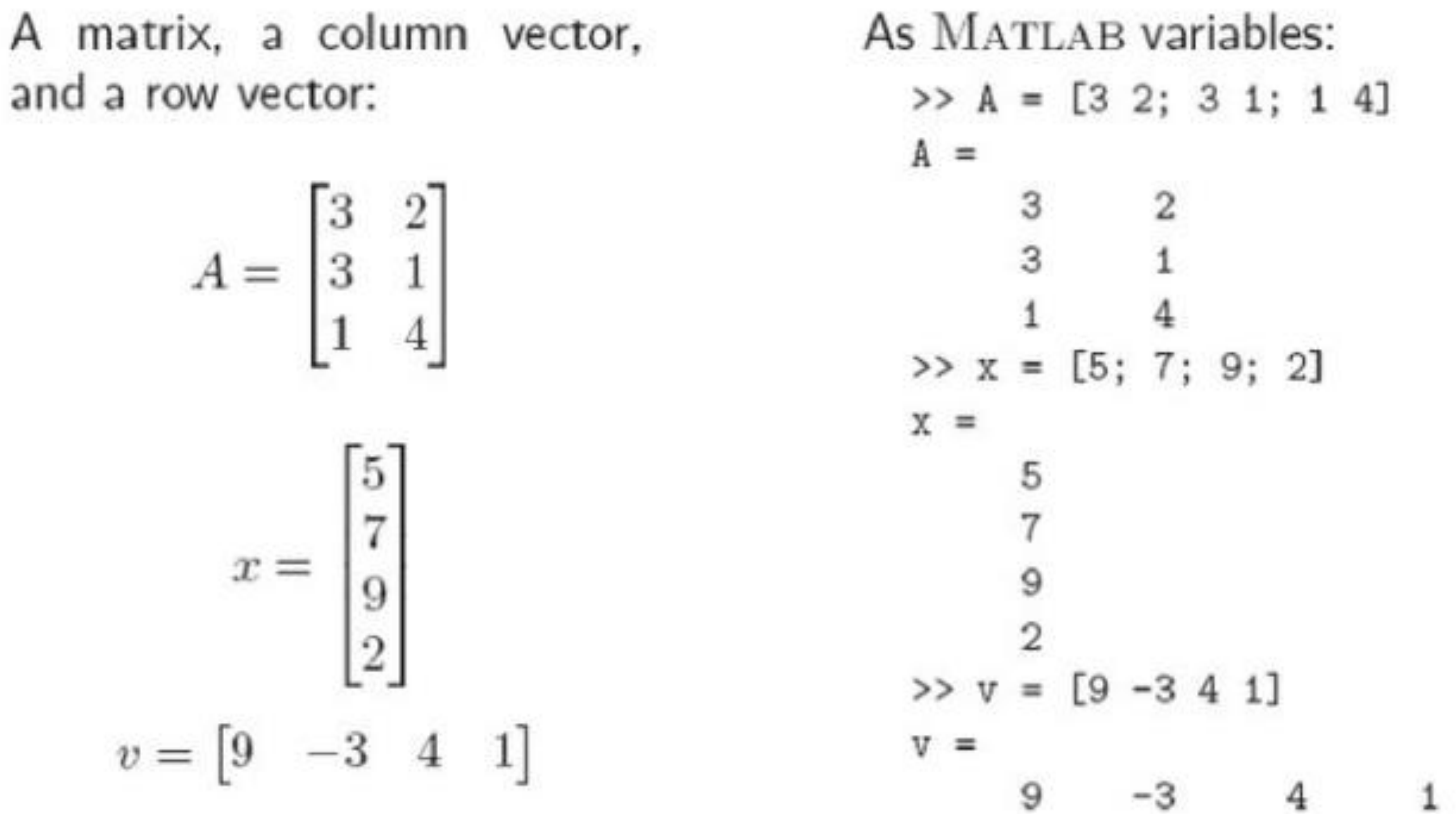
```
>> x = expression
```

where *expression* is a legal combinations of numerical values, mathematical operators, variables, and function calls.

The *expression* can involve:

- Manual entry
- Built-in functions that return matrices
- Custom (user-written) functions that return matrices
- Loading matrices from text files or "`mat`" files

Element-by-Element Creation of Matrices and Vectors



A matrix, a column vector, and a row vector:

$$A = \begin{bmatrix} 3 & 2 \\ 3 & 1 \\ 1 & 4 \end{bmatrix}$$

$$x = \begin{bmatrix} 5 \\ 7 \\ 9 \\ 2 \end{bmatrix}$$

$$v = \begin{bmatrix} 9 & -3 & 4 & 1 \end{bmatrix}$$

As MATLAB variables:

```
>> A = [3 2; 3 1; 1 4]
A =
     3     2
     3     1
     1     4
>> x = [5; 7; 9; 2]
x =
     5
     7
     9
     2
>> v = [9 -3 4 1]
v =
     9    -3     4     1
```

Element-by-Element Creation of Matrices and Vectors

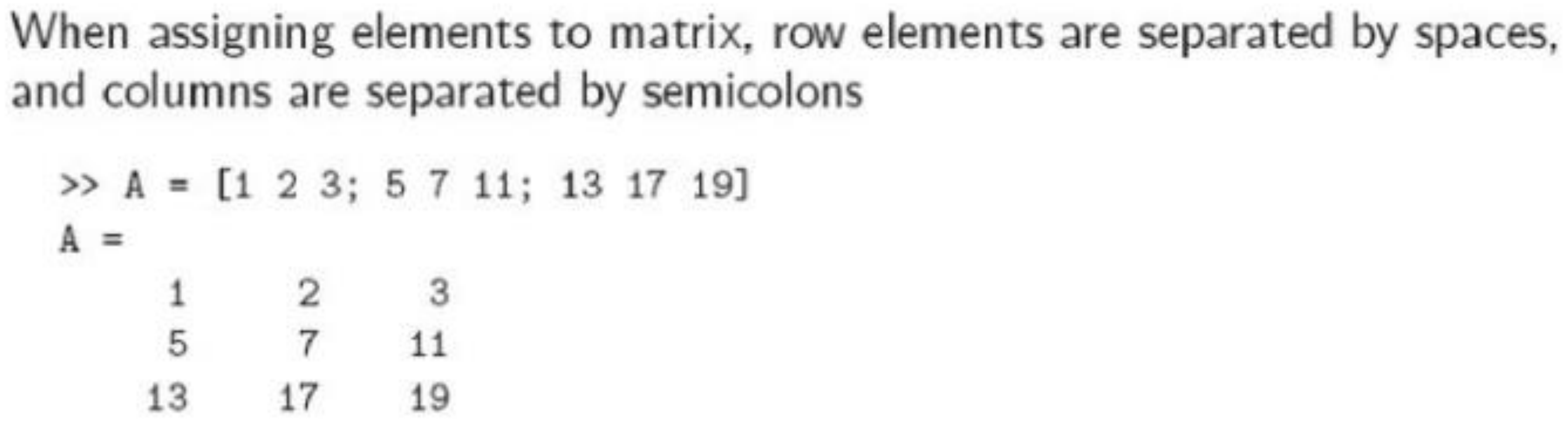
For manual entry, the elements in a vector are enclosed in square brackets. When creating a row vector, separate elements with a space.

```
>> v = [7 3 9]
v =
     7   3  9
```

Separate columns with a semicolon

```
>> w = [2; 6; 1]
w =
     2
     6
     1
```

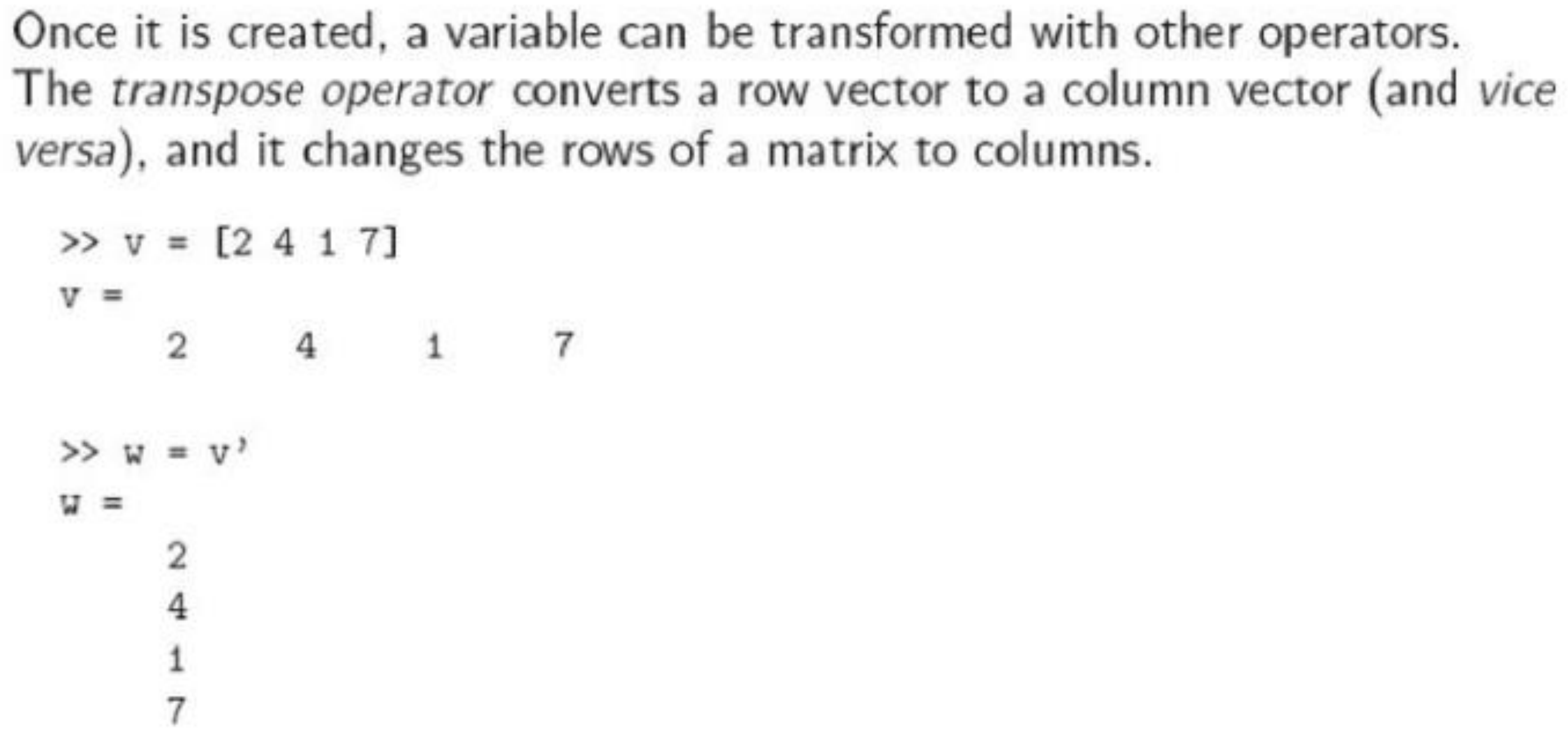
Element-by-Element Creation of Matrices and Vectors



When assigning elements to matrix, row elements are separated by spaces, and columns are separated by semicolons

```
>> A = [1 2 3; 5 7 11; 13 17 19]
A =
      1      2      3
      5      7     11
     13     17     19
```

Transpose Operator

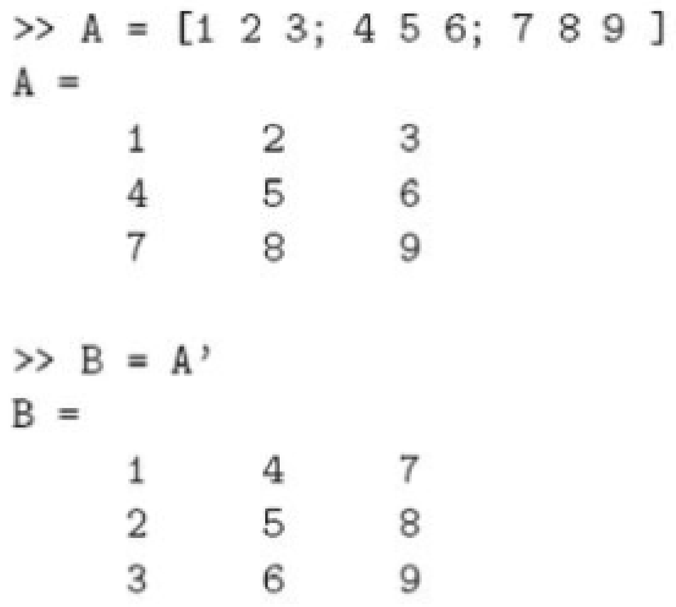


Once it is created, a variable can be transformed with other operators. The *transpose operator* converts a row vector to a column vector (and *vice versa*), and it changes the rows of a matrix to columns.

```
>> v = [2 4 1 7]
v =
     2     4     1     7

>> w = v'
w =
     2
     4
     1
     7
```

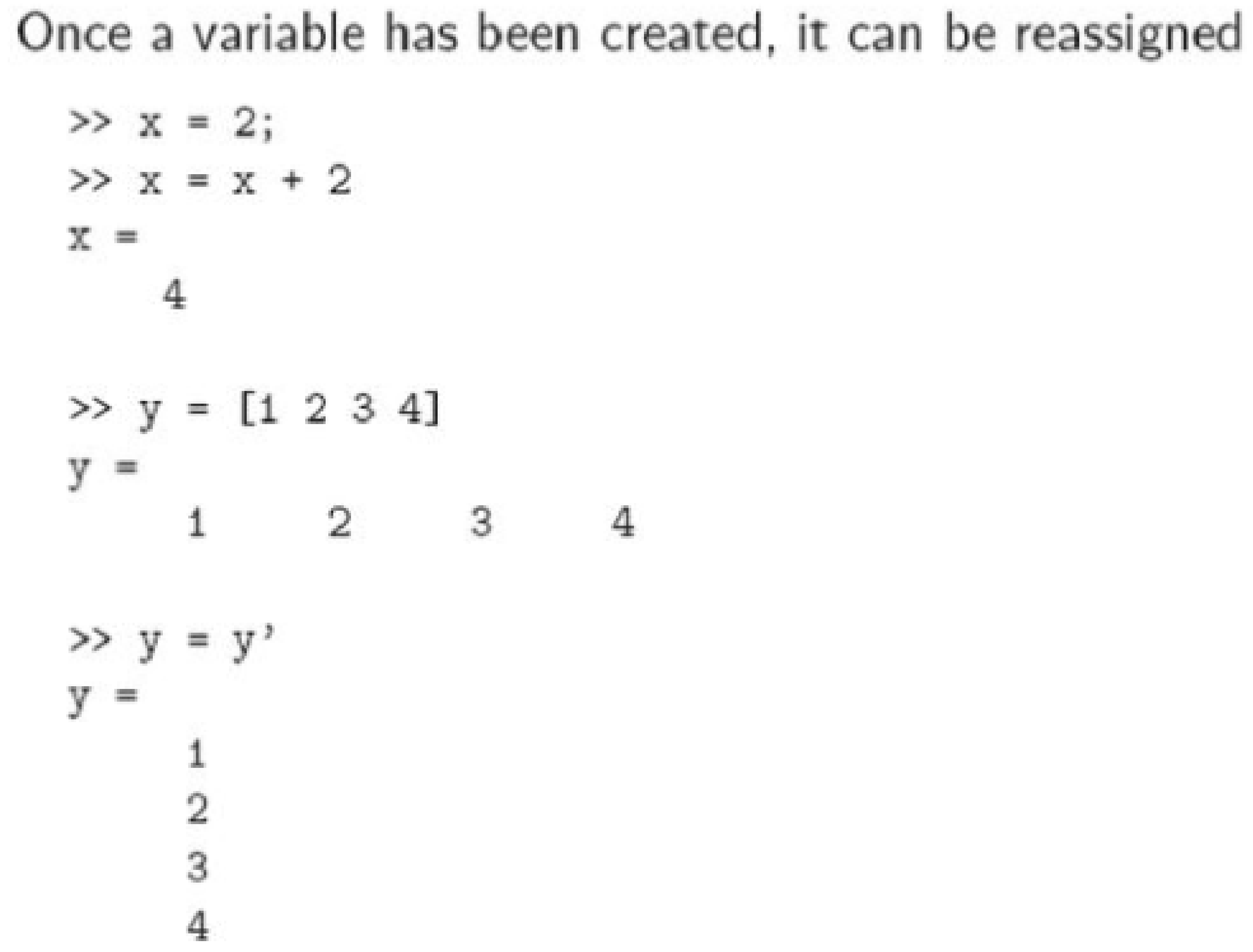
Transpose Operator(2)



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

>> B = A'
B =
     1     4     7
     2     5     8
     3     6     9
```

Overwriting Variables



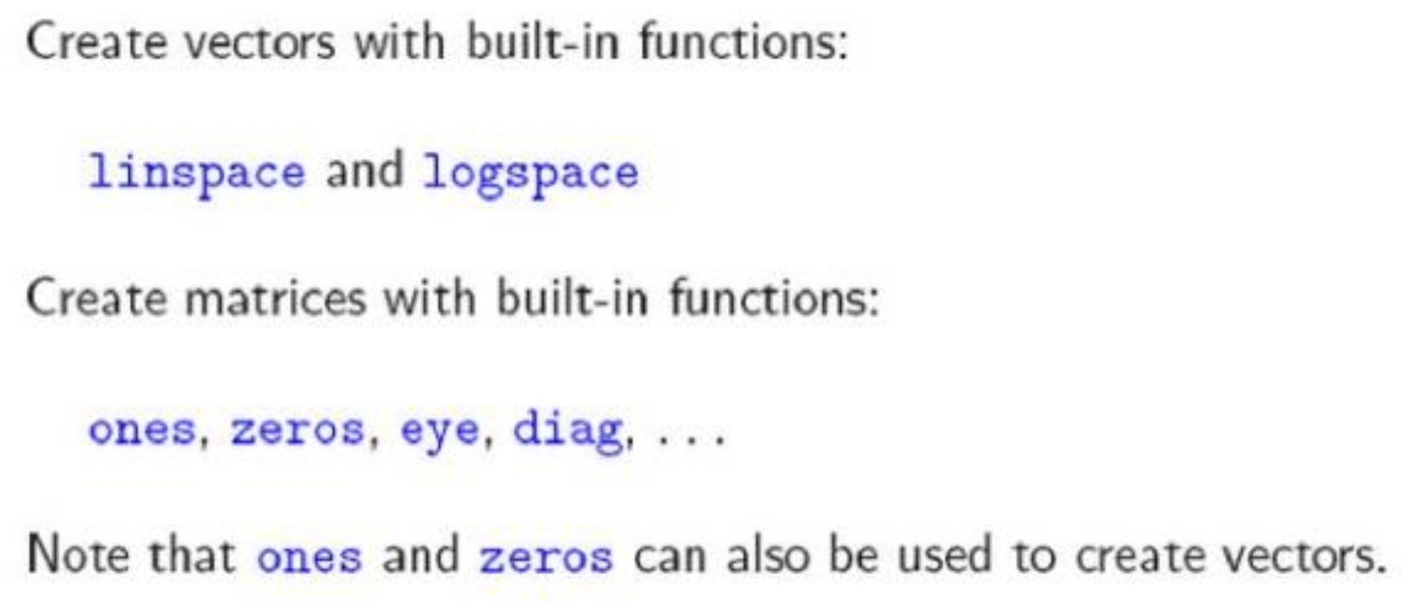
Once a variable has been created, it can be reassigned

```
>> x = 2;
>> x = x + 2
x =
    4

>> y = [1 2 3 4]
y =
     1     2     3     4

>> y = y'
y =
     1
     2
     3
     4
```

Using Functions to Create Matrices and Vectors



Create vectors with built-in functions:

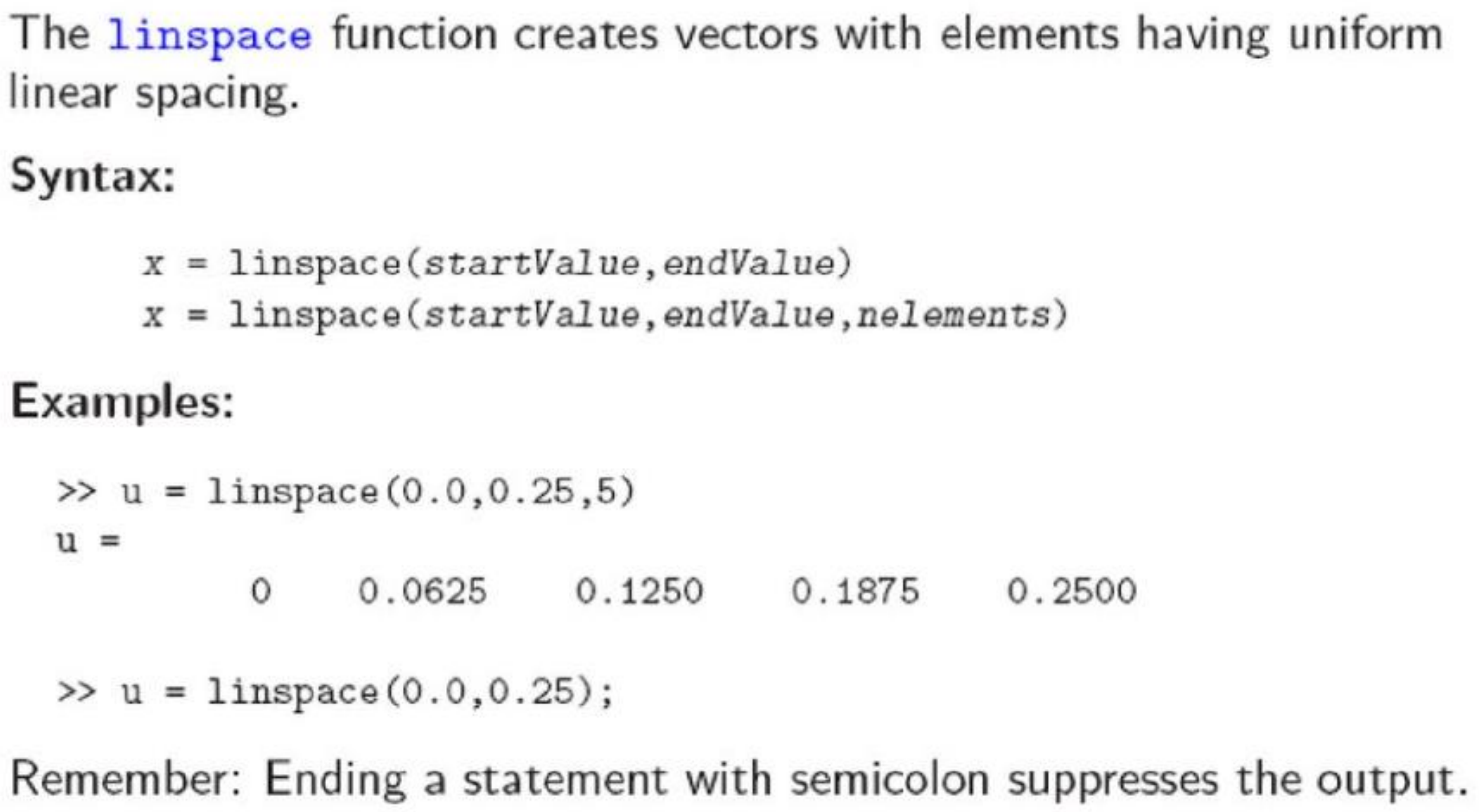
`linspace` and `logspace`

Create matrices with built-in functions:

`ones`, `zeros`, `eye`, `diag`, . . .

Note that `ones` and `zeros` can also be used to create vectors.

Creating vectors with linspace (1)



The `linspace` function creates vectors with elements having uniform linear spacing.

Syntax:

```
x = linspace(startValue,endValue)
x = linspace(startValue,endValue,nelements)
```

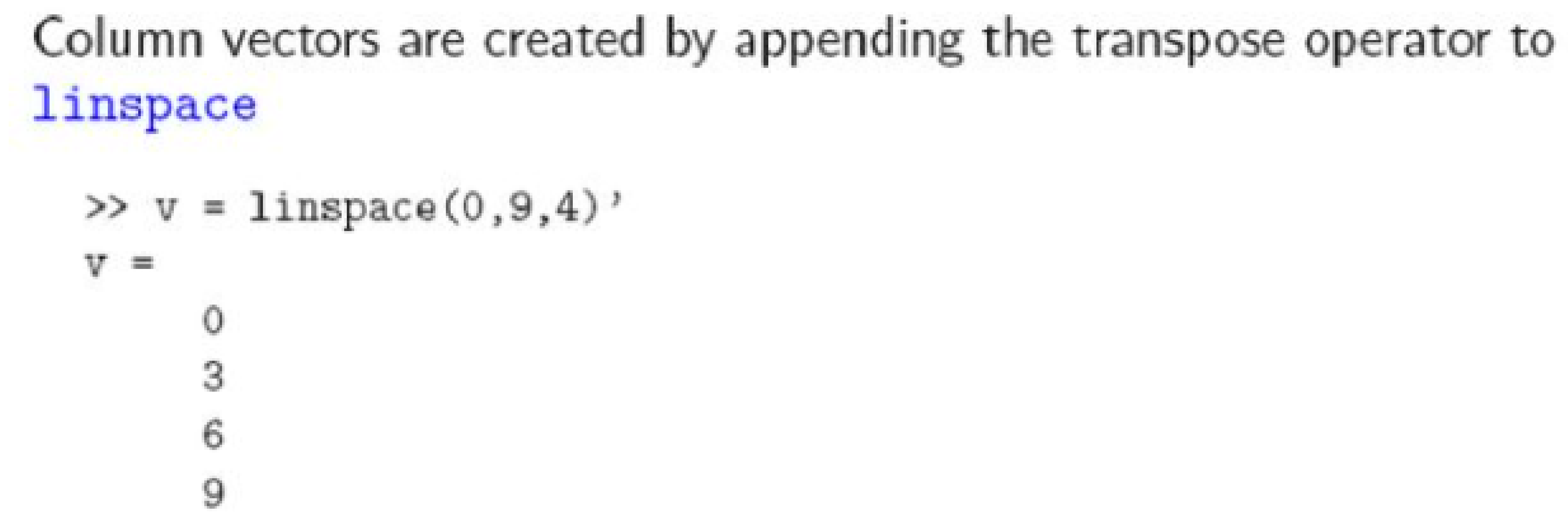
Examples:

```
>> u = linspace(0.0,0.25,5)
u =
         0    0.0625    0.1250    0.1875    0.2500

>> u = linspace(0.0,0.25);
```

Remember: Ending a statement with semicolon suppresses the output.

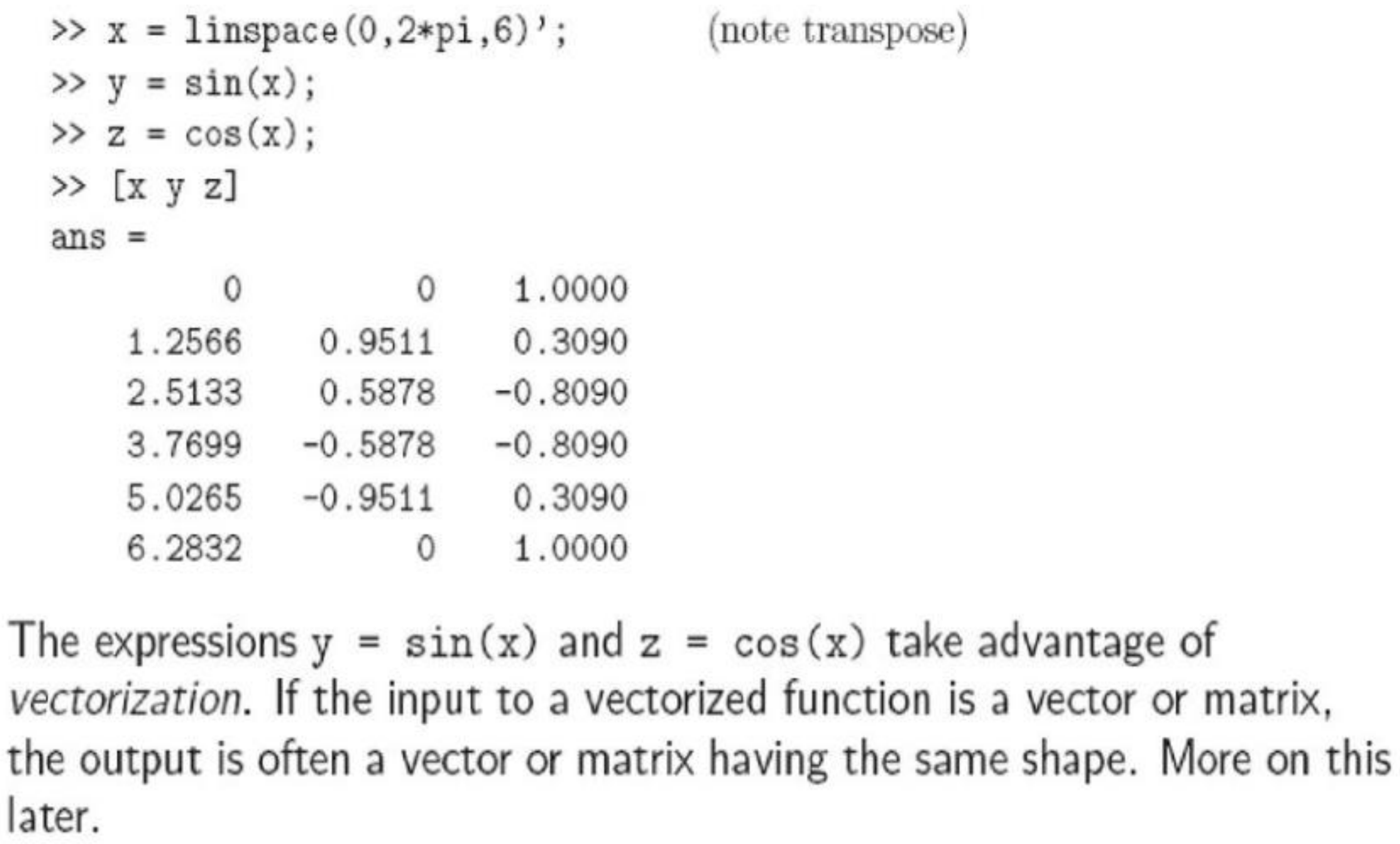
Creating vectors with linspace (2)



Column vectors are created by appending the transpose operator to `linspace`

```
>> v = linspace(0,9,4)'
v =
     0
     3
     6
     9
```

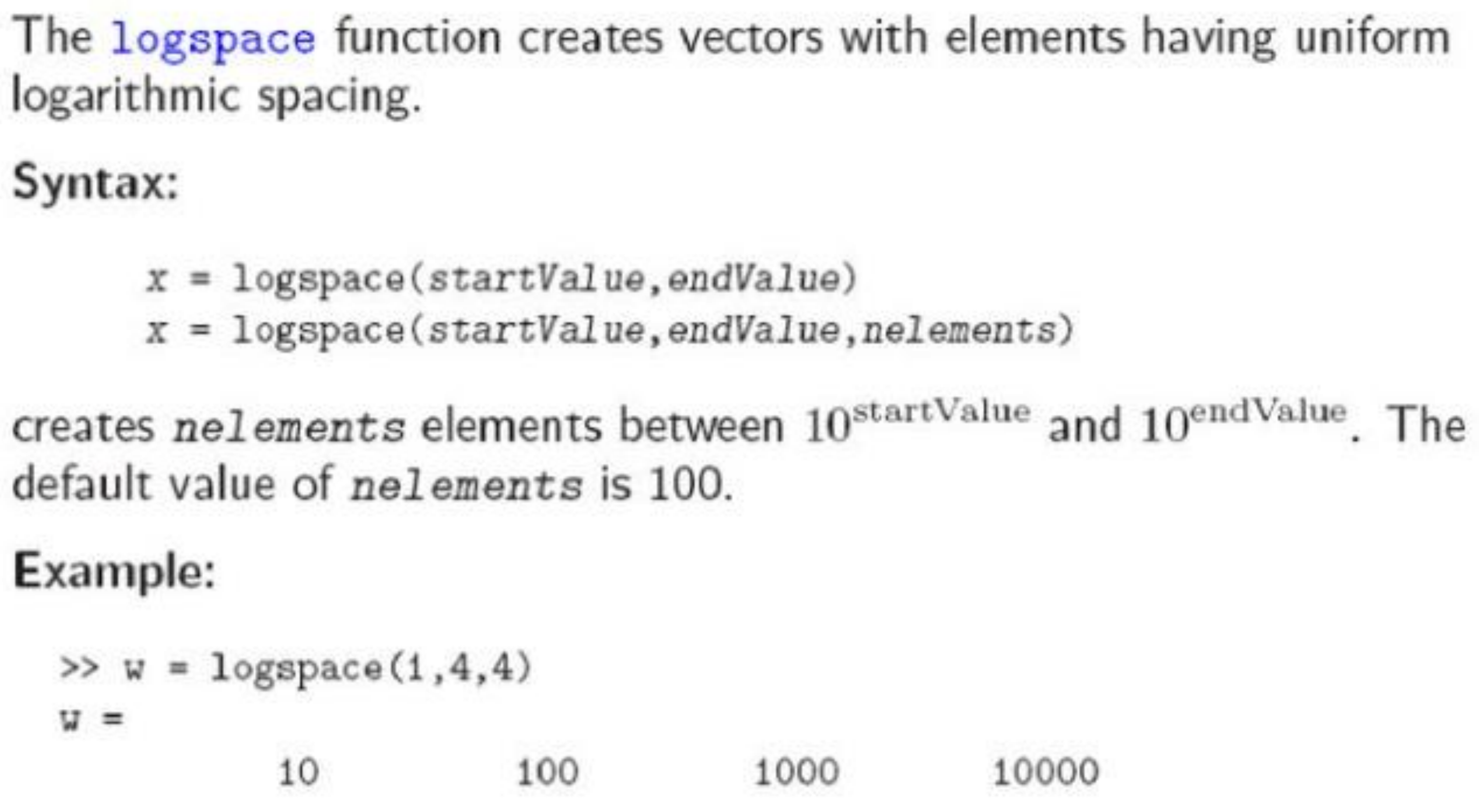
Example: A Table of Trig Functions



```
>> x = linspace(0,2*pi,6)';        (note transpose)
>> y = sin(x);
>> z = cos(x);
>> [x y z]
ans =
         0         0    1.0000
    1.2566    0.9511    0.3090
    2.5133    0.5878   -0.8090
    3.7699   -0.5878   -0.8090
    5.0265   -0.9511    0.3090
    6.2832         0    1.0000
```

The expressions `y = sin(x)` and `z = cos(x)` take advantage of *vectorization*. If the input to a vectorized function is a vector or matrix, the output is often a vector or matrix having the same shape. More on this later.

Creating vectors with logspace



The `logspace` function creates vectors with elements having uniform logarithmic spacing.

Syntax:

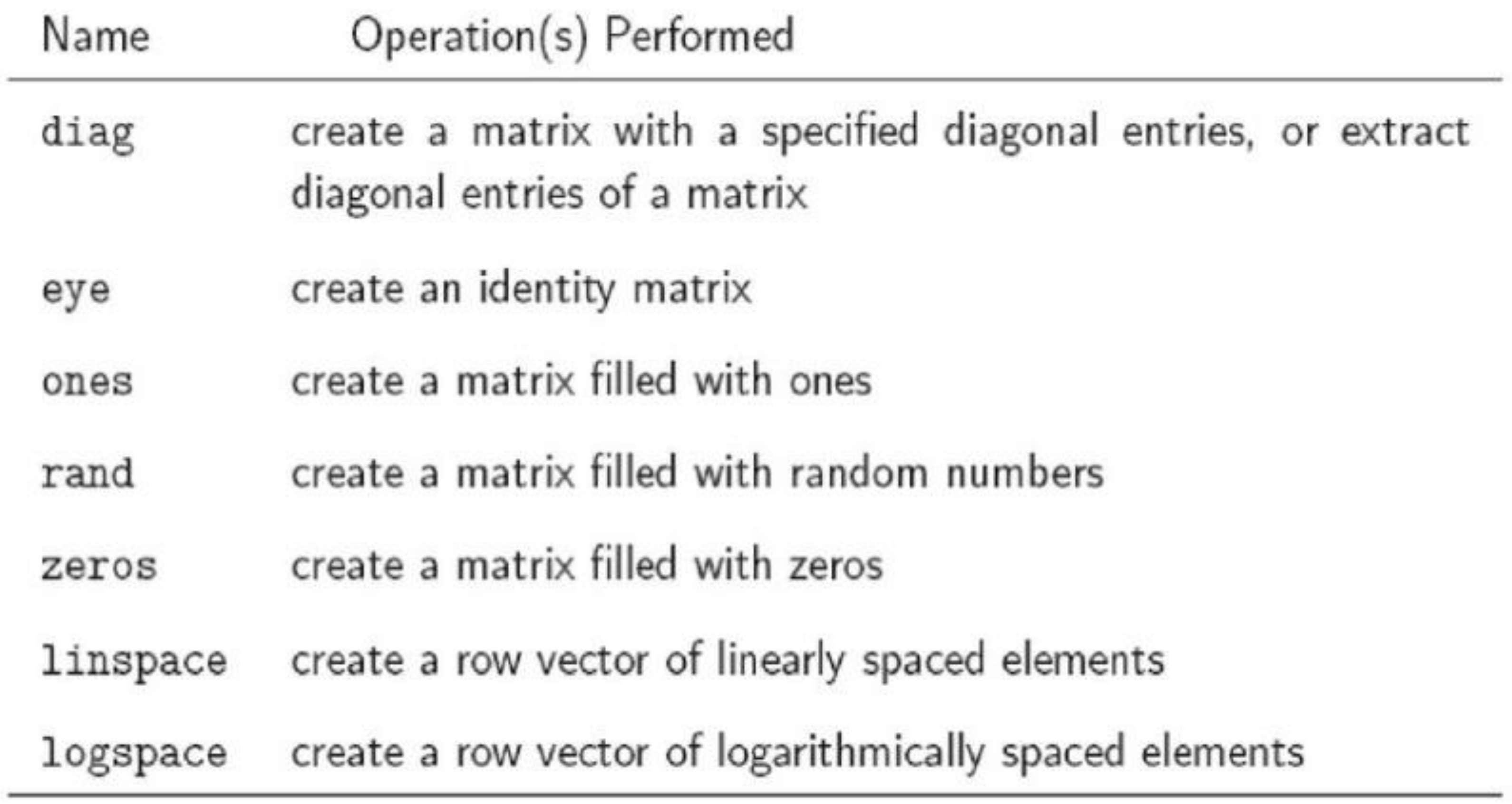
```
x = logspace(startValue,endValue)
x = logspace(startValue,endValue,nelements)
```

creates *`nelements`* elements between $10^{\text{startValue}}$ and 10^{endValue}. The default value of *`nelements`* is 100.

Example:

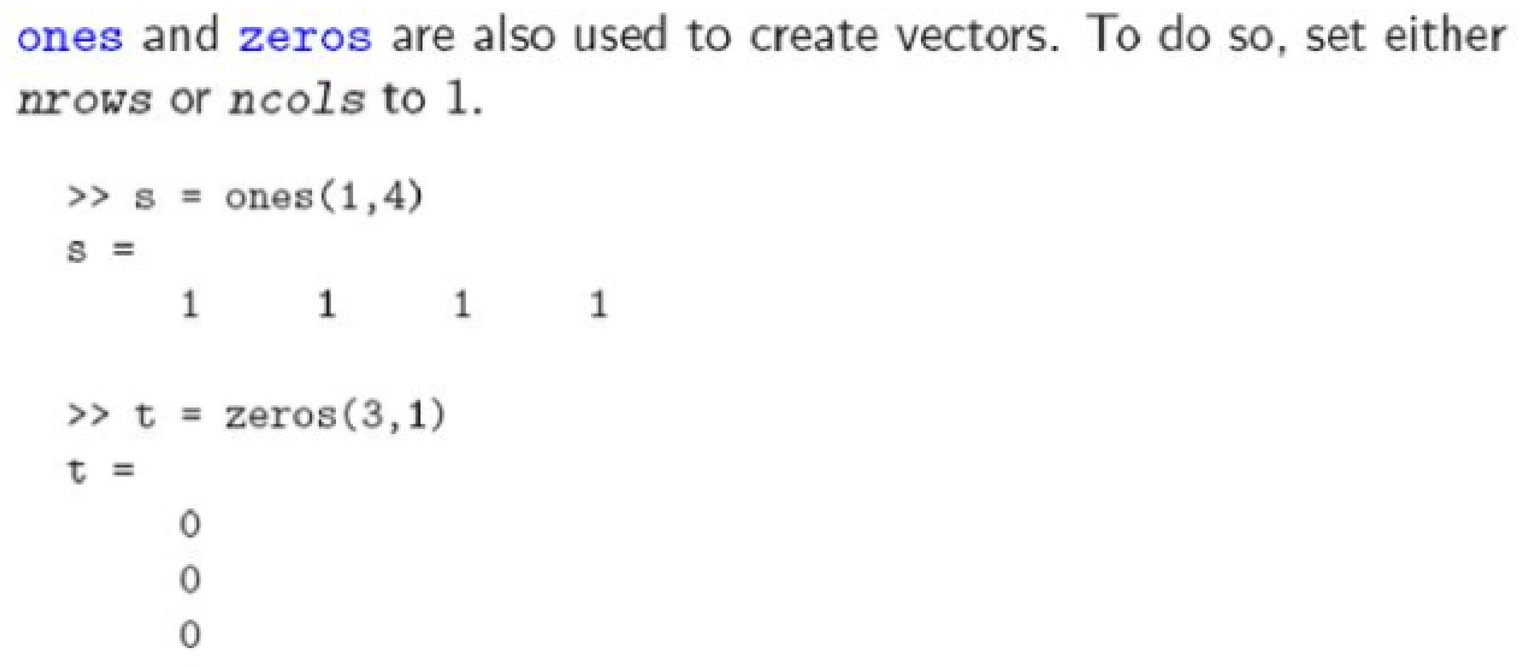
```
>> w = logspace(1,4,4)
w =
          10          100          1000          10000
```

Functions to Create Matrices (1)



Name	Operation(s) Performed
`diag`	create a matrix with a specified diagonal entries, or extract diagonal entries of a matrix
`eye`	create an identity matrix
`ones`	create a matrix filled with ones
`rand`	create a matrix filled with random numbers
`zeros`	create a matrix filled with zeros
`linspace`	create a row vector of linearly spaced elements
`logspace`	create a row vector of logarithmically spaced elements

Functions to Create Matrices (2)

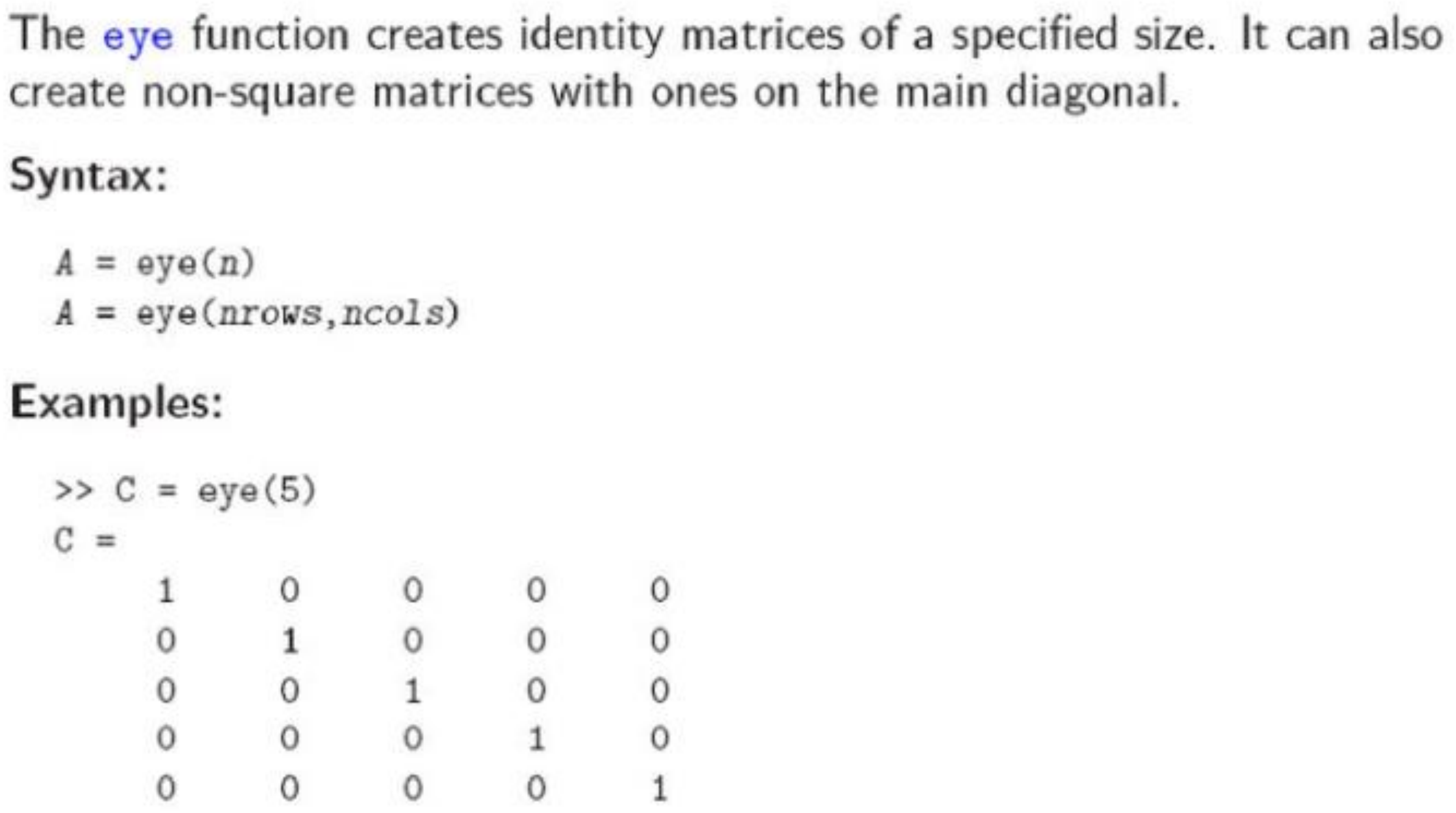


`ones` and `zeros` are also used to create vectors. To do so, set either *nrows* or *ncols* to 1.

```
>> s = ones(1,4)
s =
       1       1       1       1

>> t = zeros(3,1)
t =
       0
       0
       0
```

Functions to Create Matrices (3)



The eye function creates identity matrices of a specified size. It can also create non-square matrices with ones on the main diagonal.

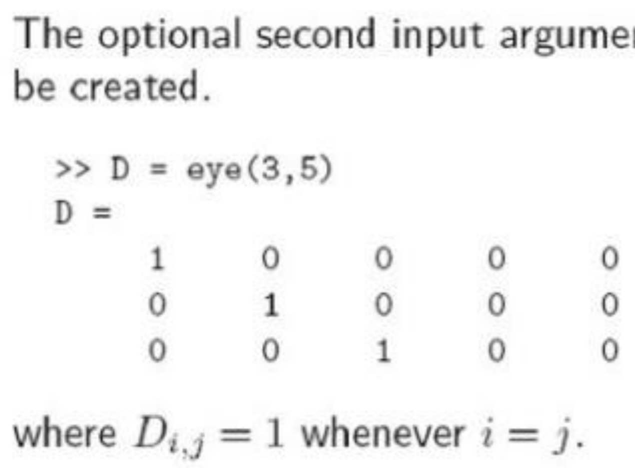
Syntax:

```
A = eye(n)
A = eye(nrows,ncols)
```

Examples:

```
>> C = eye(5)
C =
      1     0     0     0     0
      0     1     0     0     0
      0     0     1     0     0
      0     0     0     1     0
      0     0     0     0     1
```

Functions to Create Matrices (4)

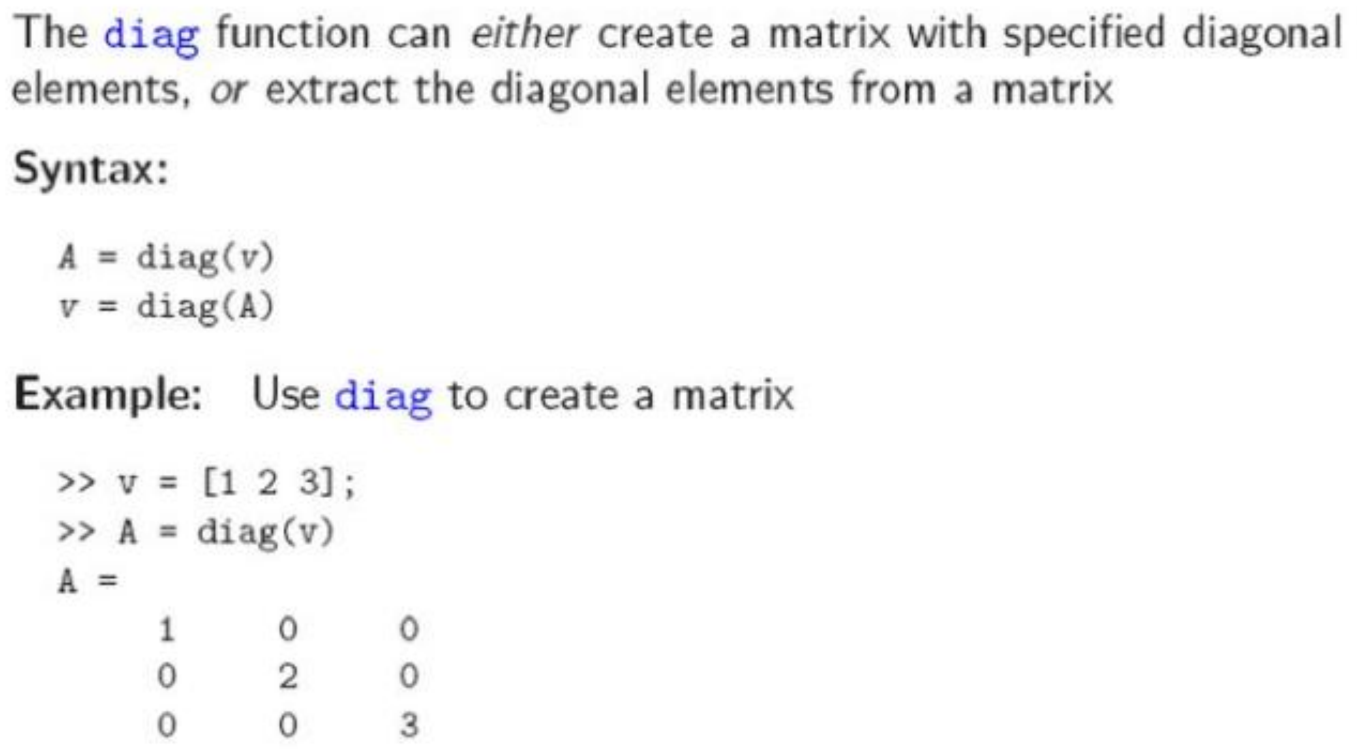


The optional second input argument to eye allows non-square matrices to be created.

```
>> D = eye(3,5)
D =
      1      0      0      0      0
      0      1      0      0      0
      0      0      1      0      0
```

where $D_{i,j} = 1$ whenever $i = j$.

Functions to Create Matrices (5)



The `diag` function can *either* create a matrix with specified diagonal elements, *or* extract the diagonal elements from a matrix

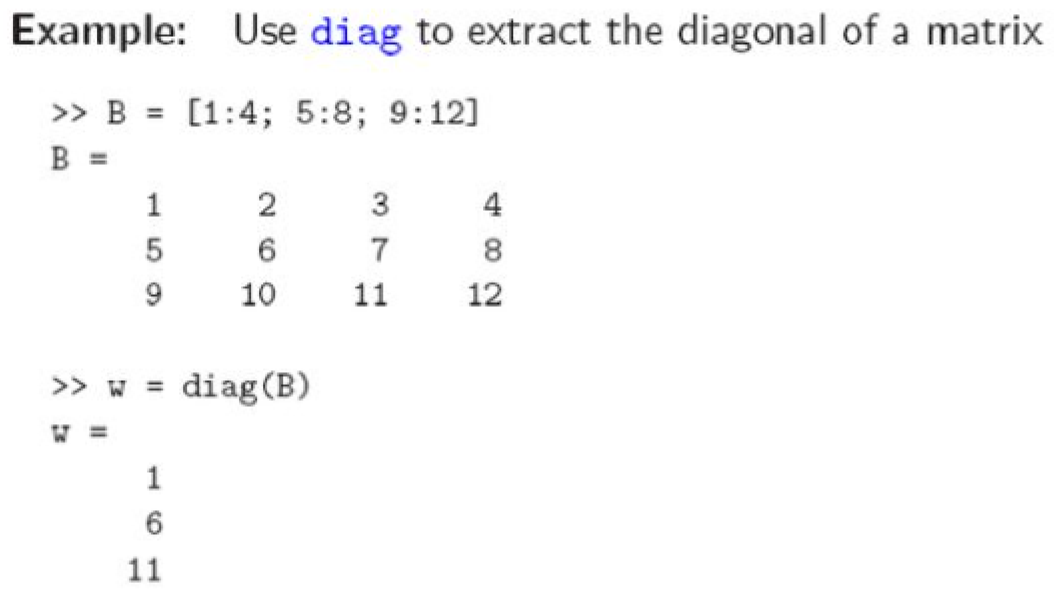
Syntax:

```
A = diag(v)
v = diag(A)
```

Example: Use `diag` to create a matrix

```
>> v = [1 2 3];
>> A = diag(v)
A =
     1     0     0
     0     2     0
     0     0     3
```

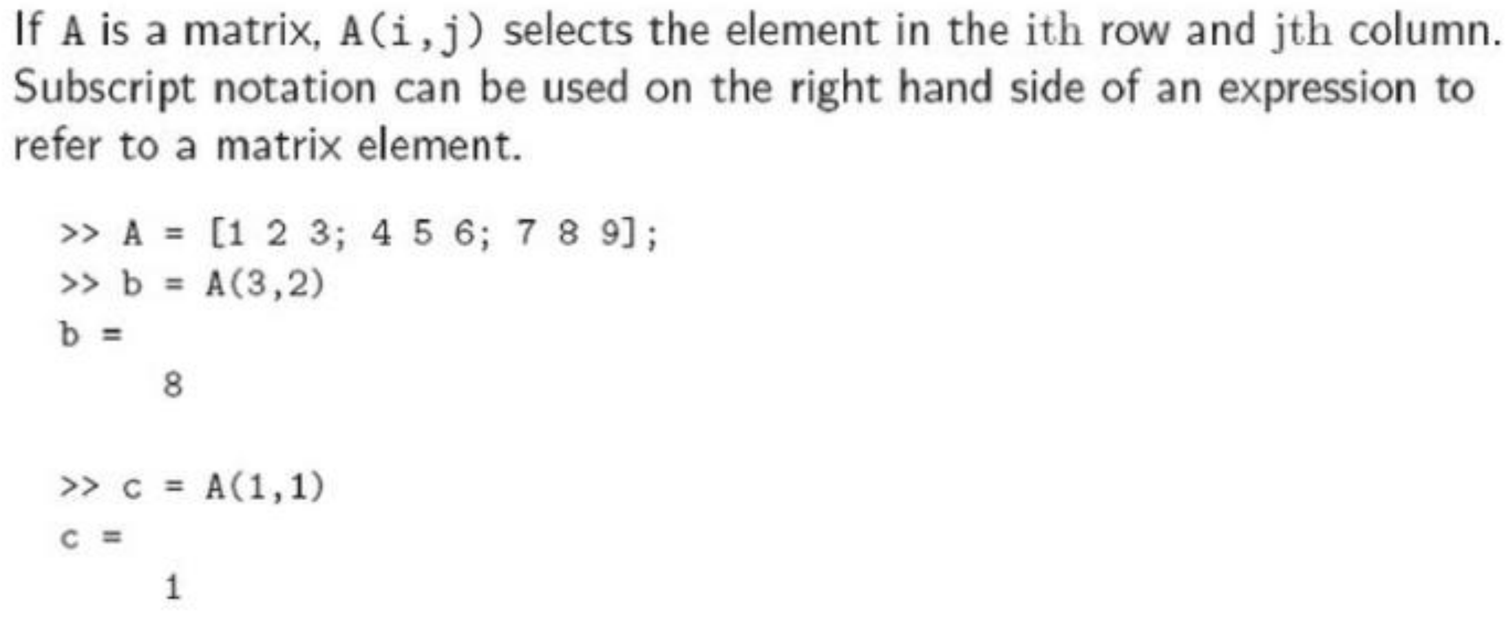
Functions to Create Matrices (6)


Example: Use `diag` to extract the diagonal of a matrix

```
>> B = [1:4; 5:8; 9:12]
B =
     1     2     3     4
     5     6     7     8
     9    10    11    12

>> w = diag(B)
w =
     1
     6
    11
```

Indexing Matrices (1)

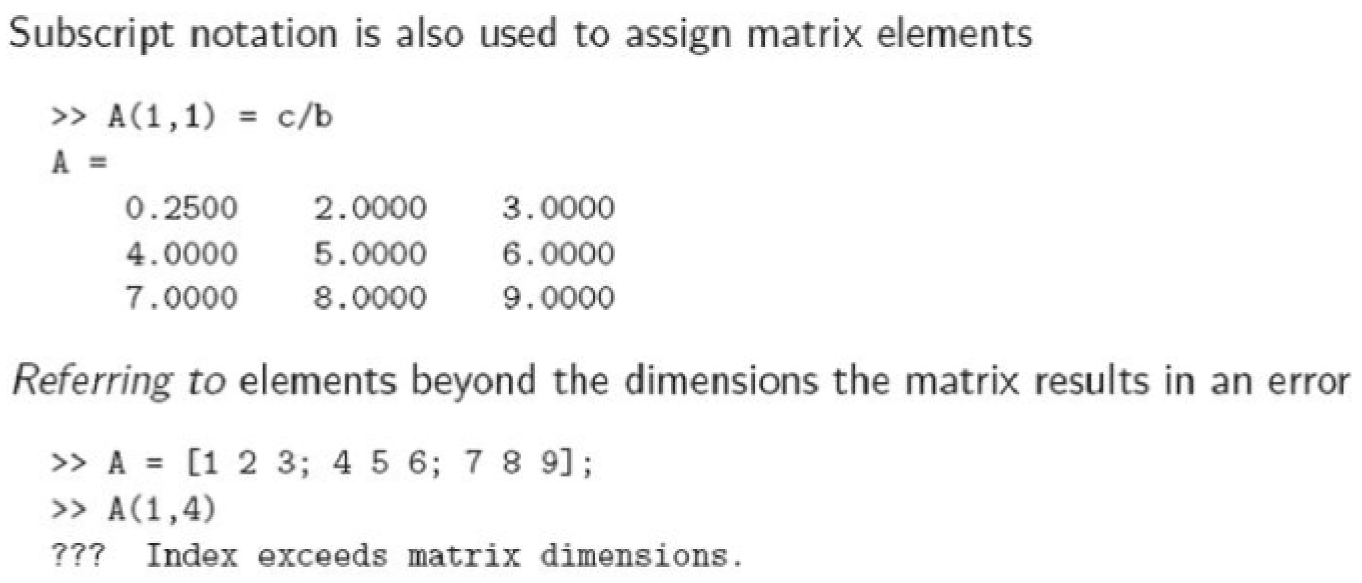


If `A` is a matrix, `A(i,j)` selects the element in the ith row and jth column. Subscript notation can be used on the right hand side of an expression to refer to a matrix element.

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> b = A(3,2)
b =
     8

>> c = A(1,1)
c =
     1
```

Indexing Matrices (2)



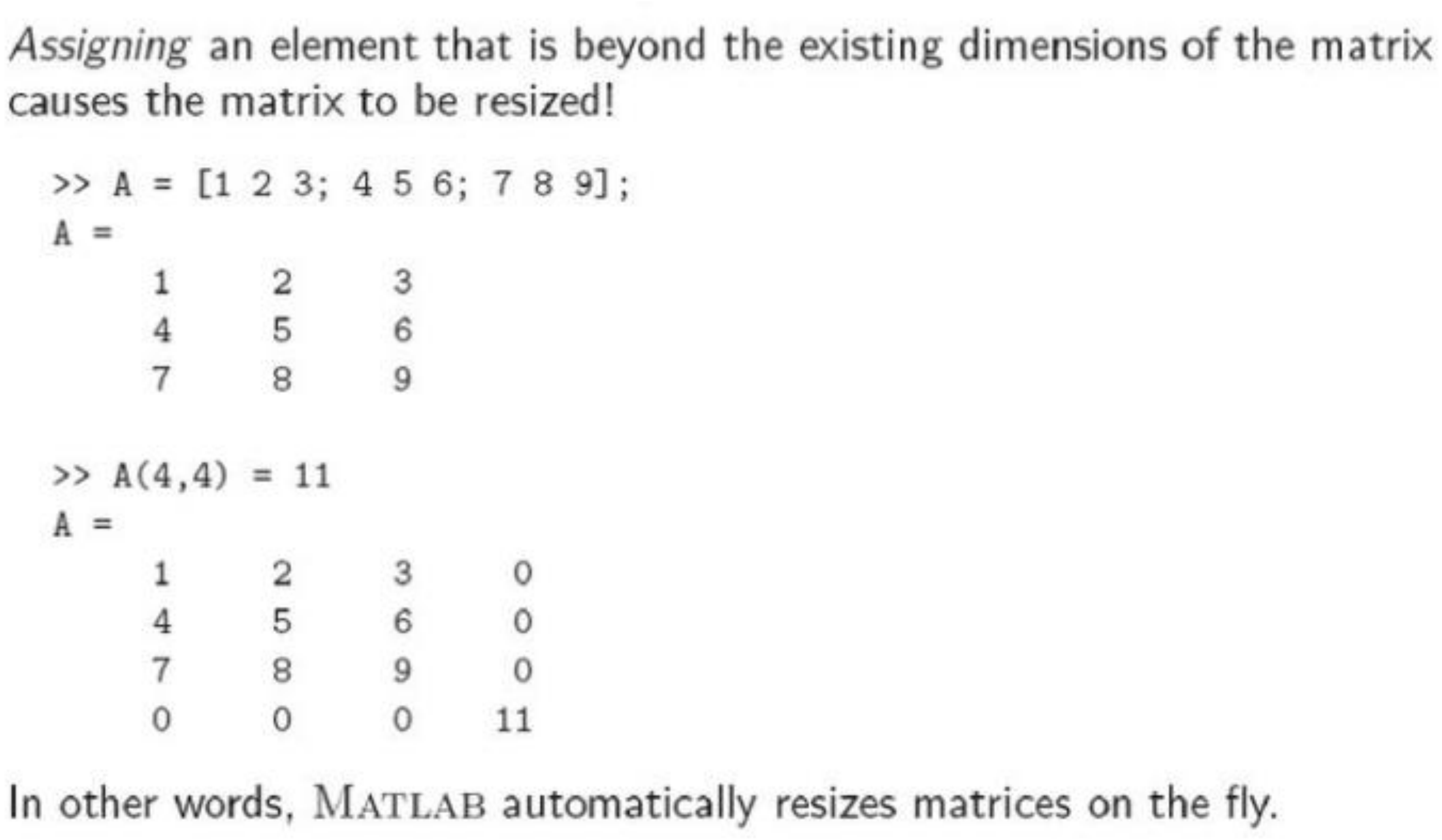
Subscript notation is also used to assign matrix elements

```
>> A(1,1) = c/b
A =
    0.2500    2.0000    3.0000
    4.0000    5.0000    6.0000
    7.0000    8.0000    9.0000
```

Referring to elements beyond the dimensions the matrix results in an error

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> A(1,4)
???  Index exceeds matrix dimensions.
```

Indexing Matrices (3)



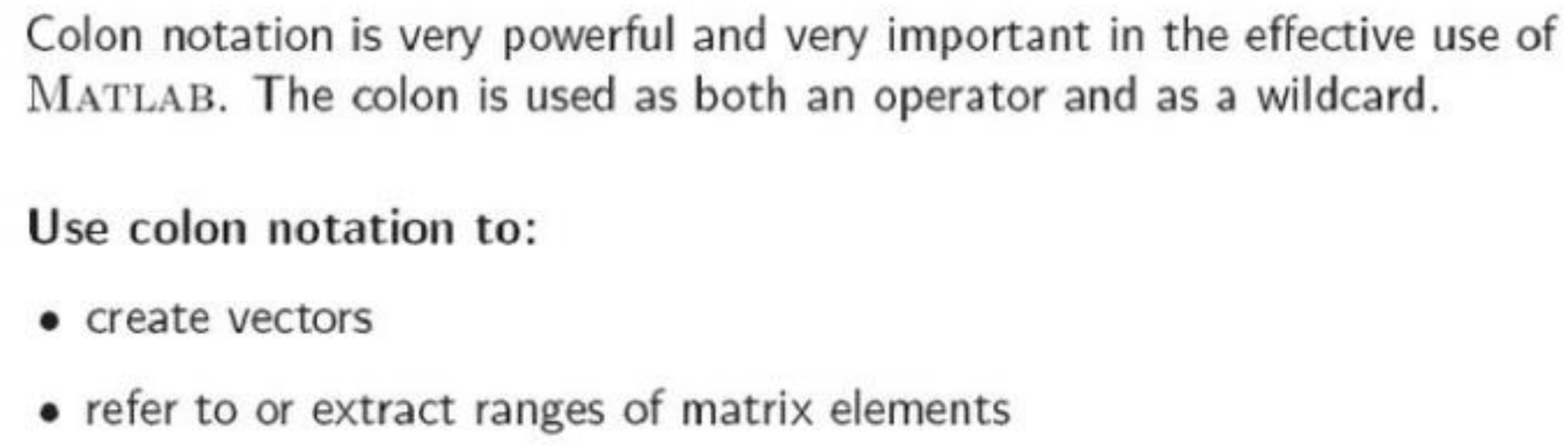
Assigning an element that is beyond the existing dimensions of the matrix causes the matrix to be resized!

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

>> A(4,4) = 11
A =
     1     2     3     0
     4     5     6     0
     7     8     9     0
     0     0     0    11
```

In other words, MATLAB automatically resizes matrices on the fly.

Colon notation



Colon notation is very powerful and very important in the effective use of MATLAB. The colon is used as both an operator and as a wildcard.

Use colon notation to:

- create vectors
- refer to or extract ranges of matrix elements

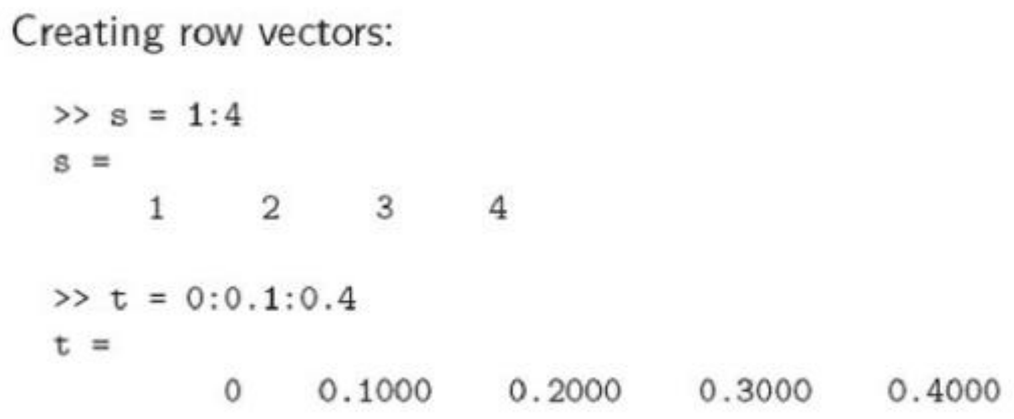
Colon Notation

Syntax:

```
startValue:endValue
startValue:increment:endValue
```

Note: *startValue*, *increment*, and *endValue* do not need to be integers

Colon Notation

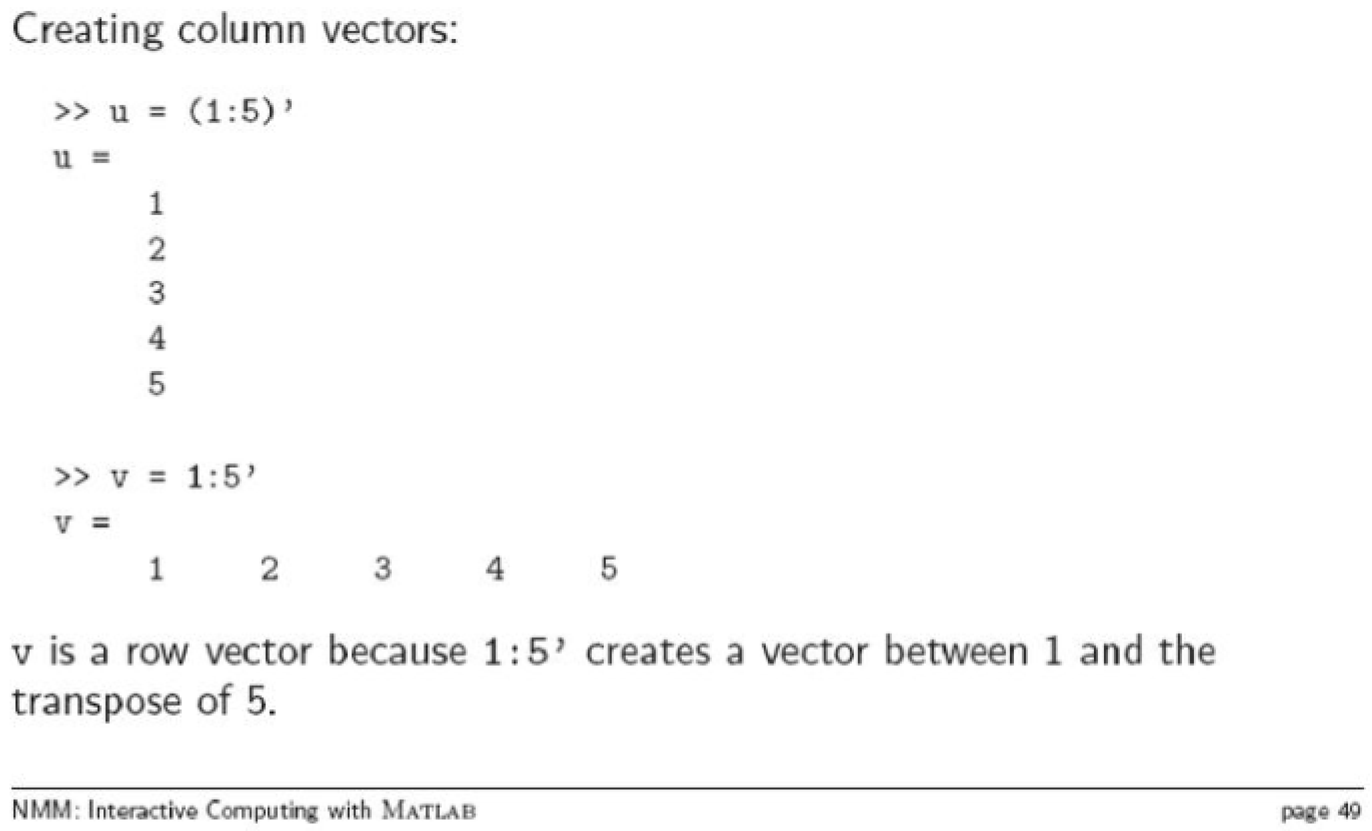


Creating row vectors:

```
>> s = 1:4
s =
     1     2     3     4

>> t = 0:0.1:0.4
t =
         0    0.1000    0.2000    0.3000    0.4000
```

Colon Notation



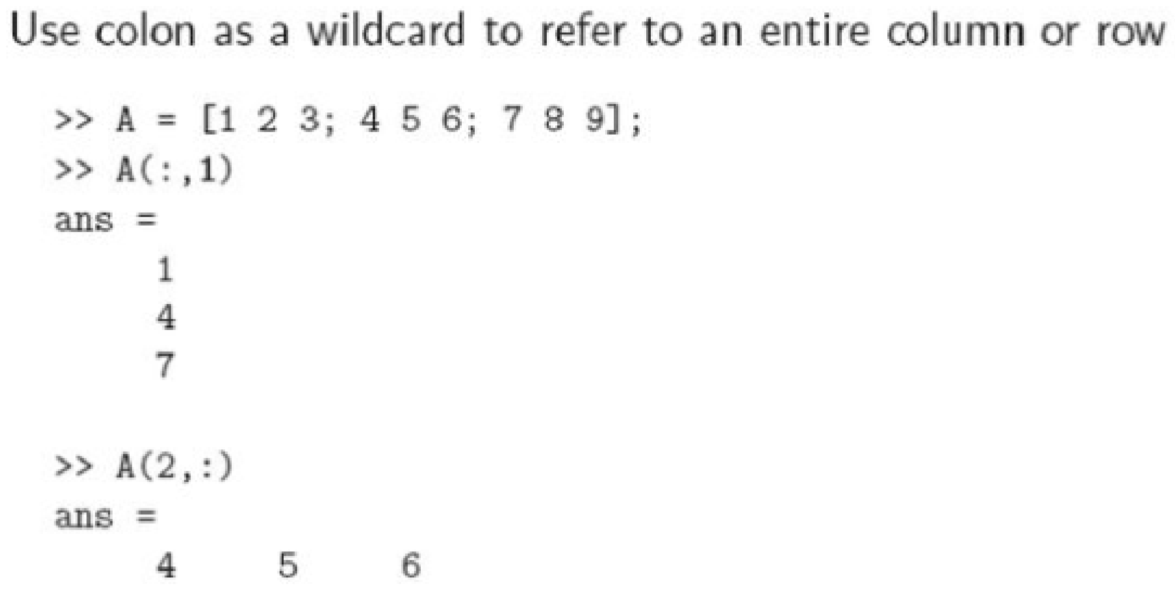
Creating column vectors:

```
>> u = (1:5)'
u =
      1
      2
      3
      4
      5

>> v = 1:5'
v =
      1      2      3      4      5
```

v is a row vector because `1:5'` creates a vector between 1 and the transpose of 5.

Colon Notation



Use colon as a wildcard to refer to an entire column or row

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> A(:,1)
ans =
     1
     4
     7

>> A(2,:)
ans =
     4     5     6
```

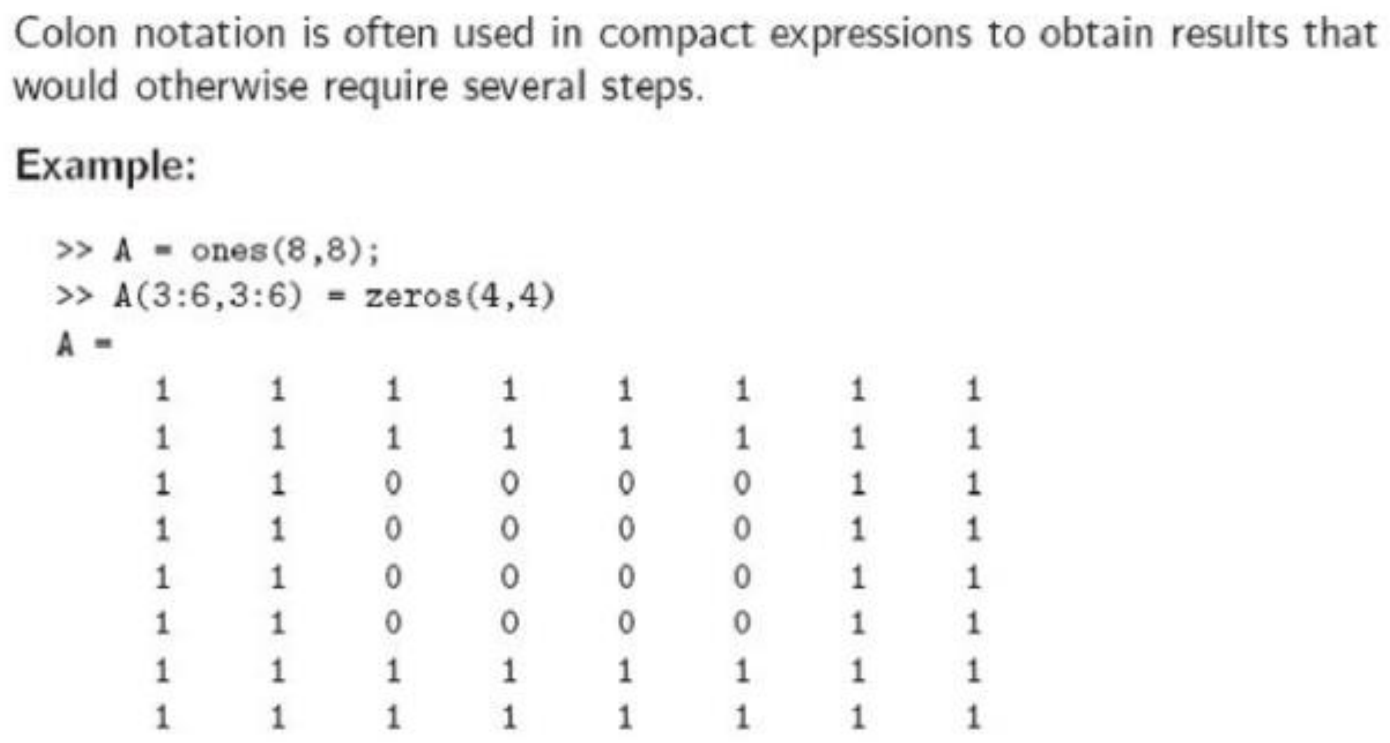
Colon Notation

Or use colon notation to refer to subsets of columns or rows

```
>> A(2:3,1)
ans =
     4
     7

>> A(1:2,2:3)
ans =
ans =
     2     3
     5     6
```

Colon Notation

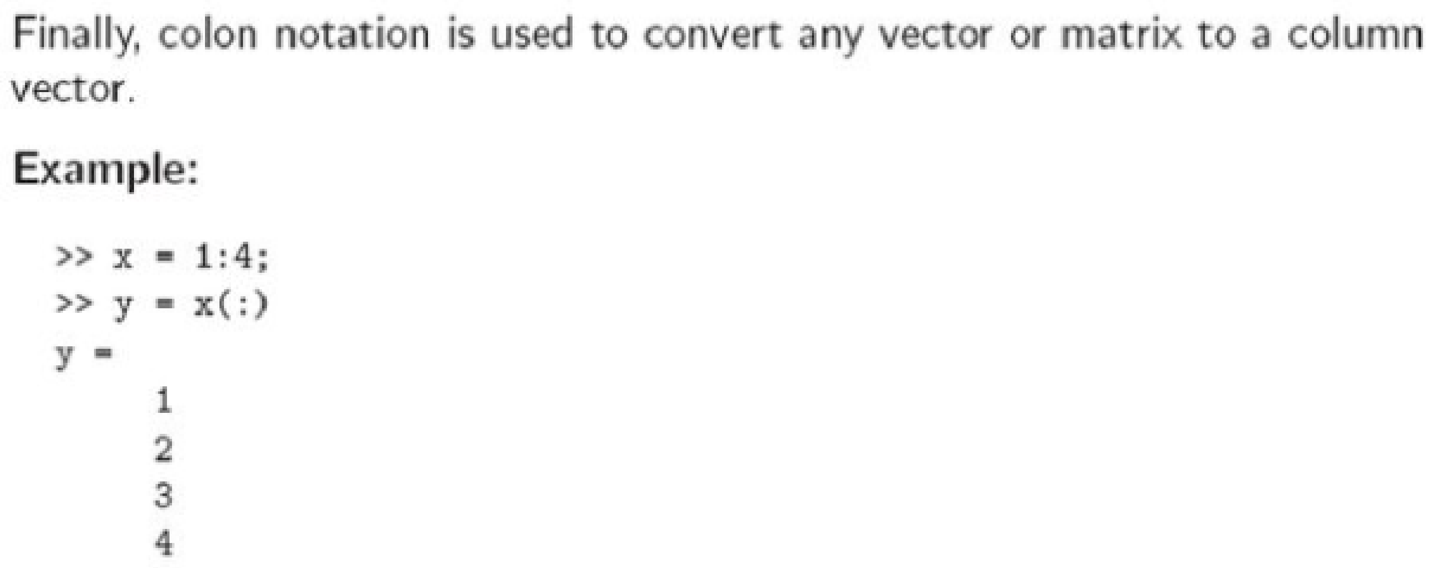


Colon notation is often used in compact expressions to obtain results that would otherwise require several steps.

Example:

```
>> A = ones(8,8);
>> A(3:6,3:6) = zeros(4,4)
A =
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     0     0     0     0     1     1
     1     1     0     0     0     0     1     1
     1     1     0     0     0     0     1     1
     1     1     0     0     0     0     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
```

Colon Notation

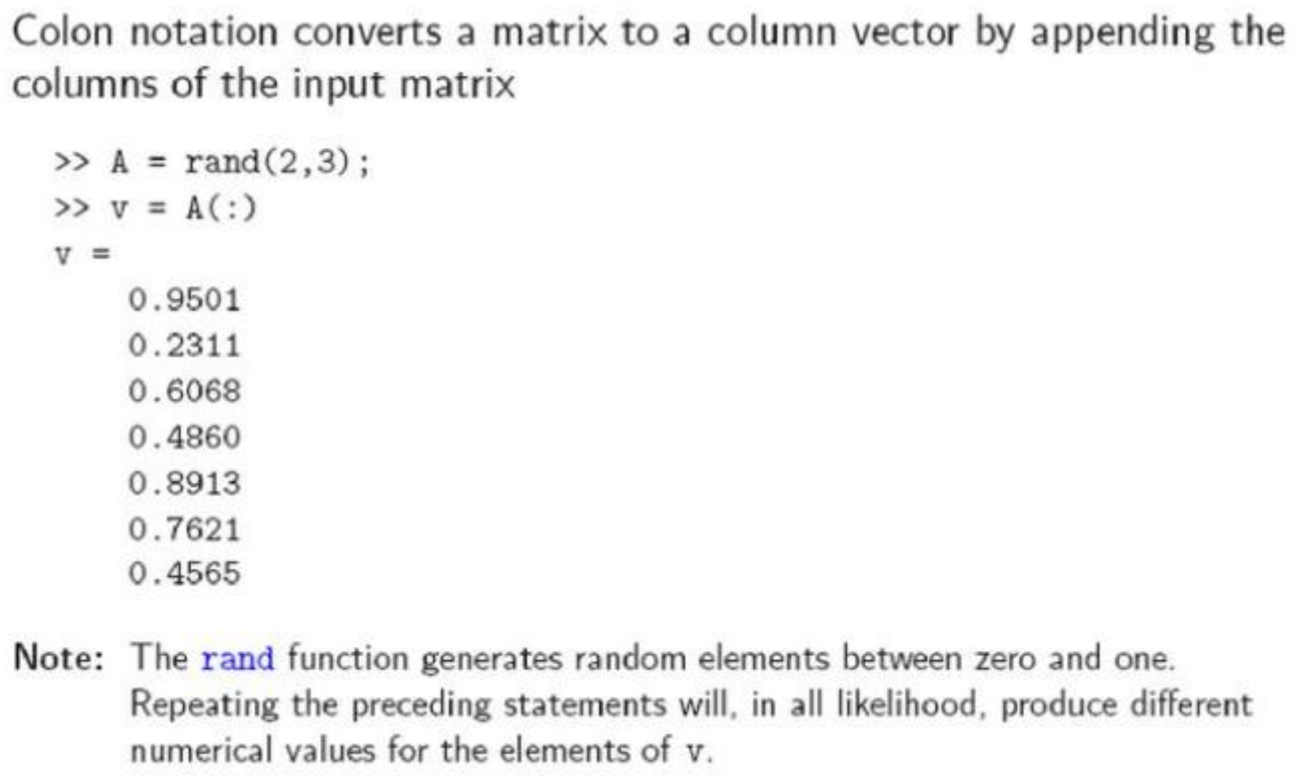


Finally, colon notation is used to convert any vector or matrix to a column vector.

Example:

```
>> x = 1:4;
>> y = x(:)
y =
      1
      2
      3
      4
```

Colon Notation



Colon notation converts a matrix to a column vector by appending the columns of the input matrix

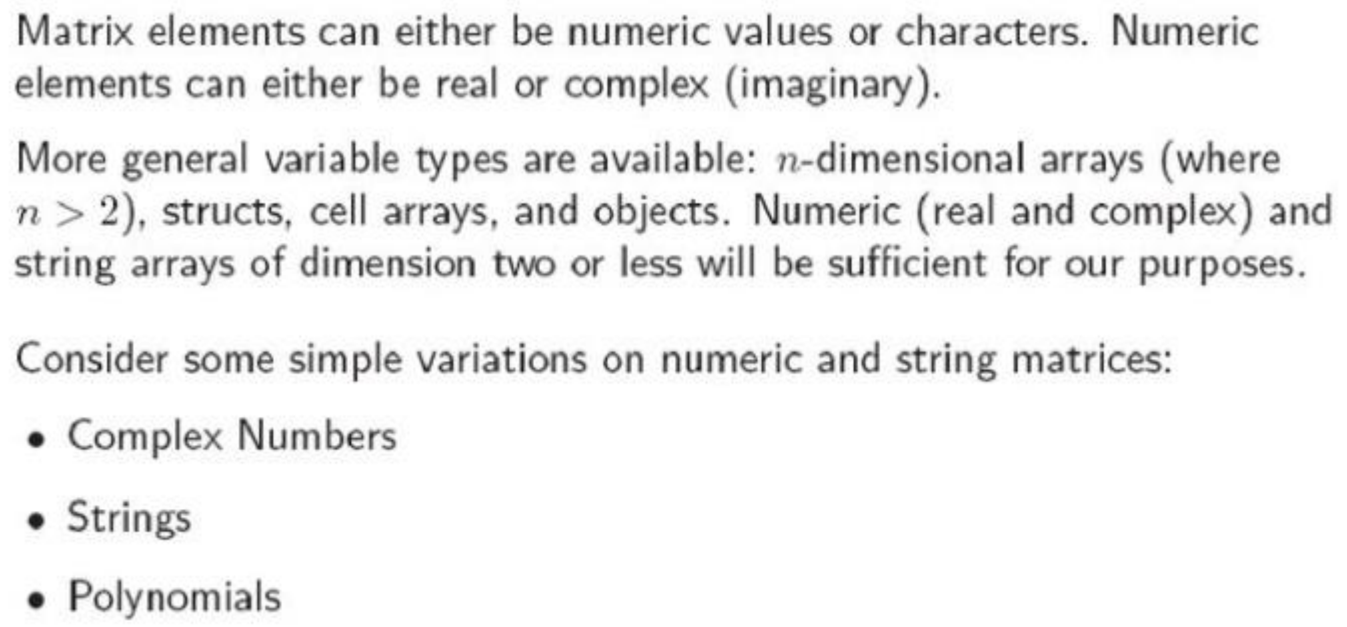
```
>> A = rand(2,3);
>> v = A(:)
v =
     0.9501
     0.2311
     0.6068
     0.4860
     0.8913
     0.7621
     0.4565
```

Note: The `rand` function generates random elements between zero and one. Repeating the preceding statements will, in all likelihood, produce different numerical values for the elements of `v`.

Exercise: Generating complex functions

- Generate a complex function $f(t) = 3ej3\pi t$ for t ranging from 0 to 1 in 0.001 increments.
- Step1: create a time variable,
  - `t = 0:0.001:1;`
- Step2: construct the vector,
  - `f = 3*exp(j*3*pi*t);`
- Step3: plot (t,f)

Additional Types of Variables



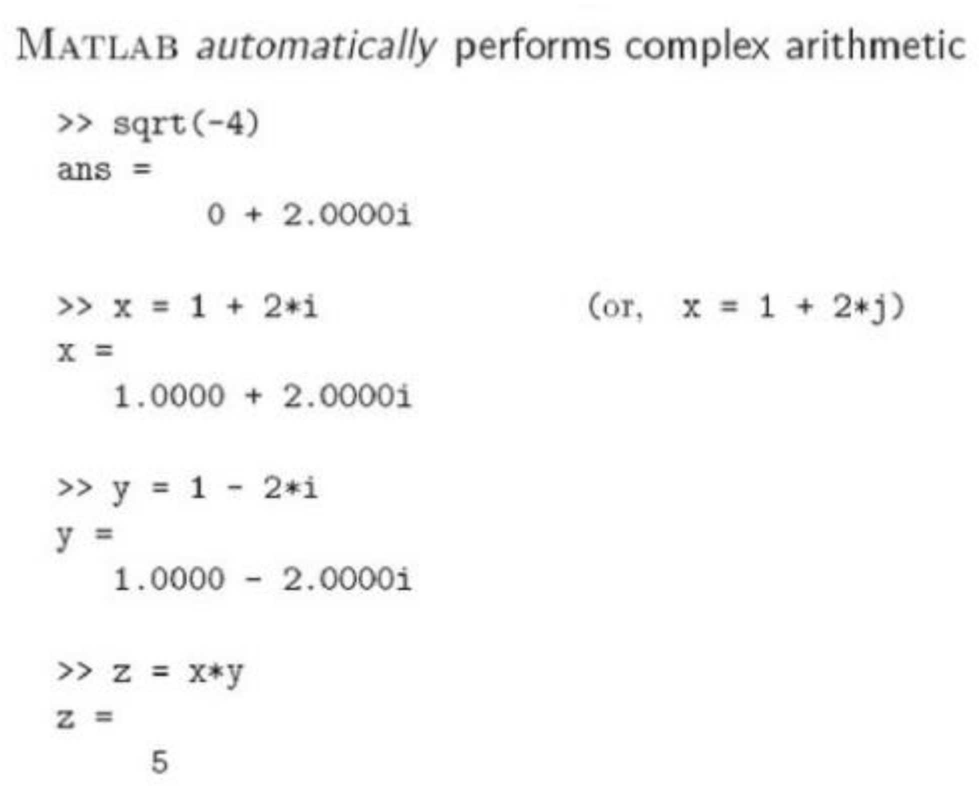
Matrix elements can either be numeric values or characters. Numeric elements can either be real or complex (imaginary).

More general variable types are available: n-dimensional arrays (where $n > 2$), structs, cell arrays, and objects. Numeric (real and complex) and string arrays of dimension two or less will be sufficient for our purposes.

Consider some simple variations on numeric and string matrices:

- Complex Numbers
- Strings
- Polynomials

Complex Numbers



MATLAB *automatically* performs complex arithmetic

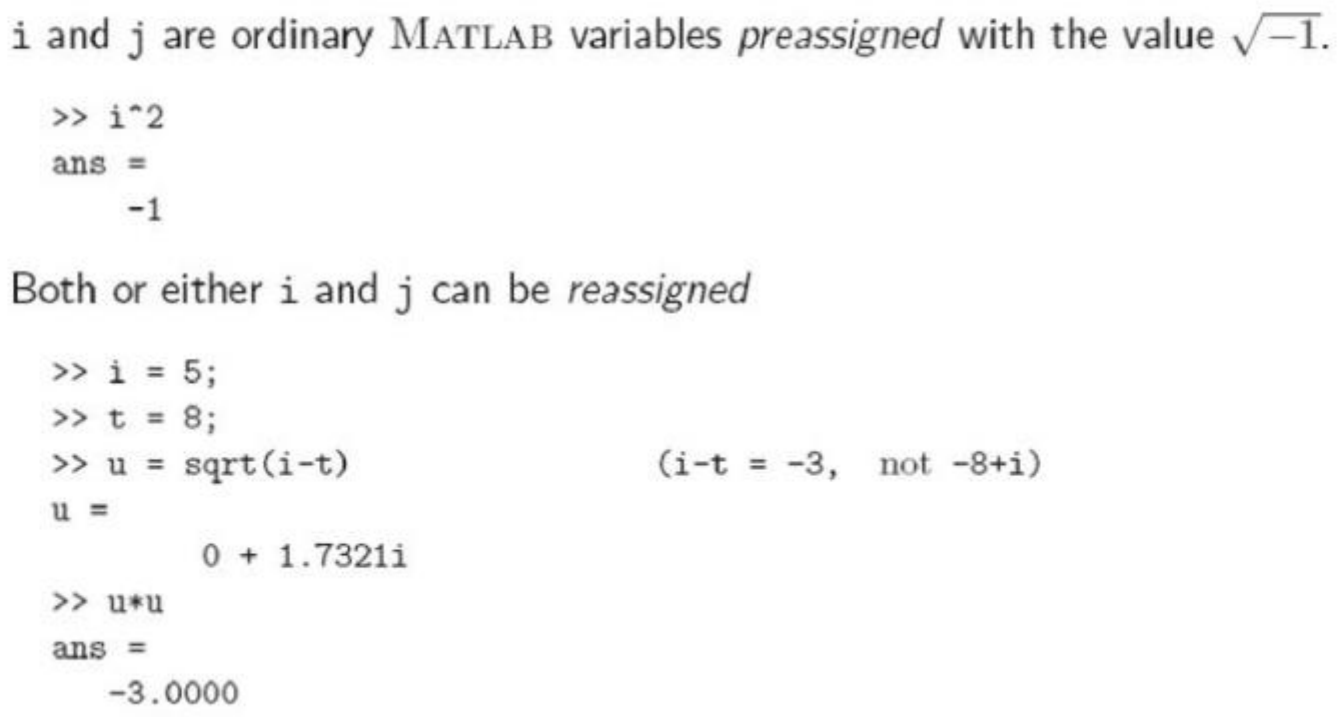
```
>> sqrt(-4)
ans =
        0 + 2.0000i

>> x = 1 + 2*i              (or,  x = 1 + 2*j)
x =
   1.0000 + 2.0000i

>> y = 1 - 2*i
y =
   1.0000 - 2.0000i

>> z = x*y
z =
     5
```

Unit Imaginary Numbers (1)



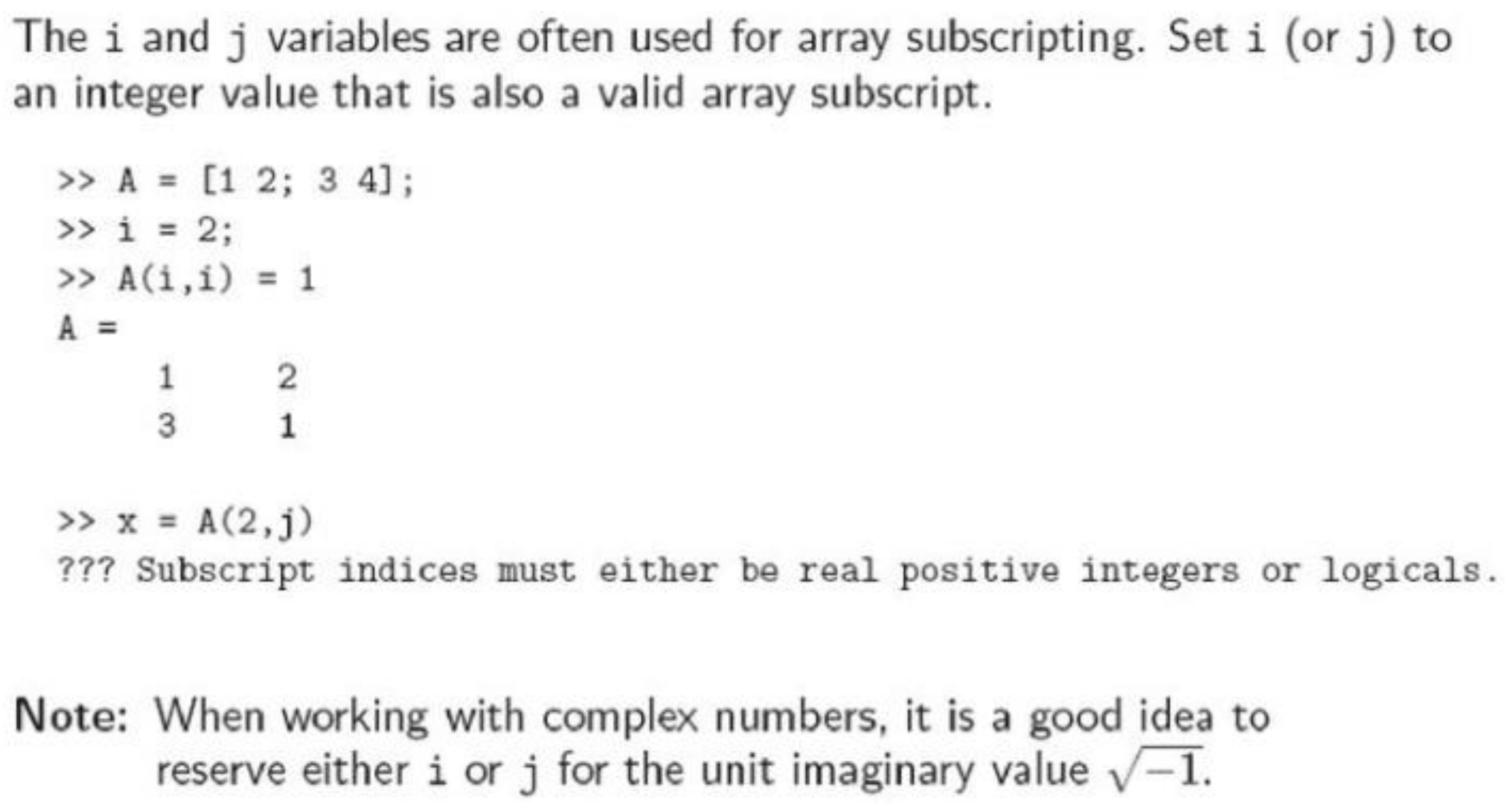
i and j are ordinary MATLAB variables *preassigned* with the value $\sqrt{-1}$.

```
>> i^2
ans =
    -1
```

Both or either i and j can be *reassigned*

```
>> i = 5;
>> t = 8;
>> u = sqrt(i-t)                    (i-t = -3,  not -8+i)
u =
         0 + 1.7321i
>> u*u
ans =
   -3.0000
```

Unit Imaginary Numbers (2)



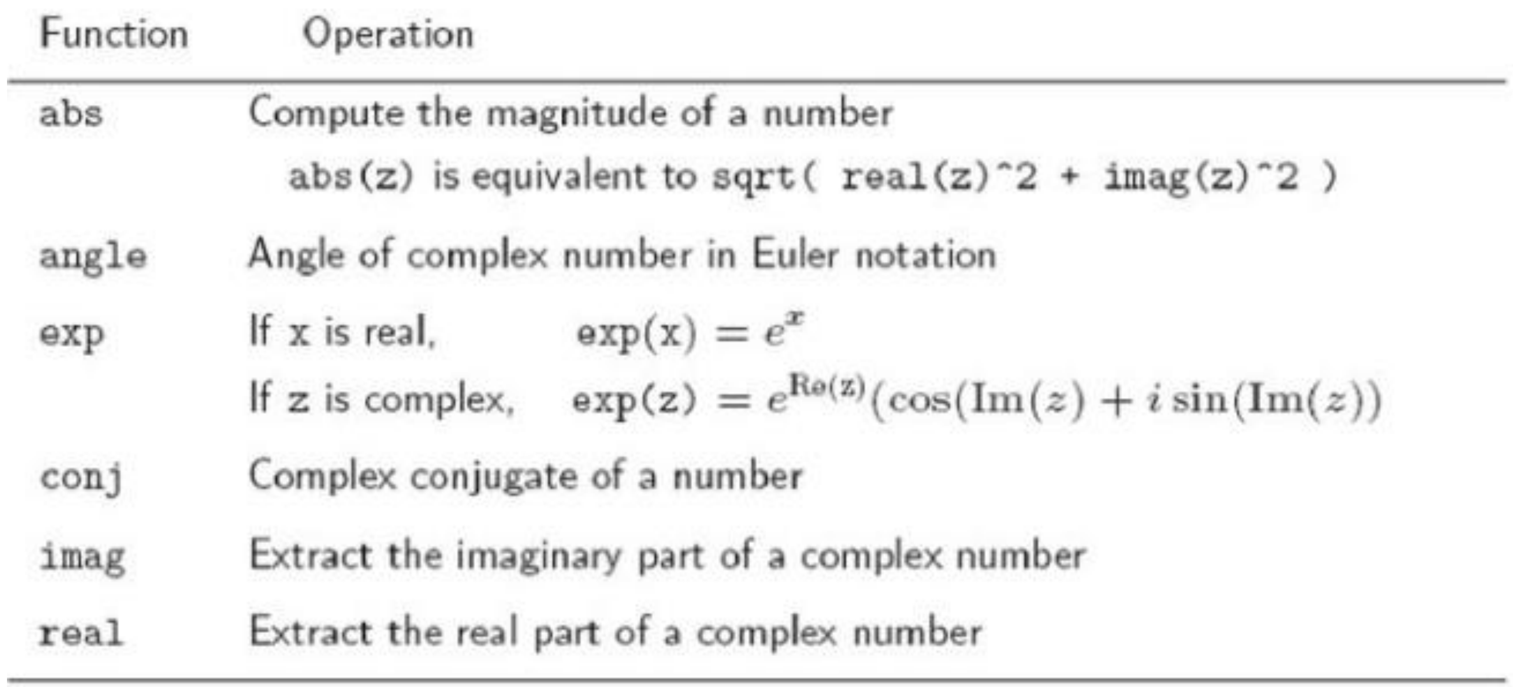
The i and j variables are often used for array subscripting. Set i (or j) to an integer value that is also a valid array subscript.

```
>> A = [1 2; 3 4];
>> i = 2;
>> A(i,i) = 1
A =
     1     2
     3     1

>> x = A(2,j)
??? Subscript indices must either be real positive integers or logicals.
```

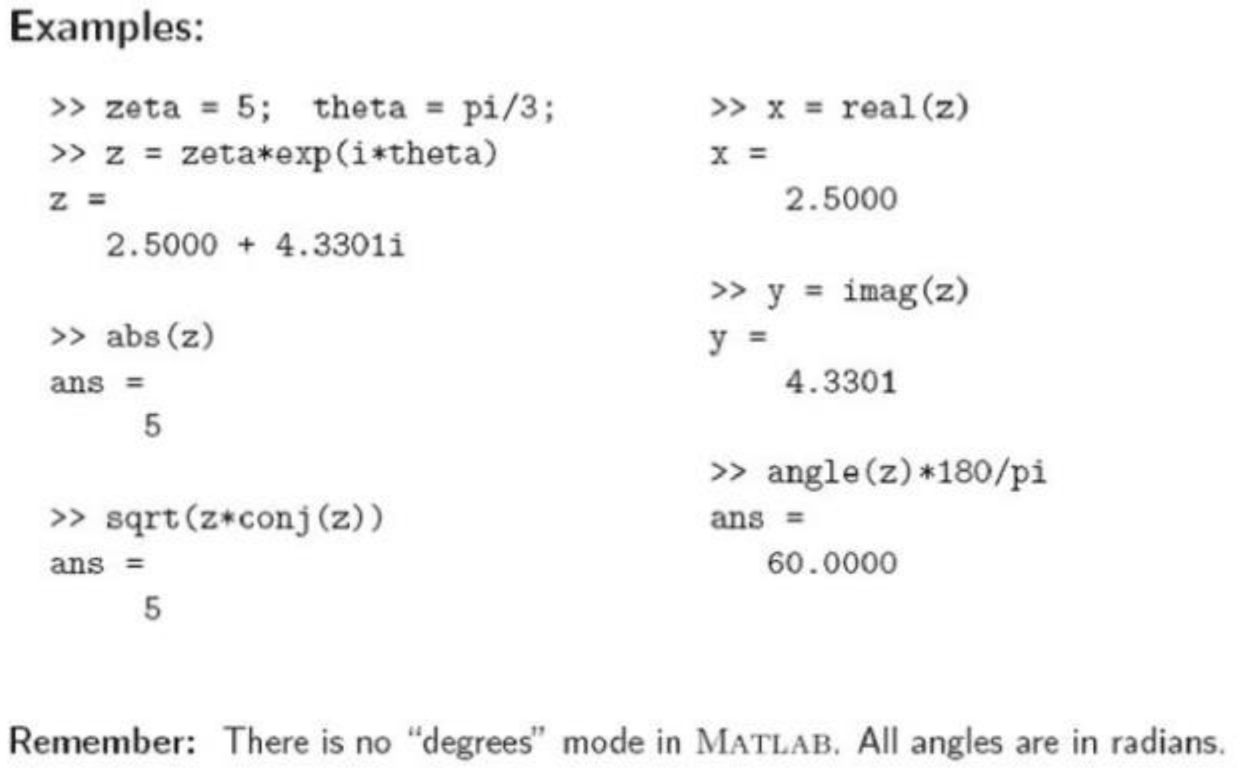
Note: When working with complex numbers, it is a good idea to reserve either i or j for the unit imaginary value $\sqrt{-1}$.

Functions for Complex Arithmetic (1)



Function	Operation
abs	Compute the magnitude of a number
	abs(z) is equivalent to sqrt(real(z)^2 + imag(z)^2)
angle	Angle of complex number in Euler notation
exp	If x is real, $\mathtt{exp(x)} = e^x$
	If z is complex, $\mathtt{exp(z)} = e^{\mathrm{Re(z)}}(\cos(\mathrm{Im}(z) + i\sin(\mathrm{Im}(z))$
conj	Complex conjugate of a number
imag	Extract the imaginary part of a complex number
real	Extract the real part of a complex number

Functions for Complex Arithmetic



Examples:

```
>> zeta = 5;  theta = pi/3;
>> z = zeta*exp(i*theta)
z =
   2.5000 + 4.3301i

>> abs(z)
ans =
     5

>> sqrt(z*conj(z))
ans =
     5
```

```
>> x = real(z)
x =
    2.5000

>> y = imag(z)
y =
    4.3301

>> angle(z)*180/pi
ans =
   60.0000
```

Remember: There is no "degrees" mode in MATLAB. All angles are in radians.

Exercise 1

Evaluating complex variables and Expressions

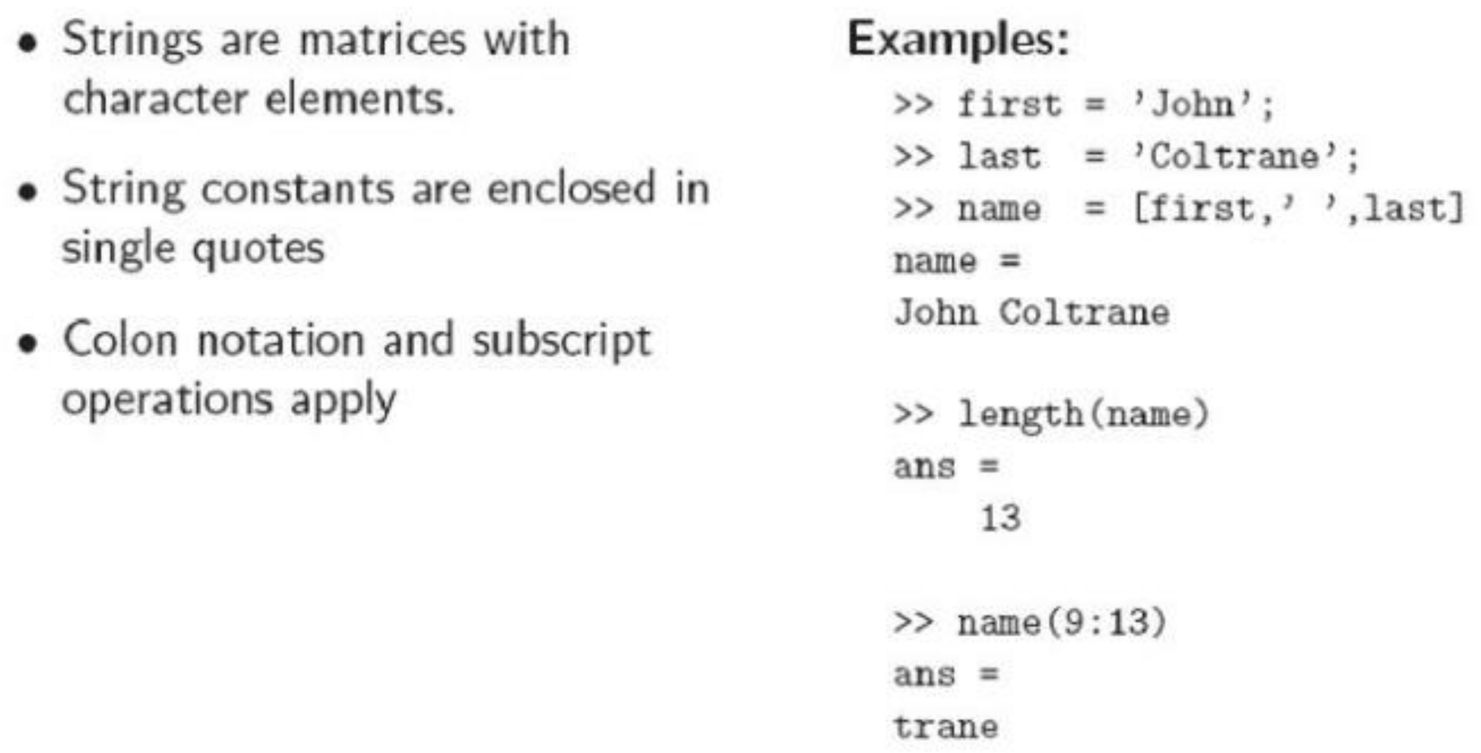
- Express the following complex numbers in Cartesian form. Plot it in the complex plane

a) $y = je^{j11\pi/4}$

b) $y= (1-j)^{10}$

```
plot(y,'x')
```

Strings



- Strings are matrices with character elements.
- String constants are enclosed in single quotes
- Colon notation and subscript operations apply

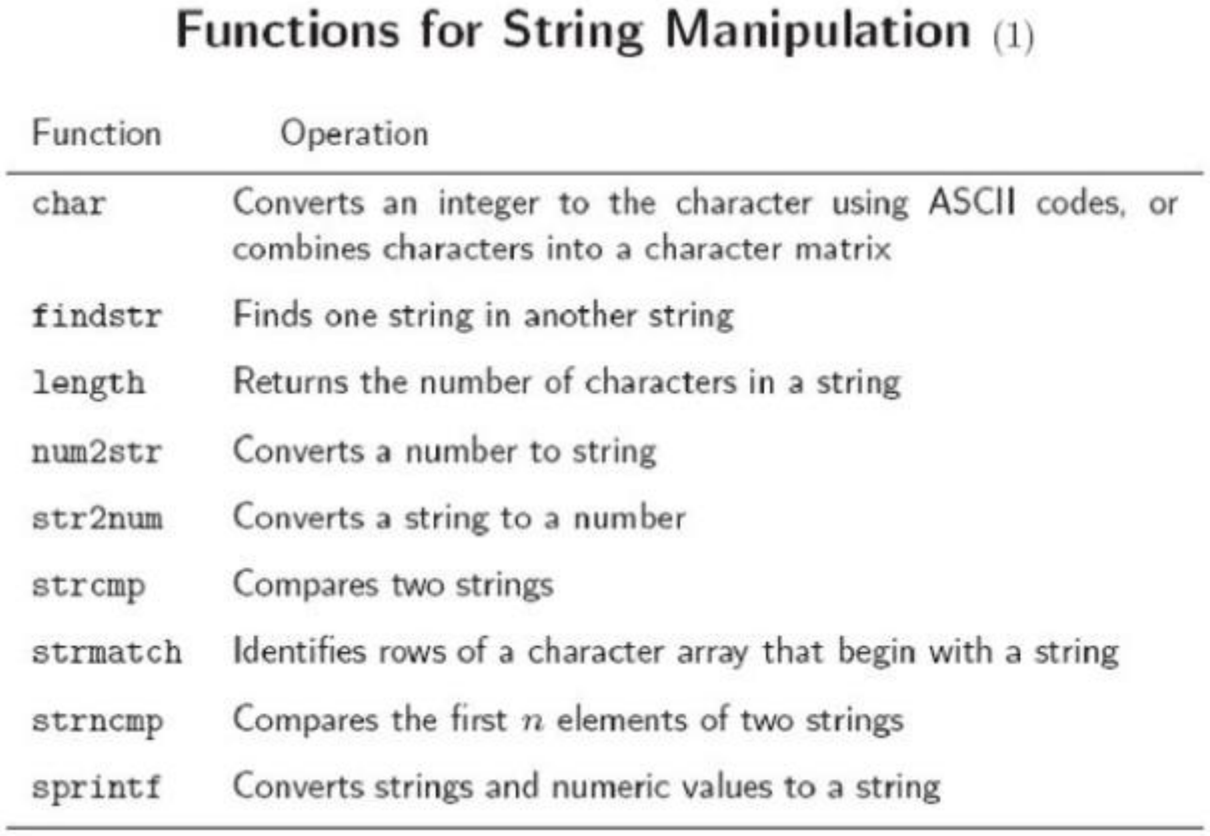
Examples:

```
>> first = 'John';
>> last  = 'Coltrane';
>> name  = [first,' ',last]
name =
John Coltrane

>> length(name)
ans =
    13

>> name(9:13)
ans =
trane
```

Functions for String Manipulation (1)



Functions for String Manipulation (1)

Function	Operation
`char`	Converts an integer to the character using ASCII codes, or combines characters into a character matrix
`findstr`	Finds one string in another string
`length`	Returns the number of characters in a string
`num2str`	Converts a number to string
`str2num`	Converts a string to a number
`strcmp`	Compares two strings
`strmatch`	Identifies rows of a character array that begin with a string
`strncmp`	Compares the first n elements of two strings
`sprintf`	Converts strings and numeric values to a string

Functions for String Manipulation (2)

num2str converts a number to a string

```
>> msg1 = ['There are ',num2str(100/2.54),' inches in a meter']
msg1 =
There are 39.3701 inches in a meter
```

For greater control over format of the number-to-string conversion, use sprintf

```
>> msg2 = sprintf('There are %5.2f cubic inches in a liter',1000/2.54^3)
msg2 =
There are 61.02 cubic inches in a liter
```

The MATLAB sprintf function is similar to the C function of the same name, but it uses single quotes for the format string.

Functions for String Manipulation (3)

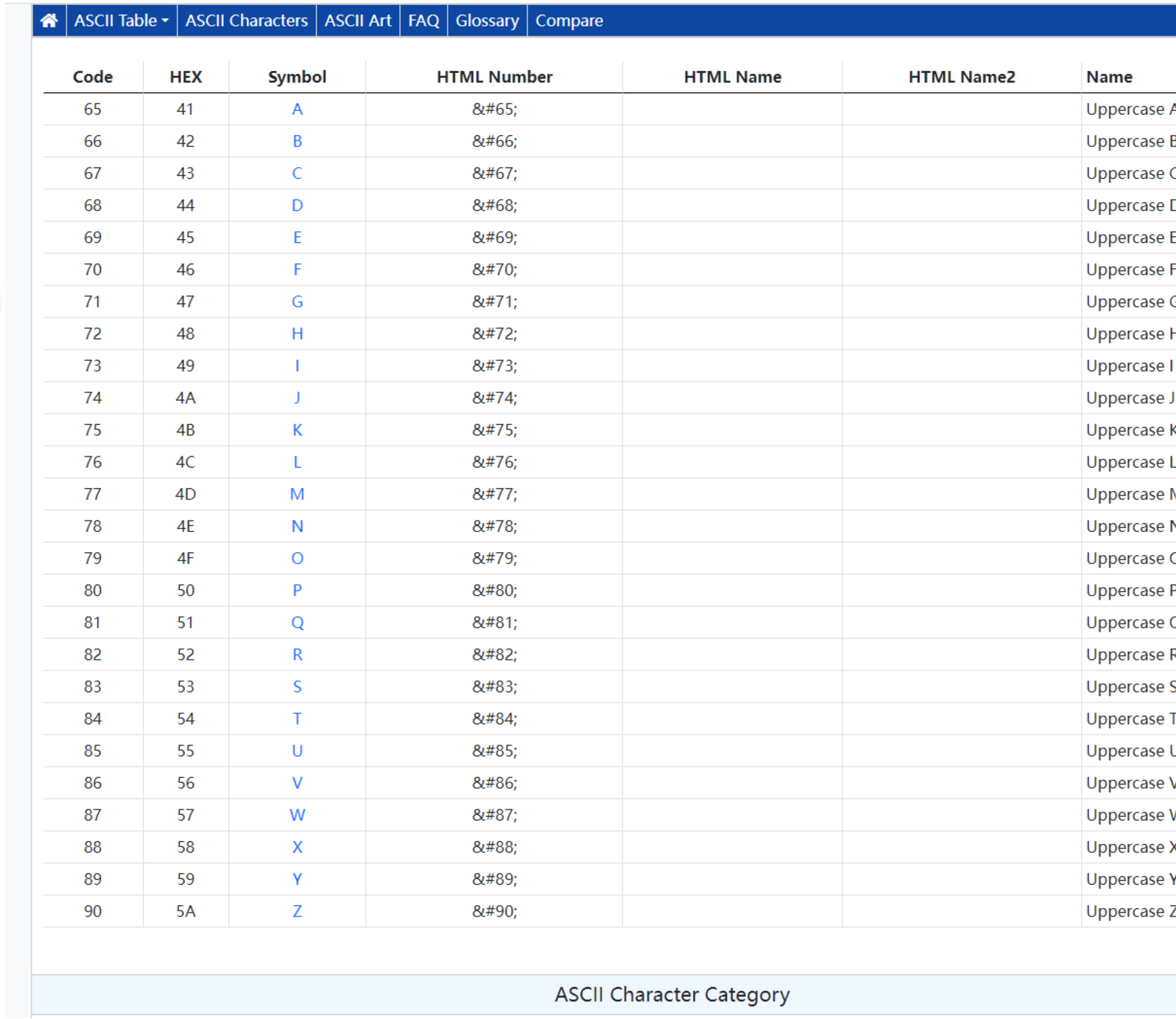
The char function can be used to combine strings

```
>> both = char(msg1,msg2)
both =
There are 39.3701 inches in a meter
There are 61.02 cubic inches in a liter
```

or to refer to individual characters by their ASCII codes[1]

```
>> char(49)
ans =
1
>> char([77 65 84 76 65 66])
ans =
MATLAB
```

[1]See e.g., www.asciicodes.com or wikipedia.org/wiki/ASCII.



ASCII Table ▾ | ASCII Characters | ASCII Art | FAQ | Glossary | Compare

Code	HEX	Symbol	HTML Number	HTML Name	HTML Name2	Name
65	41	A	&#65;			Uppercase A
66	42	B	&#66;			Uppercase B
67	43	C	&#67;			Uppercase C
68	44	D	&#68;			Uppercase D
69	45	E	&#69;			Uppercase E
70	46	F	&#70;			Uppercase F
71	47	G	&#71;			Uppercase G
72	48	H	&#72;			Uppercase H
73	49	I	&#73;			Uppercase I
74	4A	J	&#74;			Uppercase J
75	4B	K	&#75;			Uppercase K
76	4C	L	&#76;			Uppercase L
77	4D	M	&#77;			Uppercase M
78	4E	N	&#78;			Uppercase N
79	4F	O	&#79;			Uppercase O
80	50	P	&#80;			Uppercase P
81	51	Q	&#81;			Uppercase Q
82	52	R	&#82;			Uppercase R
83	53	S	&#83;			Uppercase S
84	54	T	&#84;			Uppercase T
85	55	U	&#85;			Uppercase U
86	56	V	&#86;			Uppercase V
87	57	W	&#87;			Uppercase W
88	58	X	&#88;			Uppercase X
89	59	Y	&#89;			Uppercase Y
90	5A	Z	&#90;			Uppercase Z

ASCII Character Category

Polynomials

MATLAB polynomials are stored as vectors of coefficients. The polynomial coefficients are stored in *decreasing powers* of x

$$P_n(x) = c_1x^n + c_2x^{n-1} + \ \ldots \ + c_nx + c_{n+1}$$

Example: Evaluate $x^3 - 2x + 12$ at $x = -1.5$

Store the coefficients of the polynomial in vector c:

```
>> c = [1  0  -2  12];
```

Use the built-in `polyval` function to evaluate the polynomial.

```
>> polyval(c,1.5)
ans =
   12.3750
```

Functions for Manipulating Polynomials

Function	Operations performed
`conv`	Product (convolution) of two polynomials
`deconv`	Division (deconvolution) of two polynomials
`poly`	Create a polynomial having specified roots
`polyder`	Differentiate a polynomial
`polyval`	Evaluate a polynomial
`polyfit`	Polynomial curve fit
`roots`	Find roots of a polynomial

Vector Addition and Subtraction

Vector and addition and subtraction are element-by-element operations.

Example:

```
>> u = [10 9 8];                    (u and v are row vectors)
>> v = [1 2 3];
>> u+v
ans =
    11    11    11

>> u-v
ans =
     9     7     5
```

Vectorization

- **Vectorization** is the use of single, compact expressions that operate on all elements of a vector without explicitly writing the code for a loop. The loop *is* executed by the MATLAB kernel, which is much more efficient at evaluating a loop in interpreted MATLAB code.
- Vectorization allows calculations to be expressed succintly so that programmers get a high level (as opposed to detailed) view of the operations being performed.
- Vectorization is important to make MATLAB operate efficiently[2].

[2]Recent versions of MATLAB have improved the efficiency for some non-vectorized code.

Vectorization of Built-in Functions

Most built-in function support *vectorized* operations. If the input is a scalar the result is a scalar. If the input is a vector or matrix, the output is a vector or matrix with the same number of rows and columns as the input.

Example:

```
>> x = 0:pi/4:pi               (define a row vector)
x =
         0    0.7854    1.5708    2.3562    3.1416

>> y = cos(x)                  (evaluate cosine of each x(i))
y =
    1.0000    0.7071         0   -0.7071   -1.0000
```

Vectorized Calculations (6)

More examples

```
>> A = pi*[ 1 2; 3 4]
A =
    3.1416    6.2832
    9.4248   12.5664

>> S = sin(A)
S =
     0     0
     0     0
```

```
>> B = A/2
B =
    1.5708    3.1416
    4.7124    6.2832

>> T = sin(B)
T =
     1     0
    -1     0
```

Array Operators

Array operators support element-by-element operations that are not defined by the rules of linear algebra.

Array operators have a period prepended to a standard operator.

Symbol	Operation
`.*`	element-by-element multiplication
`./`	element-by-element "right" division
`.\`	element-by-element "left" division
`.^`	element-by-element exponentiation

Array operators are a very important tool for writing vectorized code.

Using Array Operators (1)

Examples: Element-by-element multiplication and division

```
>> u = [1 2 3];
>> v = [4 5 6];
```

Use .* and ./ for element-by-element multiplication and division

```
>> w = u.*v
w =
     4    10    18

>> x = u./v
x =
    0.2500    0.4000    0.5000
```

Using Array Operators (1)

Examples: Element-by-element multiplication and division

```
>> u = [1 2 3];
>> v = [4 5 6];
>> y = sin(pi*u/2) .* cos(pi*v/2)
y =
     1     0     1

>> z = sin(pi*u/2) ./ cos(pi*v/2)

Warning: Divide by zero.
z =
     1   NaN     1
```

Using Array Operators (2)

Examples: Application to matrices

```
>> A = [1 2 3 4; 5 6 7 8];
>> B = [8 7 6 5; 4 3 2 1];
>> A.*B
ans =
     8    14    18    20
    20    18    14     8

>> A*B
??? Error using ==> *
Inner matrix dimensions must agree.
```

The last statement causes an error because the number of columns in `A` is not equal to the number of rows in `B` — a requirement for `A` and `B` to be compatible for matrix multiplication.

Using Array Operators (3)

```
>> A = [1 2 3 4; 5 6 7 8];
>> B = [8 7 6 5; 4 3 2 1];
>> A*B'
ans =
    60     20
   164     60
```

The number of columns in `A` is equal to the number of rows in B^T, so `A*B'` is a legal matrix-matrix multiplication.

Array operators also apply to matrix powers.

```
>> A.^2
ans =
     1      4      9     16
    25     36     49     64
```

Built-in Functions

Many standard mathematical functions, such as sin, cos, log, and log10, are built-in.

Example:

log(x) computes the natural logarithm of x

log10(x) is the base 10 logarithm

log2(x) is the base 2 logarithm

```
>> log(256)
ans =
5.5452

>> log10(256)    ans =
2.4082

>> log2(256) ans =
8
```

Ways to Get Help

- Use on-line help to request info on a specific function

```
>> help sqrt
```

- In Matlab version 6 and later the doc function opens the on-line version of the manual. This is very helpful for more complex commands.

```
>> doc plot
```

- Use lookfor to find functions by keywords

```
>> lookfor functionName
```

On-line Help (1)

Syntax:

```
help functionName
```

Example:

```
>> help log
```

produces

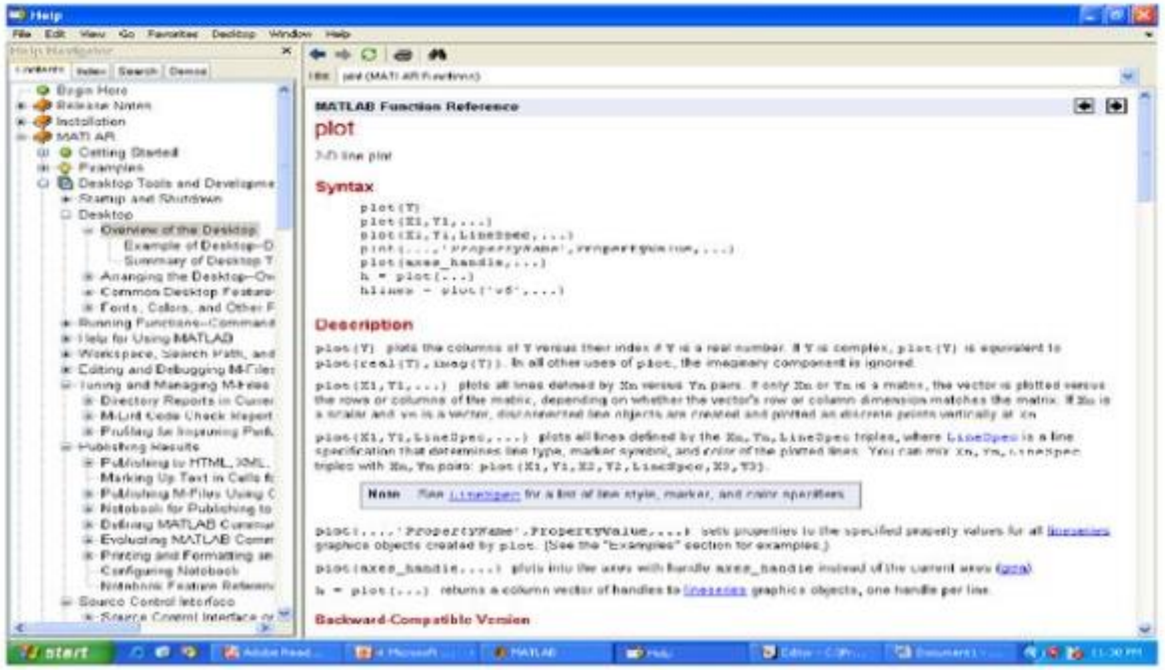
```
 LOG Natural logarithm.
     LOG(X) is the natural logarithm of the elements of X.
     Complex results are produced if X is not positive.
     See also LOG2, LOG10, EXP, LOGM.
```

The help function provides a compact summary of how to use a command. Use the doc function to get more in-depth information.

On-line Help (2)

The help browser opens when you type a doc command:

```
>> doc plot
```



MATLAB Function Reference

plot

2-D line plot

Syntax

```
plot(Y)
plot(X1,Y1,...)
plot(X1,Y1,LineSpec,...)
plot(...,'PropertyName',PropertyValue,...)
plot(axes_handle,...)
h = plot(...)
hlines = plot('v6',...)
```

Description

plot(Y) plots the columns of Y versus their index if Y is a real number. If Y is complex, plot(Y) is equivalent to plot(real(Y),imag(Y)). In all other uses of plot, the imaginary component is ignored.

plot(X1,Y1,...) plots all lines defined by Xn versus Yn pairs. If only Xn or Yn is a matrix, the vector is plotted versus the rows or columns of the matrix, depending on whether the vector's row or column dimension matches the matrix. If Xn is a scalar and Yn is a vector, disconnected line objects are created and plotted as discrete points vertically at Xn.

plot(X1,Y1,LineSpec,...) plots all lines defined by the Xn,Yn,LineSpec triples, where LineSpec is a line specification that determines line type, marker symbol, and color of the plotted lines. You can mix Xn,Yn,LineSpec triples with Xn,Yn pairs: plot(X1,Y1,X2,Y2,LineSpec,X3,Y3).

Note See LineSpec for a list of line style, marker, and color specifiers.

plot(...,'PropertyName',PropertyValue,...) sets properties to the specified property values for all lineseries graphics objects created by plot. (See the "Examples" section for examples.)

plot(axes_handle,...) plots into the axes with handle axes_handle instead of the current axes (gca).

h = plot(...) returns a column vector of handles to lineseries graphics objects, one handle per line.

Backward-Compatible Version

Looking for Functions

Syntax:

```
            lookfor   string
```

searches first line of function descriptions for “string”.

Example:

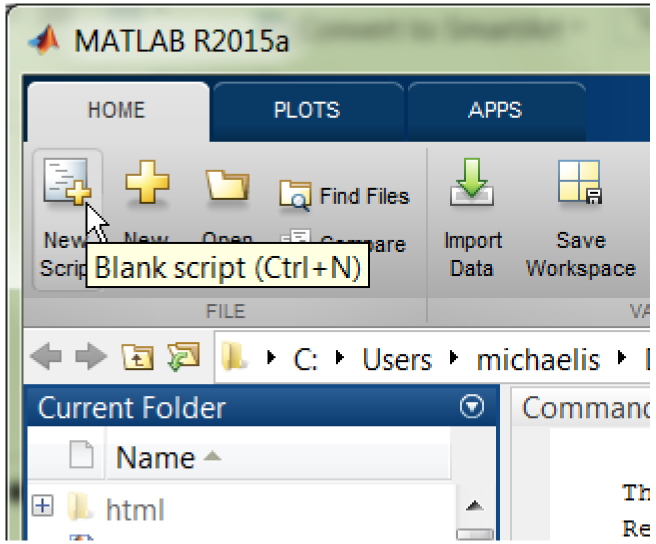
```
>> lookfor cosine
```

produces

```
   ACOS Inverse cosine.
   ACOSH Inverse hyperbolic cosine.
   COS Cosine.
   COSH Hyperbolic cosine.
```

Creating a MATLAB Program/Script

- Click on “New Script” at the upper left corner of the “Home” tab to create a new script (*.m) file.
- The MATLAB editor window is opened whenever a new script is created or an existing one is opened.



Scripts with input and output

```
% This script calculates the area of a circle
% It prompts the user for the radius

% Prompt the user for the radius and calculate
% the area based on that radius
radius = input(‘Please enter the radius: ’);
area = pi * (radius^2);
% Print all variables in a sentence format
fprintf(‘For a circle with a radius of %.2f,’,radius)
fprintf(‘the area is %.2f\n’,area)
```