

Shawna Lockhart
Eric Tilleson

AN ENGINEER'S INTRODUCTION TO
**PROGRAMMING WITH
MATLAB® 2018**



Visit the following websites to learn more about this book:



[amazon.com](https://www.amazon.com)

[Google books](https://books.google.com)

[BARNES & NOBLE](https://www.barnesandnoble.com)

PROGRAMMING BASICS: OPERATORS & VARIABLES

Introduction

This tutorial gets you started working with MATLAB as a programming language. There are a whopping number of programming languages out there, but a lot of them use the same handful of basic elements. This means that after you learn your first programming language, you've got a conceptual leg up on many of the rest and only need to learn new syntax (the "grammar" rules) to hit the ground running. The basic concepts will be familiar if you have done a bit of programming, but if so, look for MATLAB-specific syntax and usage information.

Operators

Operator is a fancy name for symbols like +, -, and <. Operators are normally broken into usage-based categories: arithmetic, relational, and logical. More operators will be discussed when we get to matrices.

Arithmetic Operators

Arithmetic operators are used on numbers and return a number. You used them in our earlier tutorial and they are similar to functions on a calculator, so these should seem familiar.

Operator	Action
+	Addition (3 + 2)
-	Subtraction (3 - 2)
*	Multiplication (3 * 2)
/	Division (3 / 2)
^	Raise to the power (3 ^ 2)

Relational Operators

Relational operators make a comparison. The comparison results in a *logical value*. A logical value is one that has just two states: true or false, which are represented by 1 for true and 0 for false.

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equals
~=	Does not equal

2

Objectives

When you have completed this tutorial, you will be able to

1. Understand and apply operator precedence.
2. Assign a value to a variable.
3. Understand the minimum and maximum value of various numeric variable types.
4. Understand the difference between string and character arrays.
5. Work with data from an imported file.

Note: *MATLAB* is the name of both the programming environment (the MATLAB program itself) and the programming language used in that environment. Both uses are fairly interchangeable in conversation, if you ever have those sorts of conversations.

Tip: *Operators also have function forms; for instance plus(3,4), which does the same thing as 3 + 4. Their reason for being is beyond our scope and we won't be using them. See the MATLAB help section on operators if you're curious to learn more.*

Tip: If you have used MathCAD, you may know that it also uses different types of equal signs for various operators but they are not the same ones as Matlab uses! Ha ha.

Tip: Don't type the `>>`. Remember that `>>` is the prompt in the Command Window.

Tip: Remember variables are placeholders for data (numerical values, text strings, etc.). We will talk about them in more detail soon. We are using `x` as our variable name. We could have named it Jacobson or `xcd123` (but not `123xcd` as variable names must start with a letter). Our variable, `x`, will stay equal to 3 until we store some other value in variable `x`, or we clear the Workspace.

Tip: The MATLAB data type `logical` is similar to the Boolean type in other languages.

Not All Equals are Equal

A quick word about the *equals sign operators*. MATLAB has more than one operator that uses equals signs (=) and they act differently. If you have done some programming, you are probably used to the idea that “=” is not used for comparison, but rather to *assign* the value from the right side of the equals sign to the variable on the left side. For example:

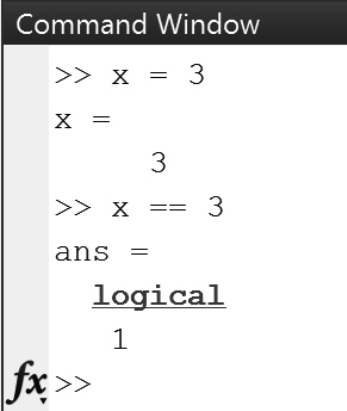
$$\text{index} = \text{index} + 1$$

is a perfectly legitimate *assignment statement* (for the new value, add 1 to the old value), but the two sides are certainly not mathematically equal.

Type the following lines in the Command Window and observe the results:

```
>> x = 3
>> x == 3
```

The results display in the Command Window similar to Figure 2.1.



```
Command Window
>> x = 3
x =
    3
>> x == 3
ans =
    logical
    1
fx>>
```

Figure 2.1

The command `x = 3` assigns the value 3 to the variable `x`. The command `x == 3` asks “Is `x` equal to 3?” and returns a value of 1, which means “true” since we just set it equal to 3. We’ll talk about this again, but the accidental use of = when you meant == is a common coding error.

Logical Operators

Here are the *logical operators*. These operators are used on logical values and result in a logical value.

Operator	Meaning
&&	and
	or
~	not

Logical operators might be new to you. Logical operators test for true or false and report either 1 (true) or 0 (false). In your code, you can either use 1 and 0 or the keywords `true` and `false`.

The not (~) Operator

The (~) operator (**not**) is straightforward: it reverses the logical value of the expression that it's applied to, so that true (1) becomes false (0) and false (0) becomes true (1). You can use either the symbol ~ or the command word, **not**.

Type this line into the Command Window for a demonstration:

```
>> ~ (x == 3)
```

Since x was previously set to 3, the value within the parentheses is evaluated is true (x is exactly equal to 3), then the ~ operator is applied to reverse it to false (0). The Command Window displays: **ans = logical 0**.

The and (&) Operator

The **& operator (and)** returns true only when *both* of the logical values it compares are true. This grid shows the results of X & Y for logical values X and Y:

X	Y	X and Y	Explanation
true (1)	true (1)	true (1)	X & Y is true if X is true and Y is true
true (1)	false (0)	false (0)	X & Y is false if X is true and Y is false
false (0)	true (1)	false (0)	X & Y is false if X is false and Y is true
false (0)	false (0)	false (0)	X & Y is false if X is false and Y is false

The or (|) Operator

The | **operator (or)** returns true when *either or both* of the logical values it compares are true. This grid shows the results of X | Y for logical values X and Y:

X	Y	X or Y	Explanation
true (1)	true (1)	true (1)	X Y is true if X is true and Y is true
true (1)	false (0)	true (1)	X Y is true if X is true and Y is false
false (0)	true (1)	true (1)	X Y is true if X is false and Y is true
false (0)	false (0)	false (0)	X Y is true if X is false and Y is false

The **&** and | operators compare expressions that return a logical value rather than comparing simple logical values. We'll see a lot of this when we talk about *if-then* and *while* statements.

For now, experiment a little by entering the next lines in the Command Window and noting the results.

Tip: The tilde (~) is usually at the very upper left of the numbers row on your keyboard.

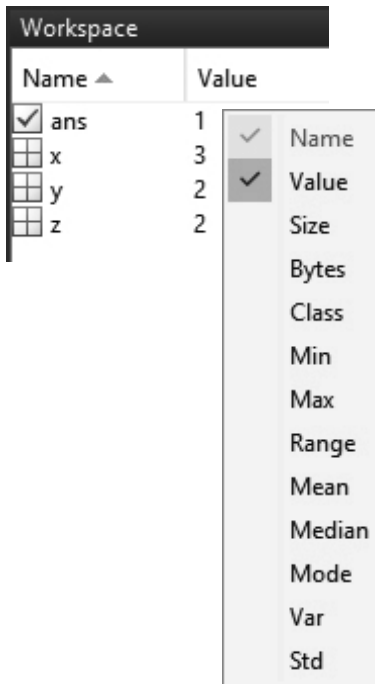
Tip: Want to get rid of a single variable from your Workspace without clearing them all? Use the **clear** command as usual, but specify the variable to delete. For example, **clear x**.

Tip: You can type **&** or you can use the command word **and**. You can type **|** or use the command word **or**.

Tip: Exclusive **or** means **A or B**, but not **A and B**. This operator is available as **XOR**.

Tip: The **or** operator has two versions that are not quite the same: **|** and **||**. The element-wise **or** operator is **|**. The **short-circuit or** operator is **||**. Short circuiting behavior for the operator means that once a condition is evaluated that causes the result to be determined, the remainder of the conditions are skipped. For example, if any element of the **or** operation is true, then the operation evaluates as true, so the remaining conditions can be skipped. The **&&** operator is the short-circuiting version for **and**.

Tip: Check the Workspace to see the value your variable has. If the Value column doesn't show, right-click to the left of the Name area and use the menu to select Value.



Tip: Remember the semicolon (;) suppresses the output from the entered line.

Tip: Functions generally have some input or "argument" that they act on. For example:
`islogical(x)`
 asks is the "argument" x of the type *logical*. If it is the answer is 1 (true), if not the answer is 0 (false). We write `islogical()` to show you that the function expects an argument to be entered in the parentheses when you use this function. You will be learning more about functions in the next tutorial.

Work through each side of the logical equations to understand why each line returns the logical 1 (true) or 0 (false) that it does.

Command Window entry	Result
<code>x = 3;</code>	assigns value of 3 to x
<code>y = 2;</code>	assigns value of 2 to y
<code>z = 2;</code>	assigns value of 2 to z
<code>x == y</code>	ans = logical 0 (false)
<code>y == z</code>	ans = logical 1 (true)
<code>x == y & y == z</code>	ans = logical 0 (false)
<code>x == y / y == z</code>	ans = logical 1 (true)
<code>~(x == y) & y == z</code>	ans = logical 1 (true)

The Logical Data Type

Relational and logical operators return a variable that is of type *logical*. This type has two values, 0 for false and 1 for true. Logical 1s and 0s are not the same as normal 1s and 0s, but the differences can be subtle. Watch out for situations where a statement expects a logical value. Though MATLAB will often automatically do any necessary conversion for you, a good programmer doesn't count on that. When a logical value is needed in your code, use the built-in MATLAB function `logical()` to create it. Type these lines into the Command Window and note the type that x and y are assigned.

```
>> x = 1;
>> y = logical(1);
>> class(x)
```

You see the result, `ans = 'double'`. Double is the default numerical data type in MATLAB. It stores values between $\pm 3.4 \times 10^{38}$.

```
>> class(y)
```

You see the result, `ans = 'logical'`.

To check whether a value is of type logical, use the MATLAB function `islogical()`. Type these lines into the Command Window:

```
>> islogical(x)
>> islogical(y)
```

While MATLAB returns the logical 1 and 0 in the Command Window, in your code you can also use the MATLAB constants `true` and `false`, which are the same as `logical(1)` and `logical(0)`.

Operator Precedence

Operator precedence is the order in which operations are executed in a statement that contains more than one operator. We just saw some examples above. The statements `x == y` and `y == z` were executed before the `&&` or `||` operator was used. This shows that the `==` relational operator has a higher precedence than the logical `&&` and `||` operators.

Operator precedence is vital to know, because the result of an operation can be wildly different than intended if precedence isn't understood and addressed. For instance, enter the following two lines into the Command Window and note the different results:

```
>> 3 + 2 * 6
>> (3 + 2) * 6
```

Multiplication has a higher precedence than addition, so the first line is equivalent to $3 + (2 * 6)$. Parentheses override precedence, so the second line first adds $3 + 2$ and then multiplies the result by 6. Someone who understands the precedence rules might use line one, but someone who assumes that everything is done left-to-right might use the first when they meant the second. Use parentheses to ensure the correct precedence!

Order of Operations

1. Parentheses
2. Logical negation (~), unary minus (-)
3. Multiplication, division
4. Addition, subtraction
5. Relational operators (<, <=, >, >=, ==, ~=)
6. Logical AND (&)
7. Logical OR (|)

In the case of a tie, operations are usually executed from left to right. There are many more operators and you can see the full list by searching for “operator precedence” in the MATLAB help. Scoff now, but if you do enough programming, you will know the full list by heart one day.

Variables

The concept of a *variable* is straightforward: a variable holds a value. It has a name, such as x or *city*, and that name is assigned a value, such as 3 or ‘Paris’.

Variable Naming Rules

As discussed previously, Matlab variable names are case-sensitive (*abc* and *Abc* are different variables), they can use letters, numbers, and the underscore character ('_') and must begin with a letter. They can be no longer than 63 characters!

Enter the following commands into the MATLAB Command Window and observe, in particular, the Workspace window. The Workspace window provides an at-a-glance list of all active variables you've defined, with their values. Some of these commands will cause an error. Don't be alarmed.

```
>> a = 5
>> A = 15
>> a + 3
>> A + 3
```

Tip: You may remember the mnemonic *My Dear Aunt Sally* (multiply divide add subtract) from elementary school for the order of arithmetic operations. MATLAB evaluates arithmetic operations in that same order.

Tip: A unary minus is what you probably know as a negative sign; for example, -3. “Unary” means that it only has one operand.

Tip: Technically, a variable's name is an alias for a location in memory where that value is stored. Sometimes that is helpful to remember.

Tip: When you provide an invalid variable name, MATLAB's error message doesn't specifically say that's the problem. Look for errors that talk about the value to the left of the equals sign, though there are others as well.

Tip: MATLAB ignores most white space, so $a=3$ and $a = 3$ are both valid, as is $a+3$ or $a + 3$.

Note: Did you notice that when you entered an invalid statement, MATLAB tried to help you out by displaying “Did you mean:” with a suggested alternative? If that alternative is correct, just hit [Enter]; otherwise, backspace the suggestion away.

```
>> A + a
>> win_win = 3
>> 2win = 3
>> win-win = 3
>> win.win = 3
```

Let’s review those last three entries:

Invalid variable name	Result	Reason
2win	Error: Unexpected MATLAB expression.	Variable begins with a number.
win-win	Error: The expression to the left of the equals sign is not a valid target for an assignment.	MATLAB thinks you’re trying to subtract a variable called win from itself.
win.win	A new structure called <i>win</i> is created with a member called <i>win</i> .	The variable name contains an invalid character, but it is a valid way to refer to a member of a structure (a more advanced data type that we’ll talk about in a later tutorial). MATLAB thinks you meant that and creates it.

Effective Variable Names

Strive to write code that is easily understood by another programmer. Right now, this probably isn’t an issue because you’re doing your work solo, but in the real world most programs are group efforts. Other programmers will review your code before adding it to a larger project, and later programmers will look at it to maintain and upgrade it over time, possibly when you’re no longer available to consult. One way to make your code understandable is to use descriptive variable names. In our examples so far, we’ve stuck to simple one-letter variables that don’t mean anything. Later, we’ll use variable names that explain the variable’s purpose and/or units.

For instance, say that you’re working on a simulation for a Mars lander and need to know the distance from the surface to the lander. You might name your variable *altitude*, and that’s not bad (better than *x* anyway), but if *altitude* = 4500, the question “4500 whats?” comes to mind. An American programmer might assume miles, while a Bulgarian programmer might assume kilometers. At least one of those two is going to see a spectacular crash. Perhaps *altitude_in_meters* is a better name, or *altitudeInMeters* if that’s your preferred style.

At the same time, don’t get carried away. You’re going to have to type that variable name over and over, so you’ll quickly learn to keep them succinct.

Tip: You can drag and drop variable names from the Workspace into the Command Window (and Editor also) to save having to type them.

Storing Numeric Values

All variables have a *class* or type, which is the sort of data it holds: numbers, strings, structures, etc. In this section, we'll talk about variables that hold numbers. If you remember from math class, integers are whole numbers (4, 288, 42), while floating-point numbers (also called “real” numbers) contain decimal values (3.1417, 6.125, 9.9).

Type the following into the Command Window:

```
>> x = 666
>> class(x)
```

Hmm. Even though you clearly entered an integer value, MATLAB created a variable of type *double*. Unless you explicitly specify the type of your numeric data, this is MATLAB's default behavior.

Back to the Command Window, enter these lines:

```
>> x = int8(666);
>> class(x)
```

Here, you specifically told MATLAB that you wanted to store the number 666 as an 8-bit integer called *x*. MATLAB has a function, `int8`, that converts the number (or variable) you provide as an *argument* (the value in parentheses) to an 8-bit integer. The `class` function reports back the class or type.

The number of bits is how many binary digits the value stores. For example, 8 bits can store a binary number such as 11011011. If the sign (positive or negative) takes up one binary digit, then the value that can be stored is reduced.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	1	1	1	1	1	1	1	
128+	64+	32+	16+	8+	4+	2+	1+	=255

There are several *integer types* for different levels of size and precision. Each comes in a *signed* and *unsigned* version. If you're only dealing with positive integers, using an unsigned type allows you to specify numbers twice as large.

Integer Types			
Type	Bits	Signed or Unsigned?	Range of values
int8	8	signed	-128 to 127
uint8	8	unsigned	0 to 255
int16	16	signed	-32,768 to 32,767
uint16	16	unsigned	0 to 65,535
int32	32	signed	-2,147,483,648 to 2,147,483,647 (Approx. -2.1 to 2.1 billion)
uint32	32	unsigned	0 to 4,294,967,295
int64	64	signed	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (Approx. -9.2 to 9.2 quintillion)
uint64	64	unsigned	0 to 18,446,744,073,709,551,615

Tip: The class (*variableName*) command is very useful, particularly in troubleshooting.

Tip: Notice the difference between `int8()` and `uint8()`. The unsigned `uint` can store a number twice as large because it doesn't have to store the sign.

Tip: Additional handy functions are `intmin()` and `intmax()`. Supply a type to one of those functions and MATLAB will tell you the minimum or maximum value that type can hold. For instance, `intmax('int32')`.

Tip: You can define a class and give it the constant property, but if you are contemplating that, you are way ahead of this introductory book.

Tip: *Inf* and *NaN* are not case-sensitive. *Inf* or *inf*, as you please. Imaginary numbers also have a special constant, *i*, used to represent the imaginary portion of the value.

```

Command Window
>> x = 128 + 2
x =
    130
>> y = int8(128) + 2
y =
    int8
    127
Yikes! 128+2
should not
equal 127
>> z = int8(128)
z =
    int8
    127
Yikes! 128
should not
equal 127

```

Floating-point types can represent numbers so big that they make an *int64* look like something you could count on your fingers. This is possible because part of their stored value is an exponent. They come in two varieties: single and double.

Floating Point Types			
Type	Bits	Signed or Unsigned?	Range of values
single	32	signed	-1.79×10^{38} to 1.79×10^{38}
double	64	signed	-1.79×10^{308} to 1.79×10^{308}

Constants

A *constant* is a value that, once assigned, cannot be changed later. MATLAB doesn't have a straightforward way for you to declare your own constants, but it does provide a handful of its own.

There are two special floating-point values to be aware of. You'll see these values as the result of an operation. Type the following statements into the Command Window and watch the result:

```
>> 3/0
>> inf-inf
```

In many programming languages, an error results if you attempt to divide by zero, but not MATLAB. It returns the constant value *Inf* for infinity, or *-Inf* for negative infinity. *Inf* can be the result of any operation that results in enormously large or small numbers.

If you've tried to perform an operation that is not mathematically defined, you'll see the constant value *NaN*. This means "not a number." This isn't quite a constant by any standard definition, but it isn't quite anything else either. In some matrix cases it's used as a placeholder.

We'll show you how to make use of these values when we discuss *if* statements in a later tutorial. The other constant of note is *pi*. You've already used it in an earlier tutorial. Try this in the Command Window:

```
>> format long
>> pi
>> pi + 7
```

Remember, *format long* tells the Command Window to display numbers with more digits. This is for display purposes only. In calculations, the full stored number is used.

Exceeding a Type's Range

Now for a chilling demonstration of where a type can cause things to go terribly wrong. Enter the following into the Command Window:

```
>> x = 128 + 2
>> y = int8(128) + 2
>> z = int8(128)
```

Well, that's not good! Notice in the Workspace that z has a value of 127, not the 128 you'd specified. If you provide a number larger or smaller than the type can handle, MATLAB returns the largest or smallest number that type can provide, with no error. 127 is as high as an int8 can go. This can lead to miscalculations that can be hard to track down. Be aware of the magnitude of your data versus your data types.

You're probably wondering why we don't just use doubles all the time. You probably can – no harm, no foul. However, if you're working with Big Data, you may work with data sets that contain billions of numbers or more. Code that is optimized for its expected data can prevent memory overruns and significantly affect run time. It literally can be the difference between getting results in minutes and getting them in years.

Numerical Functions

As you can imagine, there are almost more built-in MATLAB functions for manipulating numbers than there are stars in the sky (oh, perhaps a bit of hyperbole there). Here are some functions you might find useful.

Function	Action
<code>ceil</code>	Rounds toward positive infinity
<code>floor</code>	Rounds toward negative infinity
<code>fix</code>	Rounds toward zero
<code>round</code>	Rounds toward the nearest whole number
<code>mod (a, b)</code>	Returns the modulus of a divided by b . Retains the sign of the divisor. Example: <code>mod(10, -7)</code> returns -4.
<code>rem (a, b)</code>	Returns the remainder in a division operation. Retains the sign of the dividend. Example: <code>rem(10, -7)</code> returns 3.

Type the following commands into the Command Window and note the output that results.

Command Window Entry	Result
<code>>> ceil(3.4)</code>	<code>ans = 4</code>
<code>>> ceil(-3.4)</code>	<code>ans = -3</code>
<code>>> floor(3.4)</code>	<code>ans = 3</code>
<code>>> floor(-3.4)</code>	<code>ans = -4</code>
<code>>> fix(3.4)</code>	<code>ans = 3</code>
<code>>> fix(-3.4)</code>	<code>ans = -3</code>
<code>>> round(3.4)</code>	<code>ans = 3</code>
<code>>> round(-3.4)</code>	<code>ans = -3</code>
<code>>> mod(5, 2)</code>	<code>ans = 1</code>
<code>>> rem(5, 2)</code>	<code>ans = 1</code>
<code>>> mod(5, -2)</code>	<code>ans = -1</code>
<code>>> rem(5, -2)</code>	<code>ans = 1</code>
<code>>> mod(-5, 2)</code>	<code>ans = 1</code>
<code>>> rem(-5, 2)</code>	<code>ans = -1</code>

Tip: Functions generally have some input or "argument" that they act on. For example:

`ceil(x)`

where x is the "argument" or input that can be provided to the function. You provide the input argument inside parentheses after the function name. Use MATLAB help to look up these details when you use a function. When we write `ceil()` it is to remind you that you will provide an input when using it. Some functions may accept multiple arguments, which may also be left out depending on the use.

Tip: Don't forget the parentheses! Without them the value 3.4 is assumed to be three text characters: a 3, a period, and a 4.

```
>> ceil(3.4)
ans = 4
>> class(3.4)
ans = 'double'
>> ceil 3.4
ans = 51 46 52
>> class 3.4
ans = 'char'
```

There are also conversion functions for every numeric type. For example `int16(43)` converts 43 (a double by default) into a 16-bit integer. One word of warning with these functions: beware of losing precision and exceeding type ranges in the conversion, particularly when converting unsigned types to signed types.

Enter the following commands in the Command Window

```
>> x = uint8(255)
```

```
>> y = int8(x)
```

An `int8` can't hold the number 255, so the number becomes 127. Probably not what you wanted!

Of course, standard *trigonometric functions* are available as well, each in two versions: one which returns the value in degrees and one which returns the value in radians.

*On your own, browse the MATLAB help for **Elementary Math** (Figure 2.2) then select **Trigonometry** from the results to see the full list of trig functions. Explore the other functions, too.*

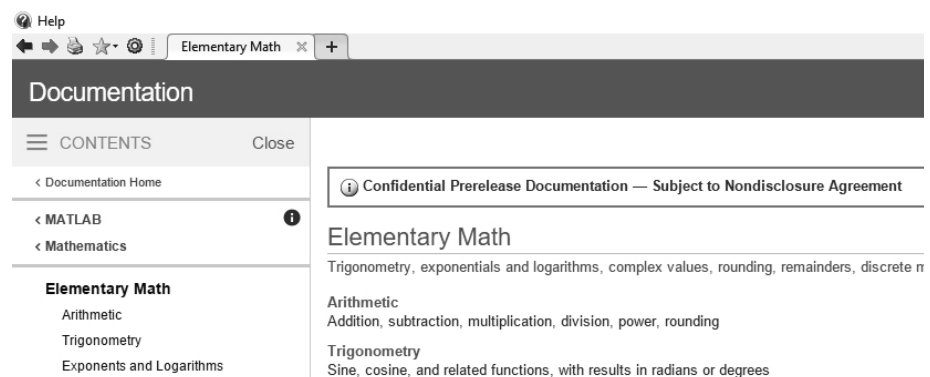


Figure 2.2

Strings

A *string* type holds text. An example of a string is this very sentence. A string can hold any valid character. “123456” can be a string. “< <= > >=” is another. To discuss strings, we need to talk about *arrays*. We’ll go into arrays and matrices in more depth in a later tutorial, so here we’ll just cover the concepts you need right now.

Array Basics

An *array* is a collection of data, all of the same type. Each item in the array has an *index* by which it can be accessed. Consider the phrase “Carpe diem.” Each letter, space, and punctuation mark in that phrase is a character. If you store that string as an array of characters, this is how it is stored:

Index	1	2	3	4	5	6	7	8	9	10	11
Character	C	a	r	p	e		d	i	e	m	.

This a one-dimensional array of 11 characters (1 x 11).

Type the following lines into the Command Window:

```
>> st = 'Carpe diem.'
>> size(st)
>> length(st)
>> st(4)
>> st(4) = '9'
>> st(4) = 33
```

The function `size()` tells you that it is a 1 x 11 array, while the function `length()` simply tells you that the array contains 11 elements. You accessed a specific element in the array with `st(4)` and overwrote that character with `st(4) = '9'`.

The statement `st(4) = 33` had the surprising result of overwriting that element with an exclamation point. This is because all elements in an array must be of the same type – characters, strings, numbers, and so forth. The variable `st` is an array of characters, so you can't overwrite an element with a numerical value. Instead, MATLAB assumed that you wanted *ASCII character 33*, which is an exclamation point. The American Standard Code for Information Interchange (ASCII) set is a collection of characters, each with its own numerical code, commonly used in computing to represent letters and symbols (including numbers). The standard ASCII set includes all the characters you can type on a US keyboard, each encoded in 7-bits. Extended ASCII sets exist for characters such as accented, Asian, Arabic, Cyrillic, and Hebrew letters, scientific and other specialized symbols, and many more. The following table shows some of the ASCII character set. You can find the entire sequence on the web.

Decimal	7-bit Binary	ASCII Character (notes)
0	0000000	NUL '\0'
1	0000001	SOH (start of heading)
2	0000010	STX (start of text)
3	0000011	ETX (end of text)
4	0000100	EOT (end of transmission)
10	0001010	LF '\n' (new line)
32	0100000	SPACE

33	!	57	9	68	D	120	x
34	"	58	:	90	Z	121	y
35	#	59	;	91	[122	z
36	\$	60	<	92	\ '\'	123	{
37	%	61	=	93]	124	
38	&	62	>	94	^	125	}
39	'	63	?	95	_	126	~
48	0	64	@	96	`	127	DEL
49	1	65	A	97	a		
50	2	66	B	98	b		
		67	C	99	c		

```
Command Window
>> st = 'Carpe diem.'
st =
    'Carpe diem.'
>> size(st)
ans =
     1    11
>> length(st)
ans =
    11
>> st(4) = '9'
st =
    'Car9e diem.'
>> st(4) = 33
st =
    'Car!e diem.'
```

Tip: The `size()` function returns information as rows first, then columns.

The Two String Types

There are two types of strings: a *character array* and a *string array*. Strings as character arrays are entered by surrounding the text in single quotes ('the'). Strings as string arrays are entered by surrounding the text in double quotes ("the"), resulting in an array with one element.

String arrays are new to MATLAB 2017. They are more memory-efficient and have a number of new functions that can be used with them, as well as using many of the familiar character array functions. Which you choose in any situation will depend on your programming needs.

Enter the following lines in the Command Window for a whirlwind tour of some differences between character arrays and string arrays:

```
>> clc
>> sc = 'This is a character array'
>> st = "This is a string array"
>> class(sc)
>> class(st)
>> size(sc)
>> length(sc)
>> size(st)
>> length(st)
>> sc(1)
>> st(1)
>> sc(2) = 'H';
>> sc
>> st(2) = "And this is the second string in the array";
>> st
```

The major difference to remember is that a string as character array is a 1 x *length* array of individual characters, while a string as a string array is a 1 x 1 array that contains a single string.

String and Character Functions

MATLAB provides many functions to use with strings and characters. Here's a list of some of the most useful. For a complete list and usage particulars, search for "Characters and Strings" in the MATLAB help.

Function	Use with...	Action	Return type
<code>ischar</code>	any variable type	Is this a character array?	logical
<code>isstring</code>	any variable type	Is this a string array?	logical
<code>isletter</code>	character arrays	Which characters are letters?	logical array

isspace	character arrays	Which characters are spaces?	logical array
length	character arrays	How long is the array?	integer
strlength	character or string arrays	How long is each string in the array?	integer array
lower	character or string arrays	Convert all letters in the string to lower-case	string
upper	character or string arrays	Convert all letters in the string to upper-case	string
strcmp	character or string arrays	Compare two strings for equality. Use this instead of =.	logical
strcat	character or string arrays	Add one string to the end of another	string
startsWith	character or string arrays	Does the string start with a particular substring?	logical
endsWith	character or string arrays	Does the string end with a particular substring?	logical
contains	character or string arrays	Does the string contain a particular substring?	logical
strfind	character or string arrays	What is the starting position of a substring?	integer
strtrim	character or string arrays	Remove any spaces from the beginning and end of the string	string
replace	character or string arrays	Replace a substring with a new substring	string
erase	character or string arrays	Remove a substring from the string	string
insertBefore	character or string arrays	Add a new substring before a specific point in the string	string
insertAfter	character or string arrays	Add a new substring after a specific point in the string	string
split	character or string arrays	Split the string into two or more substrings	strings

Type the following statements into the Command Window, noting the form of the functions and the values they return:

```
>> charArray = 'This is a character array.'
>> stringArray = "This is a string array."
>> ischar(charArray)
>> ischar(stringArray)
>> isstring(stringArray)
```

The results you see in the Command Window are just as you'd expect.

Tip: Use the MATLAB help to look up the required format for these functions when necessary. For example, the syntax for the `insertBefore` is: `newStr = insertBefore(str,endStr,newText)`. In this case, `str`, `endStr`, `newText` are the inputs for the function.

Syntax is like the exact recipe you must follow for MATLAB to understand the command input. We assume you can look these up in the help when you need them. What kind of person memorizes every instance of every function anyway?

Tip: Several of the string functions have a separate version that ignores case. For instance, `strcmp` (case-sensitive) and `strcmpi` (case-insensitive).

Tip: Keep in mind that functions and variable names are case sensitive. The name "charArray" is not the same as "Chararray".

Tip: In functions such as `contains` and `strfind` that take a substring as an argument, you can use either a character array (single quotes) or a string array (double quotes). MATLAB knows what you mean and takes care of any necessary conversions behind the scenes.

Tip: Use `clc` to clear the Command Window on your own when things get a bit too messy.

Tip: Use `clear` when you want to empty the variables from the Workspace. This will delete the values if they are not saved to a file, so make sure this is what you want before clearing the Workspace. You do not need to use `clear` to set the variable to a different value, but this also means that you can accidentally overwrite a value when you reuse a variable name.

Now let's get into trickier stuff. Type the following line into the Command Window:

```
>> isletter(charArray)
```

Interesting. This function returns a numerical array of the same size as the character array. In each position is a logical value telling you whether that character is a letter (a-z, A-Z), 1 for true, 0 for false.

Give isspace a try on your own.

Enter this command into the Command Window:

```
>> upper(stringArray)
```

Notice that it shouted "THIS IS A STRING ARRAY" back at you in the Command Window, but the variable `stringArray` in the Workspace hasn't changed. This is because we didn't assign the returned value back to a variable. Try this instead:

```
>> stringArray = upper(stringArray)
```

Lesson: Just entering the function does not store the value for later use. If you want to keep the results, use the equals sign and store the result in a variable (which you can think of as a named storage location)

Now let's talk about what's arguably the most important string function: `strcmp`. Type the following into the Command Window:

```
>> text1 = 'Four score and seven years ago'
```

```
>> text2 = '87 years ago'
```

```
>> text3 = 'Four score and seven years ago'
```

```
>> strcmp(text1, text2)
```

```
>> strcmp(text1, text3)
```

```
>> text1 == text3
```

Yipes. And that's why you need `strcmp`. The relational equality operator (`==`) looks at the string as an array, compares the individual characters, and returns an array of logical results for each index. The `strcmp` function answers the question you're really asking – "Do these two variables contain the same text?"

The situation is a little cloudier with string arrays, but the general advice remains the same: whenever you want to compare two strings, use `strcmp`.

Enter the following commands into the Command Window:

```
>> contains(text1, 'seven')
```

```
>> strfind(text1, 'seven')
```

The first command tells you that yes, the substring 'seven' is in `text1` somewhere. The second command tells you where it is – the character 's' of 'seven' is at index 16.

Finally, type the following line into the Command Window:

```
>> insertBefore(text1, strfind(text1, 'seven'), 'fifty-')
```


This command demonstrates two important points. First, a function can be used like a value within another function. The command is equivalent to the following two commands. We'll discuss the pros and cons of both forms when we talk about functions in a later tutorial.

```
index = strfind(text1, 'seven')
insertBefore(text1, index, 'fifty-')
```

The second point the original command made was about nested parentheses. This can occur in all kinds of situations and is another common source of simple coding errors. MATLAB helps you to avoid this by highlighting the matching opening parenthesis when you type a closing parenthesis. Also, if you wind up with a mismatched number of opening and closing parentheses, the error message that results will usually point you to that as the issue.

MATLAB and Type Flexibility

MATLAB is not a strongly typed language. That means that you can perform assignments and operations on variables of wildly different types, usually without causing an error. This can be a blessing in that you can usually avoid explicitly defining the type for each variable, but it can be a curse when the result of, say, adding a number to a string, is puzzling (though it can all be explained).

Type the following commands, and note the results.

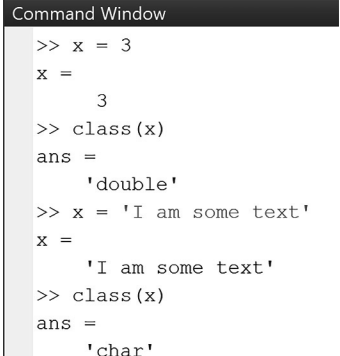
```
>> x = 3
>> class(x)
>> x = 'I am some text'
>> class(x)
```

The variable becomes the type of whatever you assign to it, regardless of what it held before. Not many languages allow that.

Now enter these commands in the Command Window. Use the up arrow on your keyboard and select from the history to save yourself some typing.

```
>> x = 3
>> class(x)
>> y = single(5)
>> z = x + y
>> class(z)
>> x = int16(5) + int8(3)
```

The last line demonstrates that this doesn't work with integer types, only floating point types. See Figure 2.3.



```
Command Window
>> x = 3
x =
    3
>> class(x)
ans =
    'double'
>> x = 'I am some text'
x =
    'I am some text'
>> class(x)
ans =
    'char'
```

```

>> x = 3
x =
    3
>> class(x)
ans =
    'double'
>> y = single(5)
y =
    single
     5
>> z = x + y
z =
    single
     8
>> class(z)
ans =
    'single'
>> x = int16(5) + int8(3)
Error using +
Integers can only be combined with integers of the same class, or scalar doubles.

```

Figure 2.3 Error Resulting from Conflict of Integer Types

Finally, enter the following into the Command Window:

```

>> x = 3
>> y = 'I am a string'
>> z = x + y
>> class(z)
>> char(z)

```

```

>> z = x + y
z =
    76    35   100   112    35   100    35   118   119
>> class(z)
ans =
    'double'
>> char(z)
ans =
    'L#dp#d#vwulqj'

```

Figure 2.4 Adding Numbers and Letters

MATLAB looked at the string as a character array, added 3 to each character's ASCII value, and returned a numerical array of those values as numbers instead of the characters. The final line, `char(z)`, casts the array back to a character array-type string based on the ASCII values. Potential uses in encoding, perhaps – a simple alphabet shift. MATLAB allows a lot of flexibility, and you can use this in creative ways to accomplish interesting solutions. Just make sure you're doing it on purpose! Use the functions available to set the data type to prevent problems.

Now that you have been introduced to types and strings, let's take a look at an example of importing data into MATLAB and some options for data types.

Genetic Data Example

Humans typically have 23 pairs of chromosomes located in each of their cells. One of the shorter ones, chromosome 22, has more than 50

million base pair building blocks. Base pair data is encoded by one of four letters: G, C, A, and T (guanine, cytosine, adenine, and thymine). A variation in the expected code, or a missing code, may result in health problems. Customizing medicine for an individual's specific genetic variation is a hot topic in improved therapies.

The data file, *Chrome22- HG00096.vcf*, included in our download, is a Variant Call Format (vcf) file. It contains information about positions in the genome and genotype information for each sample. It is a text file that starts with a header section followed by this required data on each line:

CHROM: chromosome number. The data file you will use only has chromosome 22 data.

POS: position. The number that represents the location of the gene on the chromosome. If you look at the .vcf file you will see that the range is 1 up to about 1.2 million. (This data should be a number with no decimal portion.)

ID: identifier. The dbSNP variant is given as an rs number(s). No identifier should be present in more than one data record. If there is no identifier available, then the value '.' should be used. (This should be a string with no white-space or semi-colons permitted.)

REF: reference base(s). Each base allele must be one of A,C,G,T, or N (meaning aNy of these.) This data is not case-sensitive. More than one base letter is allowable. The value in the POS field gives the position of the first base in the string. Gene variations may have insertions or deletions in which either the REF or the ALT alleles are null/empty. POS denotes the coordinate of the base preceding the polymorphism (a big word meaning there can be several different forms, one of the things you would be searching for in genetic variation). (This required data should be a string.)

ALT: alternate base(s). Similar to REF, but for alternate non-reference alleles. These alleles do not have to be called in any of the samples. Options are A,C,G,T,N, or *. The '*' is used to indicate that the allele is missing due to an upstream deletion.

QUAL: quality. Quality score based on the Phred-scale. Our data all has 100 for the quality. You can read more about this topic on your own or in a statistics class.

FILTER: filter status. "PASS" indicates the quality score is passing.

INFO: additional information. Keys such as AA (ancestral allele), AC (allele count in genotypes), and AF (allele frequency) are often used, but others are permitted.

Any other columns in the data file are optional. So now you know a bit about .vcf files, so let's get to importing one into MATLAB.

Importing Data into MATLAB

MATLAB has a handy feature for importing data using the Import Data tool. You can also use `readtable`, `csvread`, `dlmread`, `textscan`, `imread`, and other command entries. For this example, we will use the Import Data tool from the Home tab of the ribbon.

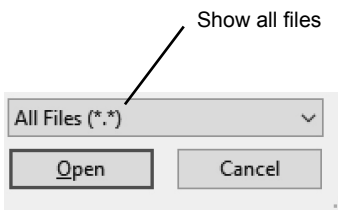
Tip: *Don't worry, nobody is going to test you on your knowledge of genetics or vcf files here. We are just going to import some data from a large file and check out some things about data types. It's fun to look at real data and who knows, maybe you will go find a cure for cancer or schizophrenia in your spare time.*

You can read more about rs numbers in genetics at en.wikipedia.org/wiki/DbSNP and many other places. The variation viewer at www.ncbi.nlm.nih.gov/variation/view is a great tool for exploring genetic information.

Tip: *Can't remember which letters are in the genetic code? Make up a mnemonic, like GAG-A-CAT. Its all Gs, As, Cs and Ts. The N is for aNy.*



WARNING: The file you will import from has more than 1 million rows in it. If you are not working on a fast computer system, use the smaller data file instead.



Click: **Import Data** from the ribbon Home tab

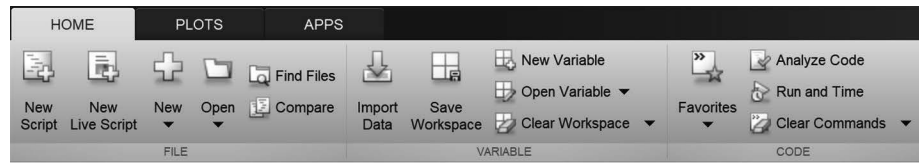


Figure 2.5 Ribbon Home Tab with Import Data Tool

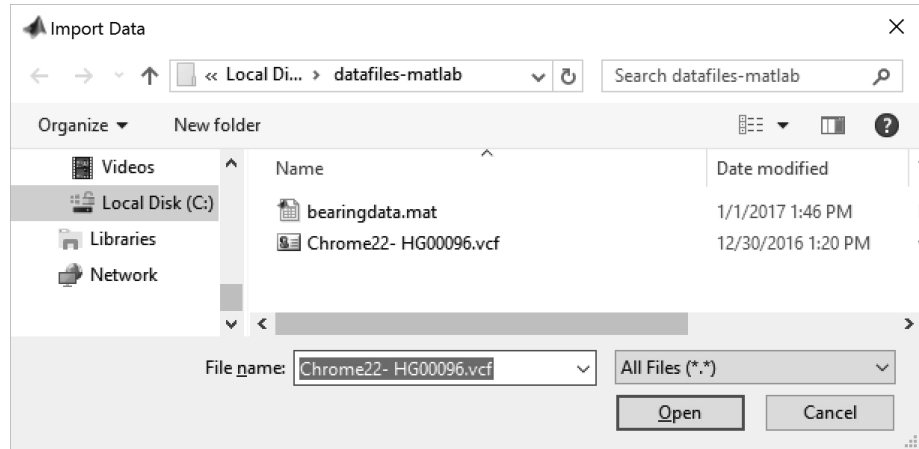


Figure 2.6 File Browser

On your own, change the file type to All Files (*.*) at the bottom right of the browser window, to show the .vcf file.

Use the file browser to locate and open the data file **Chrome22-HG00096.vcf** OR use the smaller data file named **Chrome22-small.vcf** if your system will not handle larger files easily.

The import data window opens on your screen similar to Figure 2.7. It shows a view of the data similar to spreadsheets you may have used.

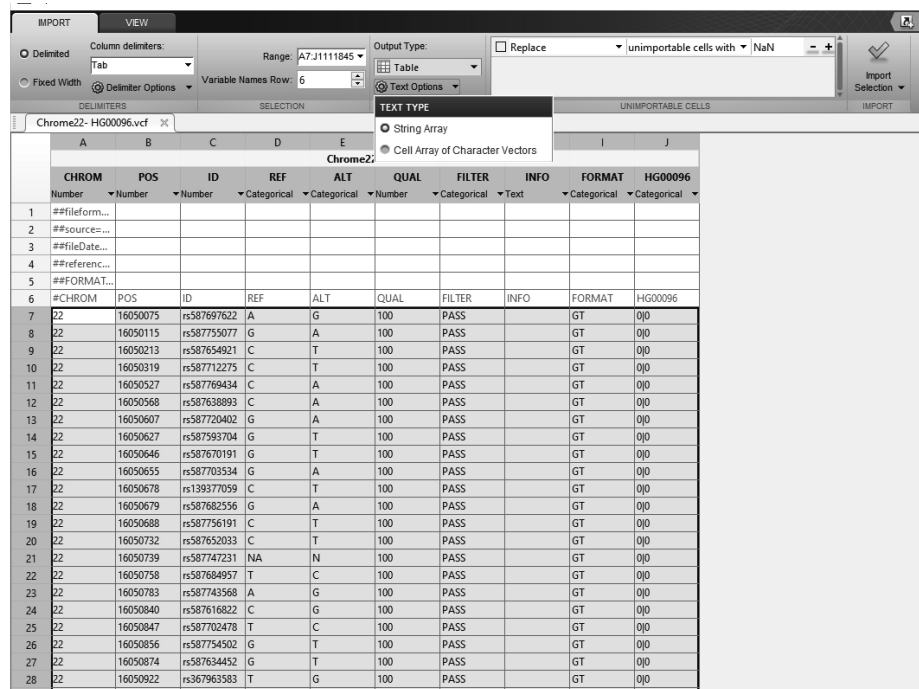


Figure 2.7 Data to Be Imported

Text Options

Notice under the drop down for Text Options, there are options to create a String Array or a Cell Array of Character Vectors, similar to entering strings using either double-quotes ("string array") or single-quotes ('character array').

Leave the Text Type set to String Array.

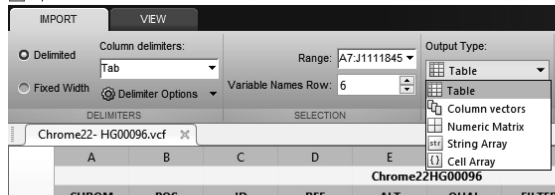


Figure 2.8 Output File Options

Output Type

Our data will work best as a table which has rows and columns, but column vectors, numeric matrix, and our old friend the string array and cell array are available options for data that would be suited to those.

*Leave **Table** set as the **Output Type**.*

Delimiters

Notice there are also options that allow you to select whether your data has delimiters, such as commas or tabs, between the entries. Our data does, so leave this set to Tab delimited. The other option is to have a fixed width of characters for each entry. This is useful when each entry is always the same length, like a list of numbers or when the data itself contains commas or tabs and you don't want the columns based on those.

Variable Names Row

This option lets you select a heading row to generate the variable names used for the columns of the table. We will use row 6, which contains headings CHROM, POS, ID, ALT, QUAL, FILTER, etc.

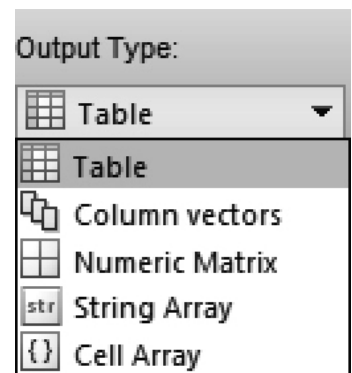
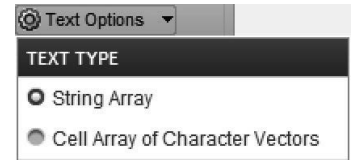
*On your own set **Variable Names Row** to row **6**.*

Range

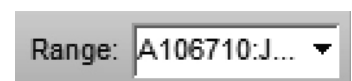
You don't have to use all of the data in your giant file. The COMT gene is associated allele variants such as rs4680, rs737865, and rs165599 which may be associated with response to antidepressants among other things.

The COMT gene is located starting at 19,941,740 up to 19,969,975, so we will extract only a section of the data around this gene to keep the imported file smaller, just for this project. We've already figured out which rows are near this region to make this easier.

*On your own, set the **Range** to **A106710:J107844**.*



Tip: If you used the short file, then select the entire range of the data.



Tip: If you used the short file, then the variable names will be in row 1.



Tip: The *Import Data* tool generates a script that you can edit and run to import files. This is a handy way to import multiple files with similar data quickly.

When the file is imported, only those rows (106710 to 107844) and columns (A-J) will be used in the MATLAB table.

Import Selection Options

Now you are ready to import the data into MATLAB, where you will see it as a new variable in the Workspace. Notice that the Import Selection button has a small triangle at the bottom that you can click to expand to see more options.

We will use Import Data, but you can also use this to generate a script, which you used in an earlier tutorial, or a function, which you will learn about in a later tutorial. This is very handy if you must convert multiple data files. We are just doing this once, so...

Click: Import Selection

The data is imported to the MATLAB table and appears in the Workspace as shown in Figure 2.9.

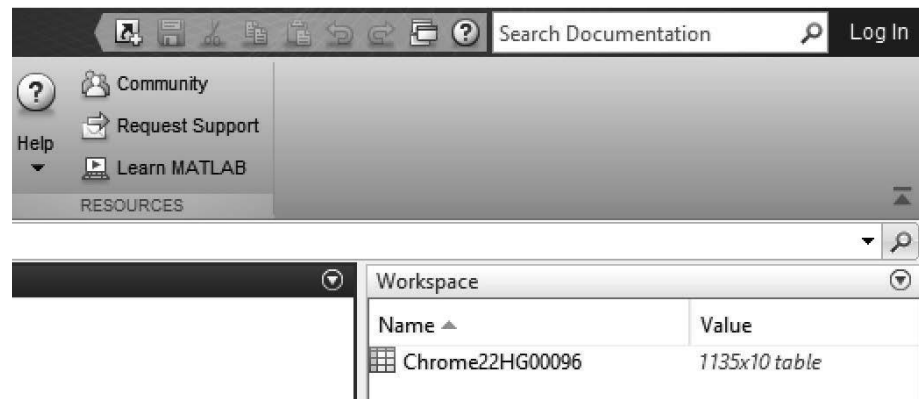


Figure 2.9 Table Imported in Workspace

Double-click: Chrome22HG00096 in the Workspace list

The table of imported data opens and the Command Window shows below it. Now you can enter MATLAB commands to interact with the data.

On your own, click and drag on the divider to resize the Command Window larger.

Enter the following command at the prompt to sort the data by 'ID', the variable heading for row 3.

```
>> tblIDs = sortrows(Chrome22HG00096,'ID')
```

The table information displays in the Command Window, sorted by the ID column values. Notice the new variable, tblIDs, in the Workspace.

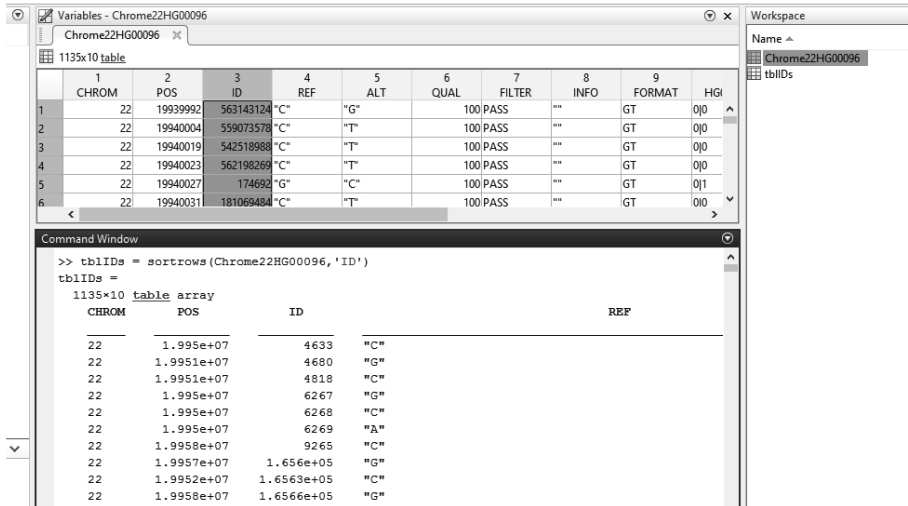


Figure 2.10 Sorted Data in Command Window

Remember that unless you save the imported file will not be saved when you either clear your variable list or exit MATLAB. Here's another way to save a file.

*Right-click: **Chrome22HG00096** in the Workspace.*

*Use the context menu to select **Save As...***

Browse to your Work folder and save the file with the name **Chrome22data.mat** as shown in Figure 2.11.

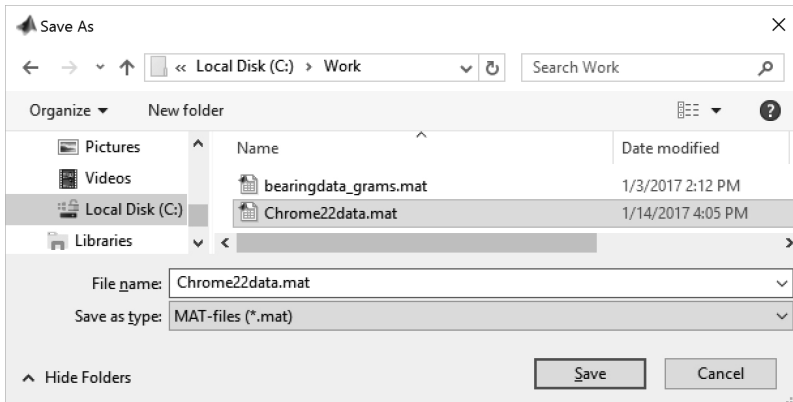
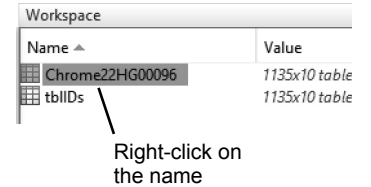


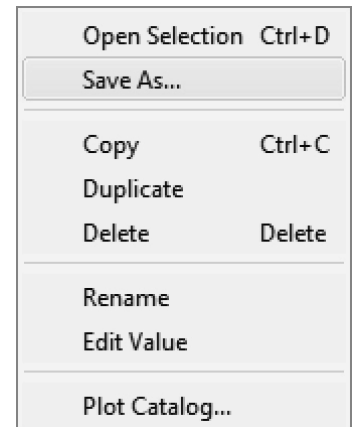
Figure 2.11 Saving the File

Congrats! You have finished Tutorial 2.

Tip: Check out the interface and notice some other ways you could sort this table.



Right-click on the name



Key Terms

<i>argument</i>	<i>equal sign operators</i>	<i>operator precedence</i>
<i>arithmetic operators</i>	<i>floating-point types</i>	<i>pi</i>
<i>array</i>	<i>index</i>	<i>relational operators</i>
<i>ascii character</i>	<i>Inf</i>	<i>signed</i>
<i>assignment statement</i>	<i>-Inf</i>	<i>string</i>
<i>character array</i>	<i>integer types</i>	<i>string array</i>
<i>class</i>	<i>logical operators</i>	<i>trigonometric functions</i>
<i>constant</i>	<i>logical type</i>	<i>unsigned</i>
<i>double</i>	<i>NaN</i>	<i>variable</i>

Key Commands

&&	format	length	startsWith
	imread	logical	strcat
and	insertAfter	lower	strcmp
ceil	insertBefore	mod	strfind
class	int16	not	strlength
contains	int8	or	strtrim
csvread	ischar	readtable	textscan
dlmread	isletter	rem	upper
endsWith	islogical	replace	xor
erase	isspace	round	
fix	isstring	size	
floor	length	split	


Exercises



Exercise 2.1

Based on operator precedence, predict the result of the calculation below without using Matlab. Once you think you know the answer, enter the statement in the Command Window. Do your answers agree?

$$5 + 2 * 8 / 2 - (3 * 2 + 10) / -1$$

 **Exercise 2.2**


Predict the value of z after the following statements:

$$x = 1$$

$$y = 5$$

$$z = \sim(x < y \parallel \sim(y < x) \&\& \text{islogical}(x))$$


Enter the commands above into the Command Window. Was your prediction correct? You have a 50:50 chance of getting it right, so make sure that if your prediction was correct, it was correct for the right reasons.

 **Exercise 2.3**

Predict both the value and the data type of x in this equation:

$$x = 55 + \text{uint32}(-22) + \text{pi}$$

Enter the command in the Command Window. Were you right? What does this equation demonstrate about variable types?

 **Exercise 2.4**

Predict the value of each of these commands:

$$\text{int8}(\text{ceil}(127.1))$$

$$\text{int8}(\text{floor}(127.9))$$

$$\text{int8}(\text{fix}(127.5))$$

$$\text{int8}(\text{round}(127.7))$$

$$\text{int8}(\text{ceil}(\text{rem}(-528.6, 200)))$$

Enter the commands in the Command Window and compare each result to your prediction. If you were mistaken on any, enter them in parts from the center outward, such as $\text{ceil}(127.1)$ then $\text{int8}(\text{ceil}(127.1))$, to see where the number did something you didn't expect.

 **Exercise 2.5**

Predict what you expect to see (generally, not precisely) as a result of the last two lines of this set of commands:

$$x = \text{'small kittens'}$$

$$y = \text{"small kittens"}$$

$$3 + x$$

$$3 + y$$

**Exercise 2.6**

List the data type you would be likely to use to store the following information in a Matlab program. Briefly explain your answers.

- a. firstname, lastname, and middle initial of people for a list which you may sort later by either first name or last name. Some people may not have middle initials.
- b. average homework scores on a test.
- c. seat numbers in a 800 seat theater.
- d. the number of base pairs in the humane genome.
- e. words for an automated poetry generating algorithm.
- f. comment entries from web site users.
- g. a list of answers for a true/false style test.
- h. a list of answers for a multiple choice test with four choices.

**Exercise 2.7**

Use Matlab to calculate answers to the following: Carpet cost \$1.69 / ft² and comes on a roll that is 12 feet wide. For each room, calculate the cost of the carpet and percentage of wasted carpet:

- a. 8' x 11'
- b. 14' x 9'
- c. 12 x 8'-6"
- d. 18' x 13'

**Exercise 2.8**

The Great Pacific Garbage Patch (GPGP) is a zone of plastic debris accumulated in the ocean between California and Hawaii. A ship and aircraft survey predicted about 79,000 tons of plastic are floating inside an area of 1.6 million km². This survey estimates 75% of the mass was from pieces larger than 5 cm, microplastics accounted for only 8% of the total mass but 94% of the estimated 1.8 trillion pieces of floating plastic in the GPGP. Use Matlab to estimate the following:

- a. the average weight of a piece of plastic debris in the GPGP.
- b. the average number of pieces of plastic debris in one square mile of the GPGP.
- c. the volume in square yards of the GPGP if it were condensed into one solid mass; base your estimate on a density of plastic at 1.20 grams per cubic centimeter.