This default configuration can be altered by clicking the down arrow at the top right corner of each window. This will show a menu of options (different for each window), including, for example, closing that particular window and undocking that window. Once undocked, bringing up the menu and then clicking on the curled arrow pointing to the lower right will dock the window again. To make any of these windows the active window, click the mouse in it. By default the active window is the Command Window.

Beginning with Version 2012b, the Desktop now has a ***toolstrip***. By default, three tabs are shown ("HOME," "PLOTS," and "APPS"), although another, "SHORTCUTS," can be added.

Under the "HOME" tab there are many useful features, which are divided into functional sections "FILE," "VARIABLE," "CODE," "ENVIRONMENT," and "RESOURCES" (these labels can be seen on the very bottom of the grey toolstrip area). For example, under "ENVIRONMENT," hitting the down arrow under Layout allows for customization of the windows within the Desktop Environment including adding the SHORTCUTS tab. Other toolstrip features will be introduced in later chapters when the relevant material is explained.

1.3 VARIABLES AND ASSIGNMENT STATEMENTS

To store a value in a MATLAB session, or in a program, a ***variable*** is used. The Workspace Window shows variables that have been created and their values. One easy way to create a variable is to use an ***assignment statement***. The format of an assignment statement is

```
variablename = expression
```

The variable is always on the left, followed by the = symbol, which is the ***assignment operator*** (unlike in mathematics, the single equal sign does *not* mean equality), followed by an expression. The expression is evaluated and then that value is stored in the variable. Here is an example of how it would appear in the Command Window:

```
>> mynum = 6
mynum =
     6
>>
```

Here, the ***user*** (the person working in MATLAB) typed "mynum = 6" at the prompt, and MATLAB stored the integer 6 in the variable called *mynum*, and then displayed the result followed by the prompt again. As the equal sign is the assignment operator, and does not mean equality, the statement should be read as "mynum gets the value of 6" (*not* "mynum equals 6").

Note that the variable name must always be on the left, and the expression on the right. An error will occur if these are reversed.

```
>> 6 = mynum
   6 = mynum
     |
 Error: The expression to the left of the equals sign is not a valid target
 for an assignment.
 >>
```

Putting a semicolon at the end of a statement suppresses the output. For example,

```
>> res = 9 – 2;
>>
```

This would assign the result of the expression on the right side, the value 7, to the variable *res*; it just does not show that result. Instead, another prompt appears immediately. However, at this point in the Workspace Window both the variables *mynum* and *res* and their values can be seen.

The spaces in a statement or expression do not affect the result, but make it easier to read. The following statement, which has no spaces, would accomplish exactly the same result as the previous statement:

Note
In the remainder of the text, the prompt that appears after the result will not be shown.

```
>> res = 9 – 2;
```

MATLAB uses a default variable named *ans* if an expression is typed at the prompt and it is not assigned to a variable. For example, the result of the expression 6 + 3 is stored in the variable *ans*:

```
>> 6 + 3
ans =
      9
```

This default variable, *ans*, is reused any time when only an expression, not an assignment statement, is typed at the prompt. Note that it is not a good idea to use *ans* as a name yourself or in expressions.

A shortcut for retyping commands is to hit the up arrow ↑, which will go back to the previously typed command(s). For example, if you decide to assign the result of the expression 6 + 3 to a variable named *result* instead of using the default variable *ans*, you could hit the up arrow and then the left arrow to modify the command rather than retyping the entire statement:

```
>> result = 6 + 3
result =
      9
```

This is very useful, especially if a long expression is entered and it contains an error, and it is desired to go back to correct it.

It is also possible to choose command(s) in the Command History window, and rerun them by right-clicking. Consecutive commands can be chosen by clicking on the first or last and then holding down the Shift and up or down arrows (new in 2014a).

To change a variable, another assignment statement can be used, which assigns the value of a different expression to it. Consider, for example, the following sequence of statements:

```
>> mynum = 3
mynum =
          3
>> mynum = 4 + 2
mynum =
          6
>> mynum = mynum + 1
mynum =
          7
```

In the first assignment statement, the value 3 is assigned to the variable *mynum*. In the next assignment statement, *mynum* is changed to have the value of the expression 4 + 2, or 6. In the third assignment statement, *mynum* is changed again, to the result of the expression *mynum* + *1*. Since at that time *mynum* had the value 6, the value of the expression was 6 + 1, or 7.

At that point, if the expression *mynum* + *3* is entered, the default variable *ans* is used since the result of this expression is not assigned to a variable. Thus, the value of *ans* becomes 10 but *mynum* is unchanged (it is still 7). Note that just typing the name of a variable will display its value (the value can also be seen in the Workspace Window).

```
>> mynum + 3
ans =
        10

>> mynum
mynum =
         7
```

1.3.1 Initializing, Incrementing, and Decrementing

Frequently, values of variables change, as shown previously. Putting the first or initial value in a variable is called ***initializing*** the variable.

Adding to a variable is called ***incrementing***. For example, the statement

```
mynum = mynum + 1
```

increments the variable *mynum* by 1.

QUICK QUESTION!

How can 1 be subtracted from the value of a variable called *num*?

This is called ***decrementing*** the variable.

Answer:

```
num = num – 1;
```

1.3.2 Variable Names

Variable names are an example of ***identifier names***. We will see other examples of identifier names, such as function names, in future chapters. The rules for identifier names are as follows:

- The name must begin with a letter of the alphabet. After that, the name can contain letters, digits, and the underscore character (e.g., *value_1*), but it cannot have a space.
- There is a limit to the length of the name; the built-in function **namelengthmax** tells what this maximum length is (any extra characters are truncated).
- MATLAB is case-sensitive, which means that there is a difference between uppercase and lowercase letters. So, variables called *mynum*, *MYNUM*, and *Mynum* are all different (although this would be confusing and should not be done).
- Although underscore characters are valid in a name, their use can cause problems with some programs that interact with MATLAB, so some programmers use mixed case instead (e.g., *partWeights* instead of *part_weights*).
- There are certain words called ***reserved words***, or ***keywords***, that cannot be used as variable names.
- Names of built-in functions (described in the next section) can, but should not, be used as variable names.

Additionally, variable names should always be ***mnemonic***, which means that they should make some sense. For example, if the variable is storing the radius of a circle, a name such as *radius* would make sense; *x* probably wouldn't.

The following commands relate to variables:

- **who** shows variables that have been defined in this Command Window (this just shows the names of the variables)
- **whos** shows variables that have been defined in this Command Window (this shows more information on the variables, similar to what is in the Workspace Window)
- **clear** clears out all variables so they no longer exist

- **clear** *variablename* clears out a particular variable
- **clear** *variablename1 variablename2* … clears out a list of variables (note: separate the names with spaces)

If nothing appears when **who** or **whos** is entered, that means there aren't any variables! For example, in the beginning of a MATLAB session, variables could be created and then selectively cleared (remember that the semicolon suppresses output).

```
>> who
>> mynum = 3;
>> mynum + 5;
>> who
Your variables are:
ans    mynum
>> clear mynum
>> who
Your variables are:
ans
```

These changes can also be seen in the Workspace Window.

1.3.3 Types

Every variable has a ***type*** associated with it. MATLAB supports many types, which are called ***classes***. (Essentially, a class is a combination of a type and the operations that can be performed on values of that type, but, for simplicity, we will use these terms interchangeably for now. More on classes will be covered in Chapter 11.)

For example, there are types to store different kinds of numbers. For float or real numbers, or in other words, numbers with a decimal place (e.g., 5.3), there are two basic types: **single** and **double**. The name of the type **double** is short for ***double precision***; it stores larger numbers than the **single** type. MATLAB uses a ***floating point*** representation for these numbers.

There are many integer types, such as **int8**, **int16**, **int32**, and **int64**. The numbers in the names represent the number of bits used to store values of that type. For example, the type **int8** uses eight bits altogether to store the integer and its sign. As one bit is used for the sign, this means that seven bits are used to store actual numbers (0s or 1s). There are also ***unsigned*** integer types **uint8**, **uint16**, **uint32**, and **uint64**. For these types, the sign is not stored, meaning that the integer can only be positive (or 0).

The larger the number in the type name, the larger the number that can be stored in it. We will for the most part use the type **int32** when an integer type is required.

The type **char** is used to store either single ***characters*** (e.g., 'x') or ***strings***, which are sequences of characters (e.g., 'cat'). Both characters and strings are enclosed in single quotes.

The type **logical** is used to store **true/false** values.

Variables that have been created in the Command Window can be seen in the Workspace Window. In that window, for every variable, the variable name, value, and class (which is essentially its type) can be seen. Other attributes of variables can also be seen in the Workspace Window. Which attributes are visible by default depends on the version of MATLAB. However, when the Workspace Window is chosen, clicking on the down arrow allows the user to choose which attributes will be displayed by modifying Choose Columns.

By default, numbers are stored as the type **double** in MATLAB. The function **class** can be used to see the type of any variable:

```
>> num = 6 + 3;
>> class(num)
ans =
double
```

1.4 NUMERICAL EXPRESSIONS

Expressions can be created using values, variables that have already been created, operators, built-in functions, and parentheses. For numbers, these can include operators such as multiplication and functions such as trigonometric functions. An example of such an expression is:

```
>>2 * sin(1.4)
ans =
    1.9709
```

1.4.1 The Format Command and Ellipsis

The ***default*** in MATLAB is to display numbers that have decimal points with four decimal places, as shown in the previous example. (The default means if you do not specify otherwise, this is what you get.) The **format** command can be used to specify the output format of expressions.

There are many options, including making the format **short** (the default) or **long**. For example, changing the format to **long** will result in 15 decimal places.

This will remain in effect until the format is changed back to **short**, as demonstrated in the following.

```
 >> format long
 >>2 * sin(1.4)
ans =
    1.970899459976920

>> format short
>> 2 * sin(1.4)
ans =
    1.9709
```

The **format** command can also be used to control the spacing between the MATLAB command or expression and the result; it can be either **loose** (the default) or **compact**.

```
>> format loose
>> 5*33
ans =

   165

>> format compact
>> 5*33
ans =
   165
>>
```

Especially long expressions can be continued on the next line by typing three (or more) periods, which is the ***continuation operator***, or the ***ellipsis***. To do this, type part of the expression followed by an ellipsis, then hit the Enter key and continue typing the expression on the next line.

```
>> 3 + 55 - 62 + 4 - 5...
+  22 -1

ans =
    16
```

1.4.2 Operators

There are in general two kinds of operators: ***unary*** operators, which operate on a single value, or ***operand***, and ***binary*** operators, which operate on two values or operands. The symbol "−," for example, is both the unary operator for negation and the binary operator for subtraction.

Here are some of the common operators that can be used with numerical expressions:

```
+  addition
-  negation, subtraction
```

```
*  multiplication
/  division (divided by e.g. 10/5 is 2)
\  division (divided into e.g. 5\10 is 2)
^  exponentiation (e.g. 5^2 is 25)
```

In addition to displaying numbers with decimal points, numbers can also be shown using ***scientific or exponential notation***. This uses ***e*** for the exponent of 10 raised to a power. For example, 2 * 10 ^ 4 could be written two ways:

```
>> 2 * 10^4
ans =
        20000
>> 2e4
ans =
        20000
```

1.4.2.1 Operator Precedence Rules

Some operators have ***precedence*** over others. For example, in the expression 4 + 5 * 3, the multiplication takes precedence over the addition, so first 5 is multiplied by 3, then 4 is added to the result. Using parentheses can change the precedence in an expression:

```
>> 4 + 5 * 3
ans =
    19
>> (4 + 5) * 3
ans =
    27
```

Within a given precedence level, the expressions are evaluated from left to right (this is called ***associativity***).

Nested parentheses are parentheses inside of others; the expression in the ***inner parentheses*** is evaluated first. For example, in the expression `5 – (6 * (4 + 2))`, first the addition is performed, then the multiplication, and finally the subtraction, to result in `–31`. Parentheses can also be used simply to make an expression clearer. For example, in the expression `((4 + (3 * 5)) – 1)`, the parentheses are not necessary, but are used to show the order in which the parts of the expression will be evaluated.

For the operators that have been covered thus far, the following is the precedence (from the highest to the lowest):

```
( )        parentheses
^          exponentiation
–          negation
*, /, \    all multiplication and division
+, –       addition and subtraction
```

PRACTICE 1.1

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

```
1\2
−5 ^ 2
(−5) ^ 2
10 − 6/2
5 * 4/2 * 3
```

1.4.3 Built-in Functions and Help

There are many built-in functions in MATLAB. The **help** command can be used to identify MATLAB functions, and also how to use them. For example, typing **help** at the prompt in the Command Window will show a list of ***help topics*** that are groups of related functions. This is a very long list; the most elementary help topics appear at the beginning. Also, if you have any Toolboxes installed, these will be listed.

For example, one of the elementary help topics is listed as **matlab\elfun**; it includes the elementary math functions. Another of the first help topics is **matlab\ops**, which shows the operators that can be used in expressions.

To see a list of the functions contained within a particular help topic, type **help** followed by the name of the topic. For example,

```
>> help elfun
```

will show a list of the elementary math functions. It is a very long list, and it is broken into trigonometric (for which the default is radians, but there are equivalent functions that instead use degrees), exponential, complex, and rounding and remainder functions.

To find out what a particular function does and how to call it, type **help** and then the name of the function. For example, the following will give a description of the **sin** function.

```
>> help sin
```

Note that clicking on the *fx* at the left of the prompt in the Command Window also allows one to browse through the functions in the help topics. Choosing the Help button under Resources to bring up the Documentation page for MATLAB is another method for finding functions by category.

To ***call a function***, the name of the function is given followed by the ***argument(s)*** that are passed to the function in parentheses. Most functions then ***return value(s)***. For example, to find the absolute value of −4, the following expression would be entered:

```
>> abs(−4)
```

which is a ***call*** to the function **abs**. The number in the parentheses, the -4, is the ***argument***. The value 4 would then be ***returned*** as a result.

QUICK QUESTION!

What would happen if you use the name of a function, for example, **abs**, as a variable name?

Answer: This is allowed in MATLAB, but then **abs** could not be used as the built-in function until the variable is cleared. For example, examine the following sequence:

```
>> clear
>> abs(-6)
ans =
     6
>> abs = 11
abs =
    11
>> abs(-6)
Subscript indices must either be real positive
integers or logicals.
>> who
Your variables are:
abs   ans
>> clear abs
>> who
Your variables are:
ans
>> abs(-6)
ans =
     6
```

All of the operators have a functional form. For example, 2+5 can be written using the **plus** function as follows.

```
>> plus(2,5)
ans =
     7
```

MATLAB has a useful shortcut that is called the ***tab completion*** feature. If you type the beginning characters in the name of a function, and hit the tab key, a list of functions that begin with the typed characters pops up. This feature has improved over the years; beginning in R2015b, capitalization errors are automatically fixed.

Also, if a function name is typed incorrectly, MATLAB will suggest a correct name.

```
>> abso(-4)
Undefined function or variable 'abso'.
Did you mean:
>> abs(-4)
```

1.4.4 Constants

Variables are used to store values that might change, or for which the values are not known ahead of time. Most languages also have the capacity to store ***constants***, which are values that are known ahead of time and cannot possibly change. An example of a constant value would be **pi**, or π, which is

3.14159... In MATLAB, there are functions that return some of these constant values, some of which include:

pi	3.14159....
i	$\sqrt{-1}$
j	$\sqrt{-1}$
inf	infinity ∞
NaN	stands for "not a number," such as the result of 0/0

1.4.5 Random Numbers

When a program is being written to work with data, and the data are not yet available, it is often useful to test the program first by initializing the data variables to ***random numbers***. Random numbers are also useful in simulations. There are several built-in functions in MATLAB that generate random numbers, some of which will be illustrated in this section.

Random number generators or functions are not truly random. Basically, the way it works is that the process starts with one number, which is called the ***seed***. Frequently, the initial seed is either a predetermined value or it is obtained from the built-in clock in the computer. Then, based on this seed, a process determines the next "random number." Using that number as the seed the next time, another random number is generated, and so forth. These are actually called ***pseudorandom***—they are not truly random because there is a process that determines the next value each time.

The function **rand** can be used to generate uniformly distributed random real numbers; calling it generates one random real number in the ***open interval*** (0,1), which means that the endpoints of the range are not included. There are no arguments passed to the **rand** function in its simplest form. Here are two examples of calling the **rand** function:

```
>> rand
ans =
    0.8147
>> rand
ans =
    0.9058
```

The seed for the **rand** function will always be the same each time MATLAB is started, unless the initial seed is changed. The **rng** function sets the initial seed. There are several ways in which it can be called:

```
>> rng('shuffle')
>> rng(intseed)
>> rng('default')
```

With 'shuffle', the **rng** function uses the current date and time that are returned from the built-in **clock** function to set the seed, so the seed will always be different. An integer can also be passed to be the seed. The 'default' option will set the seed to the default value used when MATLAB starts up. The **rng** function can also be called with no arguments, which will return the current state of the random number generator:

```
>> state_rng = rng;    % gets state
>> randone = rand
randone =
     0.1270
>> rng(state_rng);  % restores the state
>> randtwo = rand    % same as randone
randtwo =
     0.1270
```

Note
The words after the % are comments and are ignored by MATLAB.

The random number generator is initialized when MATLAB starts, which generates what is called the ***global stream*** of random numbers. All of the random functions get their values from this stream.

As **rand** returns a real number in the open interval (0, 1), multiplying the result by an integer *N* would return a random real number in the open interval (0, *N*). For example, multiplying by 10 returns a real in the open interval (0, 10), so the expression

```
rand*10
```

would return a result in the open interval (0, 10).

To generate a random real number in the range from *low* to *high*, first create the variables *low* and *high*. Then, use the expression `rand * (high – low) + low`. For example, the sequence

```
>> low = 3;
>> high = 5;
>> rand * (high – low) + low
```

would generate a random real number in the open interval (3, 5).

The function **randn** is used to generate normally distributed random real numbers.

1.4.5.1 Generating Random Integers

As the **rand** function returns a real number, this can be rounded to produce a random integer. For example,

```
>> round(rand*10)
```

would generate one random integer in the range from 0 to 10 inclusive (`rand*10`would generate a random real in the open interval (0, 10); rounding that will return an integer). However, these integers would not be evenly distributed in the range. A better method is to use the function **randi**, which in its simplest form **randi(imax)** returns a random integer in the range from 1 to imax, inclusive. For example, **randi(4)** returns a random integer in the range from 1 to 4. A range can also be passed, for example, **randi ([imin, imax])** returns a random integer in the inclusive range from imin to imax:

```
>> randi([3, 6])
ans =
     4
```

PRACTICE 1.2

Generate a random

- real number in the range (0,1)
- real number in the range (0, 100)
- real number in the range (20, 35)
- integer in the inclusive range from 1 to 100
- integer in the inclusive range from 20 to 35

1.5 CHARACTERS AND STRINGS

A character in MATLAB is represented using single quotes (e.g., 'a' or 'x'). The quotes are necessary to denote a character; without them, a letter would be interpreted as a variable name. Characters are put in an order using what is called a ***character encoding***. In the character encoding, all characters in the computer's ***character set*** are placed in a sequence and given equivalent integer values. The character set includes all letters of the alphabet, digits, and punctuation marks; basically, all of the keys on a keyboard are characters. Special characters, such as the Enter key, are also included. So, 'x', '!', and '3' are all characters. With quotes, '3' is a character, not a number.

Notice the difference in the formatting (the indentation) when a number is displayed versus a character:

```
>> var = 3
var =
     3
>> var = '3'
var =
3
```

MATLAB also handles strings, which are sequences of characters in single quotes.

```
>> myword = 'hello'
myword =
hello
```

The most common character encoding is the American Standard Code for Information Interchange, or ASCII. Standard ASCII has 128 characters, which have equivalent integer values from 0 to 127. The first 32 (integer values 0 through 31) are nonprinting characters. The letters of the alphabet are in order, which means 'a' comes before 'b', then 'c', and so forth. MATLAB actually can use a much larger encoding sequence, which has the same first 128 characters as ASCII. More on the character encoding and converting characters to their numerical values will be covered in Section 1.7.

1.6 RELATIONAL EXPRESSIONS

Expressions that are conceptually either true or false are called ***relational expressions***; they are also sometimes called ***Boolean expressions*** or ***logical expressions***. These expressions can use both ***relational operators***, which relate two expressions of compatible types, and ***logical operators***, which operate on **logical** operands.

The relational operators in MATLAB are:

Operator	Meaning
`>`	`greater than`
`<`	`less than`
`>=`	`greater than or equals`
`<=`	`less than or equals`
`==`	`equality`
`~=`	`inequality`

All these concepts should be familiar, although the actual operators used may be different from those used in other programming languages, or in mathematics classes. In particular, it is important to note that the operator for equality is two consecutive equal signs, not a single equal sign (as the single equal sign is already used as the assignment operator).

For numerical operands, the use of these operators is straightforward. For example, `3 < 5` means "3 less than 5," which is, conceptually, a true expression. In MATLAB, as in many programming languages, "true" is represented by the **logical** value 1, and "false" is represented by the **logical** value 0. So, the expression `3 < 5` actually displays in the Command Window the value 1 (**logical**) in MATLAB. Displaying the result of expressions like this in the Command Window demonstrates the values of the expressions.

```
>> 3 < 5
ans =
     1

>> 2 > 9
ans =
     0
>> class(ans)
ans =
logical
```

The type of the result is **logical**, not **double**. MATLAB also has built-in **true** and **false**.

```
>> true
ans =
     1
```

In other words, **true** is equivalent to **logical(1)** and **false** is equivalent to **logical (0)**. (In some versions of MATLAB, the value shown for the result of these expressions is **true** or **false** in the Workspace Window.) Although these are **logical** values, mathematical operations could be performed on the resulting 1 or 0.

```
>> logresult = 5 < 7
logresult =
    1
>> logresult + 3
ans =
    4
```

Comparing characters (e.g., 'a' < 'c') is also possible. Characters are compared using their ASCII equivalent values in the character encoding. So, 'a' < 'c' is a **true** expression because the character 'a' comes before the character 'c'.

```
>> 'a' < 'c'
ans =
    1
```

The logical operators are:

Operator	Meaning
||	or
&&	and
~	not

All logical operators operate on **logical** or Boolean operands. The **not** operator is a unary operator; the others are binary. The **not** operator will take a **logical** expression, which is **true** or **false**, and give the opposite value. For example, `~(3 < 5)` is **false** as `(3 <5)` is **true**. The **or** operator has two **logical** expressions as operands. The result is **true** if either or both of the operands are **true**, and **false** only if both operands are **false**. The **and** operator also operates on two **logical** operands. The result of an **and** expression is **true** only if both operands are **true**; it is **false** if either or both are **false**. The or/and operators shown here are used for ***scalars***, or single values. Other or/and operators will be explained in Chapter 2.

The || and && operators in MATLAB are examples of operators that are known as ***short-circuit*** operators. What this means is that if the result of the expression can be determined based on the first part, then the second part will not even be evaluated. For example, in the expression:

```
2 < 4 || 'a' == 'c'
```

the first part, 2 < 4, is **true** so the entire expression is **true**; the second part 'a' == 'c' would not be evaluated.

In addition to these logical operators, MATLAB also has a function **xor**, which is the exclusive or function. It returns **logical true** if one (and only one) of the arguments is **true**. For example, in the following, only the first argument is **true**, so the result is **true**:

```
>> xor(3 < 5, 'a' > 'c')
ans =
     1
```

In this example, both arguments are **true** so the result is **false**:

```
>> xor(3 < 5, 'a' < 'c')
ans =
     0
```

Given the **logical** values of **true** and **false** in variables x and y, the ***truth table*** (see Table 1.1) shows how the logical operators work for all combinations. Note that the logical operators are ***commutative*** (e.g., $x \,||\, y$ is the same as $y \,||\, x$).

As with the numerical operators, it is important to know the operator precedence rules. Table 1.2 shows the rules for the operators that have been covered thus far in the order of precedence.

QUICK QUESTION!

Assume that there is a variable *x* that has been initialized. What would be the value of the expression

```
3 < x < 5
```

if the value of *x* is 4? What if the value of *x* is 7?

Answer: The value of this expression will always be **logical true**, or 1, regardless of the value of the variable *x*. Expressions are evaluated from left to right. So, first the expression 3 < *x* will be evaluated. There are only two possibilities: either this will be **true** or **false**, which means that either the expression will have the **logical** value 1 or 0. Then, the rest of the expression will be evaluated, which will be either 1 < 5 or 0 < 5. Both of these expressions are **true**. So, the value of *x* does not matter: the expression 3 < *x* < 5 would be **true** regardless of the value of the variable *x*. This is a logical error; it would not enforce the desired range. If we wanted an expression that was **logical true** only if *x* was in the range from 3 to 5, we could write `3 < x && x < 5` (note that parentheses are not necessary).

Table 1.1 Truth Table for Logical Operators

x	*y*	~*x*	*x* || *y*	*x* && *y*	xor(*x*,*y*)
true	true	false	true	true	false
true	false	false	true	false	true
false	false	true	false	false	false

Table 1.2 Operator Precedence Rules

Operators	Precedence
Parentheses: ( )	Highest
Power ^	
Unary: Negation (-), not (~)	
Multiplication, division *, /, \	
Addition, subtraction +, -	
Relational <, <=, >, >=, ==, ~=	
And &&	
Or ||	
Assignment =	Lowest

PRACTICE 1.3

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

```
3 == 5 + 2

'b' < 'a' + 1

10 > 5 + 2

(10 > 5) + 2

'c' == 'd' - 1 && 2 < 4

'c' == 'd' - 1  ||  2 > 4

xor('c' == 'd' - 1, 2 > 4)

xor('c' == 'd' - 1, 2 < 4)

10 > 5 > 2
```

Note: Be careful about using the equality and inequality operators with numbers. Occasionally, ***roundoff errors*** appear, which means that numbers are close to their correct value but not exact. For example, cos(pi/2) should be 0.