Basic data types, operators & expressions

Scientific Computing
Fall, 2019
Paul Gribble

1 Expressions

When you start MATLAB you are greeted with a command prompt:

```
>>
```

You are now in the read-eval-print loop and MATLAB is waiting for you to enter an *expression*, so that MATLAB can evaluate that expression and provide you with the result. For example, you might enter something that looks like arithmetic:

```
>> 1+2

ans =

     3
```

MATLAB evaluates that expression 1+2 and prints out the value of that expression, which is 3, and assigns that output value to a new *variable* called `ans`. We will talk about variables soon.

Try typing in another arithmetic expression, for example:

```
>> 1/3

ans =

    0.3333
```

So you can see that MATLAB can do division too.

Expressions don't have to be arithmetic. They could be logical expressions, such as:

```
>> 1+2 == 3

ans =

     1
```

In this case the double-equal sign is an *operator* which means “is equal to?”. Essentially our expression is asking MATLAB a logical question (a question with a TRUE or FALSE answer): Is `1+2` equal to `3`? MATLAB evaluates that expression and returns the answer: `1`. In MATLAB a logical TRUE is the same as the number 1, and a logical FALSE is the same as the number `0`. Try another logical expression:

```
>> 1+1 == 0

ans =

     0
```

In this case we are asking MATLAB “Does 1+1 equal 0?” and MATLAB returns `0`, which is MATLAB’s way of saying `FALSE`.

Here’s another one in which we combine multiple operators into one expression:

```
>> 5+6-1+20>25

ans =

     1
```

With the numbers and operators all squished next to each other this is a bit hard to read. I might prefer to write this expression with spaces in between, and round brackets surrounding the left hand side, to make it more readable:

```
>> (5 + 6 - 1 + 20) > 25

ans =

     1
```

It’s up to you how to write your code, but I would suggest to you that writing your code in such a way that it is easy to read is a good idea in the long run. It will make it easier for other people to read your code (including yourself in the future).

Here’s an example to illustrate this point. Can you figure out what the result of this expression is?

```
>> 2*6*3*4/3/4/2/5>1
```

It’s difficult and annoying to try to do this. How about this re-written version:

```
>> (2*6 * 3*4) / (3*4 * 2*5)  >  1

ans =

     1
```

They are both valid code, they both evaluate to the same result, but one version (the second version) is much more readable (in my opinion).

Here's a puzzling result:

```
>> 0.1 + 0.2 == 0.3

ans =

     0
```

This is a rather surprising result, isn't it. I'll leave it as an exercise for you to research why this happens, and what a potential solution to this kind of unexpected result might be.

Let's move on and talk about operators.

2 Operators

In the example code snippets above we saw a number of operators already. We saw the `+` and `/` mathematical operators, and we saw the logical operator `==`. There are in fact a wide variety of operators in MATLAB. The MathWorks (the company that makes MATLAB) has a web page that lists them all:

Operators and Elementary Operations

There are a variety of arithmetics operators, relational and logical operators, and others that you can read about as well.

One concept that is important to talk about is *operator precedence*. This refers to the order in which MATLAB evaluates expressions and operators when there are multiple operations in a single expression. Take the following expression for example:

```
>> 2 + 3 * 5
```

What does this evaluate to? There are two possibilities. If you proceed left-to-right and evaluate each operator in the order in which it appears, then this would evaulate to `2+3`, which is `5`, multiplied by `5`, which equals `25`. This is not what happens in MATLAB (nor in most programming languages). Instead the multiply operator `*` takes precedence over the addition operator `+`, and the `3*5` sub-expression is evaluated first, and then the result (which is `15`) is substituted, and then the resulting expression `2+15` is evaulated, which returns `17`.

```
>> 2 + 3 * 5

ans =

    17
```

Here is a page from the MathWorks documentation on MATLAB that describes operator precedence in MATLAB:

Operator Precedence

For arithmetic the easy rule to remember is that multiply and divide take precedence over add and subtract.

You can force particular parts of an expression to be evaluated first by using round brackets, which take the highest precedence in MATLAB. For example we could rewrite the expression above to force the `2+3` to occur first, like this:

```
>> (2 + 3) * 5

ans =

    25
```

3 Variables

In the above examples we have been typing in numbers, along with arithmetic and logical or relational operators, and MATLAB evaluates those expressions and returns the result. In fact when you don't provide any output variable to store the results of your expression, MATLAB automatically stores the result in a variable called `ans` (short for *answer*). So for example:

```
>> 1 + 2

ans =

     3
```

MATLAB has stored the answer in a *variable* called `ans`. You can think of a variable as a human-readable name of some data that is stored in MATLAB's memory. You can refer to data by it's variable name. Under the hood, MATLAB keeps track of how these variable names correspond to the location (and type) of the data stored in memory.

This memory we are referring to is RAM or Random-access memory. This is a form of data storage in your computer which is to be considered temporary. Once MATLAB quits, or your computer is turned off, the data that was stored in RAM is gone. To permanently store data on your computer you need to store it on a more permanent form of memory, such as the hard drive in your computer, or an external drive such as a memory stick.

By naming your data using a variable name, you can easily view and manipulate those data. Here's an example where we store the result of a calculation in a variable that we will name `fred`:

```
>> fred = 1 + 2

fred =

     3
```

We type our expression `1+2` and on the left hand side we type our variable name `fred`, and set it to be equal to (using the equal sign `=`) the expression. MATLAB evaluates this whole expression and in its return statement we can see that now `fred` is equal to `3`.

Here's another one:

```
>> bob = 4 * 5

bob =

    20
```

Now we have defined a second variable called `bob` which we have set to be equal to the result of the expression `4*5`. We can see in MATLAB's return statement that now bob is equal to `20`.

We can use variable names within expressions and MATLAB will substitute the value of those variables within the expression:

```
>> joe = bob + fred

joe =

    23
```

We have defined a new variable called `joe` and assigned it to be equal to the value of `bob` (which is `20`) added to the value of `fred` (which is `3`). MATLAB returns that now `joe` is equal to `23`.

What happens if we do this?

```
>> mike = joe + bob + fred + danny
Undefined function or variable 'danny'.
```

MATLAB returns an error: `Undefined function or variable 'danny'`. The problem here is that we have never defined a variable called `danny` and so when MATLAB attempts to evaluate `danny`, it can't find anything. When it evaluates `joe` and `bob` and `fred` MATLAB knows the data that those variable names refers to, but we have not named any data using a variable called `danny` and so MATLAB has no idea what we are referring to.

In fact this is exactly the right way to think about this error: when we type `danny`, MATLAB does not know what we are referring to.

At any time we can get a list of which variables are defined in MATLAB by using a command called `who`:

```
Your variables are:

bob   fred  joe
```

We can see we have three variables defined. You can use a command called `whos` to get a more detailed list:

```
>> whos
  Name      Size            Bytes  Class     Attributes

  bob       1x1                 8  double
  fred      1x1                 8  double
  joe       1x1                 8  double
```

We see our variables in a table now with their name, their size, the number of Bytes that they occupy in MATLAB's memory, their class (what *type* of variable they are, which relates to what kind of digital representation holds those data) and a column called `Attributes`.

Note that if you assign a new value to an existing variable, the old data is wiped out. Here is an example. We first assign the number `3` to the new variable `jane`:

```
>> jane = 3

jane =

     3
```

Now we verify that indeed `jane` is `3`:

```
>> jane

jane =

     3
```

Now we reassign the number `4` to `jane`, and check the value:

```
>> jane = 4

jane =

     4

>> jane

jane =

     4
```

Indeed, `jane` is now `4` and there is no trace of `3`.

There are some rules governing how you can name your variables. Variable names cannot start with a number or a symbol, only with a letter. There can be no spaces or symbols in variable names. Capitalization matters, so `joe` is different than `Joe`.

The other thing to talk about in this context is that MATLAB has some commands and functions that are already defined by MATLAB, and so you should avoid using those as your own variable names. So for example MATLAB has a built-in function called `sort()` that will sort a vector of values:

```
>> sort([4 3 2 6 5 7 9 8 1])

ans =

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

When you type `sort` MATLAB executes its built-in sorting algorithm. Nothing stops you however from defining your own variable with the same name:

```
>> sort = 23

sort =

    23
```

Now when you try typing the sorting expression in again you get this:

```
>> sort([4 3 2 6 5 7 9 8 1])
Index exceeds matrix dimensions.
```

MATLAB throws an error. Now when MATLAB sees `sort` it thinks you are referring to your variable called `sort` which equals `23`. Actually it equals a `1x1` matrix (a single value) containing `23`.

Why do we get this particular error message? The round brackets when put next to a variable cause MATLAB to try to index into a vector or matrix, and since our variable sort has only a single value, when MATLAB tries to retrieve the 4th, then 3rd, then 2nd, values, etc, it throws an error. We haven't talked about vectors or matrices or indexing yet, so don't worry about that. The point here is that we have essentially wiped out the reference to the sorting algorithm originally referred to by sort by defining our own variable called `sort`. Oops!

We can remedy this situation by clearing the variable `sort` using the built-in command `clear`:

```
>> clear sort
```

Now we have erased our variable called `sort` and when we type `sort` again, MATLAB will no longer refer to our variable containing `23` (since we just cleared it from memory) and MATLAB will go back to referring to its own built-in function called `sort()`:

```
>> sort([4 3 2 6 5 7 9 8 1])

ans =

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

Now it's time to talk about variable types.

4 Basic Data Types

So far we have been dealing with data in the form of single numbers. The number `1` for example, or the number `0.5`. There are in fact a number of different numeric *types* of data that MATLAB can store in variables. Here is a webpage from the MathWorks that describes the full constellation of data types used in MATLAB:

Data Types

Numeric data can be stored in a number of different Numeric Types. The default type of numeric data in MATLAB is `double`, which stands for double-precision floating-point format. The *floating-point* part of this means essentially that this data type can store a real number, i.e. numbers along a continuous line such as `1.0` or `1.33` or `3.14159`. The *double-precision* part of this refers to how many *bytes* are used by MATLAB to represent that number. More bytes means more precision.

When you just type in numbers, or have MATLAB compute the result of an arithmetic expression, you will typically be using, by default, the `double` data type:

```
>> a = 1

a =

     1

>> b = 2

b =

     2

>> c = a/b

c =

    0.5000

>> d = b/a

d =

     2

>> whos
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  b         1x1                 8  double
  c         1x1                 8  double
  d         1x1                 8  double
```

If you want to convert a variable to another numeric data type, you can do it using one of MATLAB's built-in conversion functions. So for example to convert a `double` variable to a 32-bit integer, use `int32()`:

```
>> x = 1.3

x =

    1.3000

>> y = int32(x)

y =

           1

>> whos
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
  y         1x1                 4  int32
```

You can see that when the `double` `x` (which equals `1.3`) is converted into an `int32` it is rounded down to `1`.

MATLAB also has data types to deal with individual characters (letters like `'a'` and `'b'`) and strings of characters (like `"joe"`), and a selection of built-in functions to manipulate strings:

Characters and Strings

For example:

```
>> x = 'a'

x =

a

>> y = 'b'

y =

b

>> z = "fred"

z =

"fred"

>> zs = 'fred'

zs =

'fred'

>> whos
  Name      Size            Bytes  Class     Attributes

  x         1x1                 2  char
  y         1x1                 2  char
  z         1x1               156  string
  zs        1x4                 8  char
```

Above we have defined three variables all of type `char` (which stands for character string). The first two, named `x` and `y` both contain a single character (`'a'` and `'b'`, respectively) and the third, `z`, contains a string of four characters (`'fred'`). You can see that `x` and `y` occupy 2 bytes of memory and `z` uses 8 bytes. Two bytes are required to store a single character in MATLAB.

You can dive deeper here in the documentation for the char function, which describes how characters are represented. The first 7 bits (values 0 to 127) code 7-bit ASCII characters. The next 9 bits code values 128 to 65535 and represent characters that depend on your locale (i.e. other languages besides plain english ASCII).

You can quickly see the integer codes for different characters in MATLAB by doing the following:

```
>> int8('a')

ans =
   97

>> int8('b')

ans =

   98

>> int8('z')

ans =

  122
```

In fact you can get the integer codes for all 26 lower case letters in one go, like this:

```
>> int8('a':'z')

ans =

  Columns 1 through 15

   97   98   99  100  101  102  103  104  105  106  107  108  109  110  111

  Columns 16 through 26

  112  113  114  115  116  117  118  119  120  121  122
```

You can display a string to the screen using the `disp` command:

```
>> disp('hello, world, my name is fred')
hello, world, my name is fred
```

You can concatenate multiple strings using the square brackets `[` and `]` to construct a new string:

```
>> a = 'fred';
>> b = 'joe';
>> c = 'jane';
>> s = ' ';
>> z = [a,s,b,s,c];
>> disp(z)
fred joe jane
```

I've introduced some new syntax here, the use of the semicolon `;` after an expression. This prevents MATLAB from echoing the value of the expression to the screen. The expression is still evaluated but MATLAB doesn't echo the result back to us on the screen. Use this when you want to suppress the output of expressions. If you don't need to see the result of an expression on the screen then this makes for a cleaner MATLAB session.

You can get attributes of a string such as its length:

```
>> disp(['z is ', num2str(l), ' characters long'])
z is 13 characters long
```

I've also introducted the built-in function `num2str()` which will convert a numeric type into a character string.

Another way to generate a character string out of many parts is to use the `sprintf()` function. This mimics the `printf()` function that is familiar to C programmers:

```
>> m = sprintf('z is %d characters long, & pi is approx. %.5f', l, pi);
>> disp(m)
z is 13 characters long, and pi is approximately 3.14159
```

You can also use `fprintf()` to do it in one go:

```
>> fprintf('z is %d characters long, & pi is approx. %.5f\n', l, pi);
z is 13 characters long, and pi is approximately 3.14159
```

The `%d` notation tells the sprintf function that an integer numeric type will be provided here. The `%.5f` notation says that a floating-point value will be provided, and please show it using 5 decimal places. At the end of the string is where you supply the needed values, in the order in which they appear in the string. Note that `pi` is a built-in value in MATLAB.

In MATLAB you can use the `class()` function to get the type of a variable. For example:

```
>> a = 3.14159

a =

    3.1416

>> class(a)

ans =

double
```

You can use the `isa()` function to ask whether a variable is a certain type. For example:

```
>> isa(a,'char')

ans =

     0

>> isa(a,'double')

ans =

     1
```

Remember in MATLAB `0` is "FALSE" or "NO" and `1` is "TRUE" or "YES".

So far we have seen numeric types and character string types. These are basic data types. MATLAB also allows for complex data types such as vectors, matrices, structures and cell arrays. These you can think of as container types, in other words data types that can store not just one value but many values.

Actually, the character string is already a sort of container type, in that it stores many single characters all strung together. You can think of a character string as a vector of single characters.

In the next section in the notes we will talk about some of these complex data types and how to use them.

5 Special values

The MathWorks online documentation has a page on various special values built-in to MATLAB:

Special Values

There is a special numeric value in MATLAB called `NaN` (not a number). It is often used to denote missing data.

There is also a special value called `Inf` which stands for infinity. Try typing the expression `1/0` and you will get `Inf`.

There are other mathematical special values such as `pi`:

```
>> pi

ans =

    3.1416

>> help pi
 PI     3.1415926535897....
    PI = 4*atan(1) = imag(log(-1)) = 3.1415926535897....

    Reference page in Help browser
       doc pi
```

and imaginary numbers `i` and `j`:

```
>> help i
 I  Imaginary unit.
    As the basic imaginary unit SQRT(-1), i and j are used to enter
    complex numbers.  For example, the expressions 3+2i, 3+2*i, 3+2j,
    3+2*j and 3+2*sqrt(-1) all have the same value.

    Since both i and j are functions, they can be overridden and used
    as a variable.  This permits you to use i or j as an index in FOR
    loops, etc.

    See also J.

    Reference page in Help browser
       doc i

>> help j
 J  Imaginary unit.
```

```
    As the basic imaginary unit SQRT(-1), i and j are used to enter
    complex numbers.  For example, the expressions 3+2i, 3+2*i, 3+2j,
    3+2*j and 3+2*sqrt(-1) all have the same value.

    Since both i and j are functions, they can be overridden and used
    as a variable.  This permits you to use i or j as an index in FOR
    loops, subscripts, etc.

    See also I.

    Reference page in Help browser
       doc j
```

Also of note is the special function `eps()`:

```
>> help eps
 EPS  Spacing of floating point numbers.
    D = EPS(X), is the positive distance from ABS(X) to the next larger in
    magnitude floating point number of the same precision as X.
    X may be either double precision or single precision.
    For all X, EPS(X) is equal to EPS(ABS(X)).

    EPS, with no arguments, is the distance from 1.0 to the next larger double
    precision number, that is EPS with no arguments returns 2^(-52).

...
```

If we type `eps(1.0)` we get:

```
>> eps(1.0)

ans =

   2.2204e-16
```

which is a pretty small number: `0.00000000000000022204`. This is the distance between the floating-point representation of `1.0` and the next largest number that the `double` floating-point representation can represent. You can think of it as the precision of the `double` floating-point representation of numbers, near the number `1.0`.

Try `eps(2^54)` (which equals `18,014,000,000,000,000`):

```
>> eps(2^54)

ans =

     4
```

Huh? So near the number `2^54`, the precision of our `double` floating-point representation of continuous numbers is `4.0`! This is terrible! This is however just a limitation of representing continuous (infinite) numbers using a finite digital representation.

6 Getting help

We can get help about MATLAB built-in commands and functions using the `help` command:

```
>> help who
 WHO    List current variables.
    WHO lists the variables in the current workspace.

    In a nested function, variables are grouped into those in the nested
    function and those in each of the containing functions.  WHO displays
    only the variables names, not the function to which each variable
    belongs.  For this information, use WHOS.  In nested functions and
    in functions containing nested functions, even unassigned variables
    are listed.

    WHOS lists more information about each variable.
    WHO GLOBAL and WHOS GLOBAL list the variables in the global workspace.
    WHO -FILE FILENAME lists the variables in the specified .MAT file.

    WHO ... VAR1 VAR2 restricts the display to the variables specified. The
    wildcard character '*' can be used to display variables that match a
    pattern.  For instance, WHO A* finds all variables in the current
    workspace that start with A.

    WHO -REGEXP PAT1 PAT2 can be used to display all variables matching the
    specified patterns using regular expressions. For more information on
    using regular expressions, type "doc regexp" at the command prompt.

    Use the functional form of WHO, such as WHO('-file',FILE,V1,V2),
    when the filename or variable names are stored in strings.

    S = WHO(...) returns a cell array containing the names of the variables
    in the workspace or file. You must use the functional form of WHO when
    there is an output argument.

    Examples for pattern matching:
        who a*                      % Show variable names starting with "a"
        who -regexp ^b\d{3}$        % Show variable names starting with "b"
                                    %   and followed by 3 digits
        who -file fname -regexp \d  % Show variable names containing any
                                    %   digits that exist in MAT-file fname

    See also WHOS, CLEAR, CLEARVARS, SAVE, LOAD.

    Other functions named who:
       Simulink.who

    Reference page in Help browser
       doc who
```

We can also get a GUI interface to help using the `doc` command.

There is also a way of searching the help documentation files for keywords, using the `lookfor` command. The `lookfor` command will return the names of all functions or commands for which the associated help documentation contains the given keyword. So for example let's say we need to find the `invkine()` function but we've forgotten what it's called, we just remember it's something to do with a robot. We can search using: `lookfor robot`

```
>> lookfor robot
invkine                         - Inverse kinematics of a robot arm.
invkine_codepad                 - Modeling Inverse Kinematics in a Robotic Arm
idnlgreydemo13                  - Modeling an Industrial Robot Arm
idnlgreydemo8                   - Industrial Three-Degrees-of-Freedom Robot: C MEX-File Modeling of MIMO Syste
robot_m                         - A simplified Manutec r3 robot with three arms.
robotarm_m                      - A physically parameterized robot arm.
refmodel_dataset                - ROBOTARM_DATASET Reference model dataset
robotarm_dataset                - Robot arm dataset
mech_robot_data                 - Data defining the manutec robot.
RobotArmExample                 - Multi-Loop PID Control of a Robot Arm
```

7 Script M-files

Instead of typing in commands into the MATLAB command-line, you can instead save them in a file, called a MATLAB script, and then type the name of the script on the command line to execute all code within that script. Scripts typically have a `.m` filename suffix.

So for example you might have a file called `random8.m` that contains the following code:

```
% script M-file example random8.m
%
rlist = round(rand(1,8)*10);
disp(rlist);
disp(['mean = ',num2str(mean(rlist))]);
disp(['median = ',num2str(median(rlist))]);
disp(['standard deviation = ',num2str(std(rlist))]);
```

The script generates a list of 8 random numbers chosen from a uniform distribution between 0 and 10, and then displays the mean, median and standard deviation of those values.

If the script file called `random8.m` is in your MATLAB path, then typing `random8.m` on the MATLAB command line will execute the script:

```
>> random8
     8     9     1     9     6     1     3     5

mean = 5.25
median = 5.5
standard deviation = 3.3274
```

8 MATLAB path

When you first start MATLAB, you will be faced with the command line prompt, and MATLAB will be started up looking at a particular location in your file system. This location is known as the *current working directory*. If you are using the MATLAB GUI (graphical user interface) you will see your current working directory displayed in a toolbar just above the command line. On my computer it shows as `/Users/plg/Desktop`. On your computer it will be something different.

The other way to query MATLAB about the current working directory is to type `pwd` into the command line:

```
>> pwd

ans =

/Users/plg/Desktop
```

When you type something into the command line, like `random8`, MATLAB will go through a number of steps to find out what you mean:

- is `random8` defined as a variable in memory?
- is `random8` defined as a function or script file or data file in MATLAB's current working directory?
- is `random8` defined as a function or script file or data file somewhere else in MATLAB's path?

The MATLAB *path* is a list of directories on your computer's hard disk where MATLAB knows to look for scripts and functions. You can see what's defined in your MATLAB path by typing `path` at the MATLAB command line:

```
>> path

          MATLABPATH

     /Users/plg/Documents/MATLAB
     /Applications/MATLAB_R2015a.app/toolbox/matlab/addons
     /Applications/MATLAB_R2015a.app/toolbox/matlab/addons/cef
     /Applications/MATLAB_R2015a.app/toolbox/matlab/addons/fallbackmanager
     /Applications/MATLAB_R2015a.app/toolbox/matlab/demos
     /Applications/MATLAB_R2015a.app/toolbox/matlab/graph2d
     /Applications/MATLAB_R2015a.app/toolbox/matlab/graph3d
     /Applications/MATLAB_R2015a.app/toolbox/matlab/graphics
     ...
        ...
```

On my computer I get a list of more than 600 directories—almost all of them subdirectories of the MATLAB main application directory. This is where all of MATLAB's built-in functions and scripts are located, and where the various MATLAB toolbox code is located.

The other way to see (and alter) your MATLAB path is by using the MATLAB GUI. Type `pathtool` on the MATLAB command line and you get a nice GUI interface where you can scroll through all of the directories that are in your MATLAB path, you can delete some, add some, and change the order.

On the issue of the order: remember that MATLAB goes through its path in the order in which the directories appear in the path list. So if you have a function called `random8()` defined in multiple places in your path, when you type `random8` on the MATLAB command line, MATLAB will use the first one it finds in the path.

My personal approach to the MATLAB path is to basically never mess with it. Instead of adding data directories and script directories associated with my various projects to the MATLAB path, instead I just start MATLAB from the appropriate location when I am working on different projects.

To change the current working directory you can either click on the toolbar in the MATLAB GUI, or use the `cd` command on the MATLAB command line, for example:

```
>> cd /Users/plg/Documents/Research/projects/Heather_fMRI/
>> pwd

ans =

/Users/plg/Documents/Research/projects/Heather_fMRI
```