

# Lesson #9



## User-Defined Functions and Functions Files

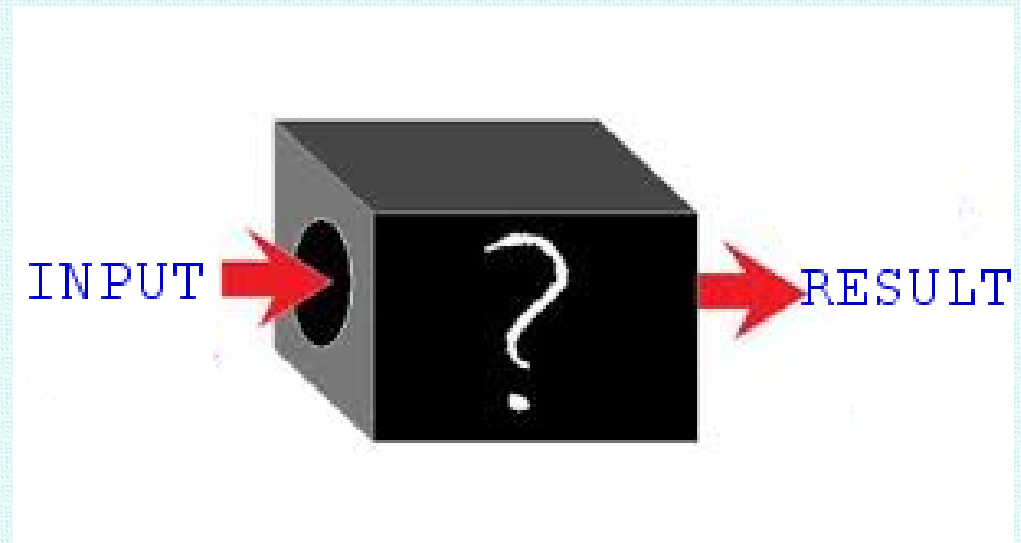
# What are functions?

- A function is like a command that you create yourself. It can behave like a command already defined in MATLAB.
- There are also functions that programmers of the MATLAB organization have created and have placed in MATLAB for us to use. We've seen a number of the "built in" functions up to this point (sqrt, sin, ...), but in this chapter we will be mainly concerned with functions we will write ourselves.



# What are functions?

- Functions are “black boxes” that accept some data and return results. Many commands that we already know have one piece of data going in as **input** and 1 piece of data coming out as **result** (like *sqrt*) but we have encountered some with two inputs going in (like *nthroot*). Functions defined by you can have the number of inputs you need (0, 1, 2, 3, and more...).



# Function Arguments

- A function is called or executed (run) by a program or another function. That program can pass its data to the function and receive the function's result.
- We call the values sent to the function when it is called, arguments. There can be one or more arguments.
- With `sqrt(x)`, `x` is the argument. The result is `sqrt(x)`, which can be assigned to a variable like in `y = sqrt(x);`

# Why use functions?

- With a function, you can use the same code in more than one place in a program without rewriting the code. For example you can use *sqrt* many times in the same program.
- You can reuse the code by calling in the same function from different programs. For example, you can use *sqrt* in all your m. files if needed.

# Creating a function

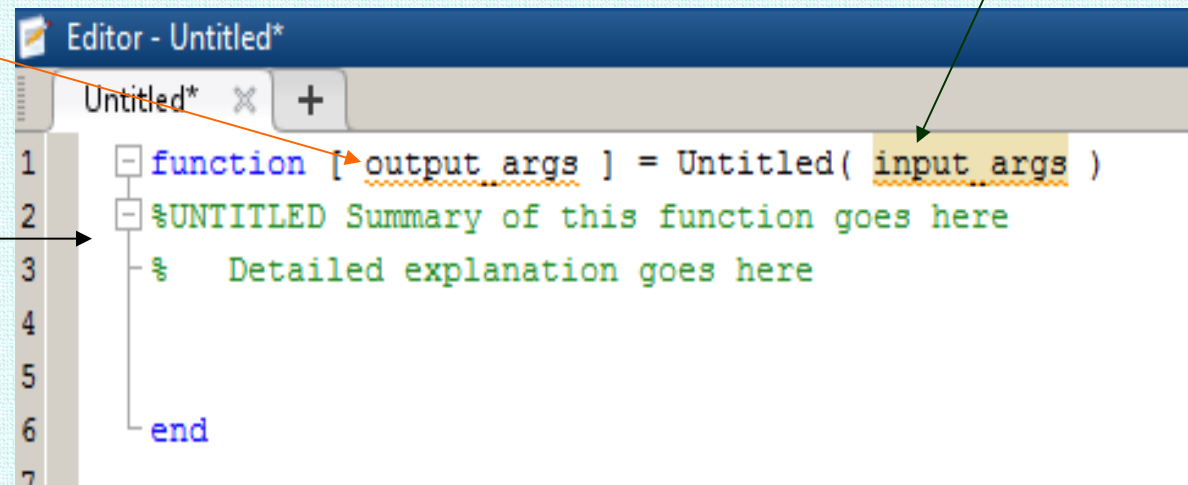
A function is written almost exactly like a script, except that it starts with the **function** keyword. Like a script file, it is saved with a `.m` extension (it is recommended to use the same file name as your function's name). You can make the file in any text editor but it is easiest to do it with the MATLAB Editor/Debugger Window. To create a new function, select the *Function* item from the *New* menu.

Results  
(Output values)

Data (Input values)

Code generated  
automatically by  
MATLAB

*Note: You can use New/Script and write all the keywords yourself instead of New/Function.*



```
1 function [output_args] = Untitled(input_args)
2 %UNTITLED Summary of this function goes here
3 % Detailed explanation goes here
4
5
6 end
```

# A simple function

- This is a simple function that converts miles to kilometers.

```
1 function [km] = miles_to_km (miles)
2 %miles_to_km converts distances in miles to distances in kilometers.
3 %Input:
4 %miles = distance in miles.
5 %Results:
6 %km = distance in kilometers.
7 km = miles * 1.60934;
8 end
```

RESULT

DATA (INPUT)

THE FUNCTION DEFINITION LINE

H1 LINE

HELP TEXT

FUNCTION BODY AND ASSIGNMENT OF VALUE TO RESULT VARIABLE

We will explain all the elements in detail in the coming slides.



# Sending Data to Functions

- Data input parameters (in the function header) and arguments (used when calling/invoking the function) are used to transfer data into the function from the calling program.
- You can have zero, one or more items of data sent to the function. If there are more than one, you separate the multiple data items with commas inside the parentheses of the function definition line.
- Data can be scalars, vectors, or arrays. Their values are specified by the calling program.



# Getting Results from Functions

- Results (or output) are used to transfer data from the function to the calling program.
- You can have zero, one or more results from a function. If there are none, you can omit the assignment operator (=). If there is only one, you can omit the brackets. If more than one, you separate them with commas inside the brackets [ ] of the function definition line.
- Results can be scalars, vectors, or arrays.

# Function Definition Line

```
function [output arguments] = function_name(input arguments)
```



The diagram illustrates the syntax of a function definition line. A box at the top contains the code: `function [output arguments] = function_name(input arguments)`. Below this box, four arrows point to specific parts of the code, each accompanied by a descriptive text block. The first arrow points to the word `function`. The second arrow points to the brackets and text `[output arguments]`. The third arrow points to the text `function_name`. The fourth arrow points to the text `(input arguments)`. The text blocks provide rules for each part: the word `function` must be in lower-case; the output arguments must be in brackets; the function name must consist of letters, digits, and underscores, and cannot be a built-in function name; and the input arguments must be in parentheses and are also known as parameters.

The word `function` must be the first word, and must be typed in lower-case letters.

A list of output arguments typed inside brackets.

The name of the function.

A list of input arguments typed inside parentheses.  
(also known as **parameters**)

- A function name is made up only of letters, digits, and underscores. It cannot have spaces. It follows the same rules as variable names.
- *Avoid making function names that are same as names of built-in functions.*

# Function Definition Line

- The function definition line must be first executable line (not a blank line nor a comment line) of the function file. If not, MATLAB considers file to be a script file.
- The function definition line establishes the file as a function file, defines the name of the function, and defines the number and order of input items and results.
- As you see, a function file is really a script file but with special keywords and a special purpose.

# The H1 Line and Help Text Lines

- *Help text lines* are comment lines (ones that start with a percent sign %) placed just below the function definition line. The first help text line, often called the H1 line, typically includes the program name and a brief description.
- End your help text with a blank line (without a %). The help system ignores any comment lines that appear after the help text block. Note that help text lines are optional.
- We will see an example of a function with H1 and help text lines on the next slide.



# The Simplest Function: No Arguments, No Result

```
function why
%displays 25 '?' on the screen.
%Input: none.
%Results: none.
```

H1 line and  
help text lines.

If you type **help**  
**why**, you will get  
them in the  
command window.

```
disp ('????????????????????????????????');
end
```

You call (use) the function by simply entering the name of the function in the command window or your script.

```
>> why
????????????????????????????????
```

# A Simple Function: One Argument, No Result

- The *why* function only displays the same number of question marks every time.
- What if I wanted to display a different number of question marks that could be linked to a numerical value that I send to the function?
- All I would need to add is a parameter to the function definition line to specify the number of question marks. Now the function will have to be called with a corresponding argument to be passed (copied) to the parameter.
- See **why2** on the next slide. *why2*, like *why*, has no result.

# A Simple Function: One Argument, No Result

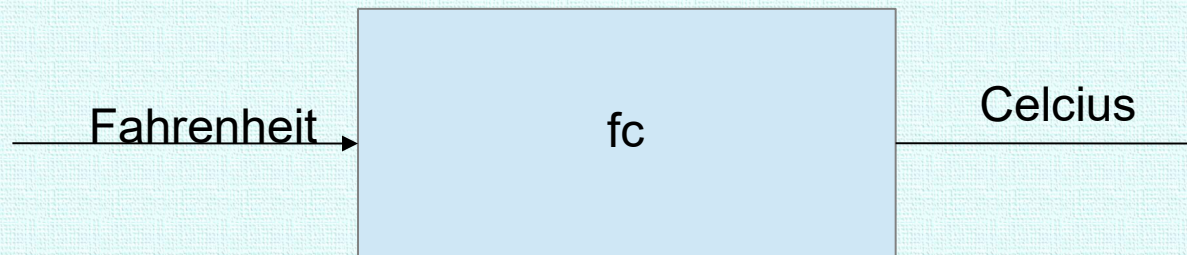
```
function why2 (n)
%displays question marks on the screen.
%Input: n = number of question marks.
%Results: none.

for k=1:1:n
    fprintf ('?');
end
fprintf ('\n');
end
```

```
>> why2 (6)
??????
>> why2 (10)
??????????
```

# A Simple Function: One Argument, One Result

- The `miles_to_km` function seen earlier is a typical example of that very common type of function of having one input and one result.
- Let's have a similar example of converting degrees Fahrenheit into degrees Celcius.





# A Simple Function: One Argument, One Result

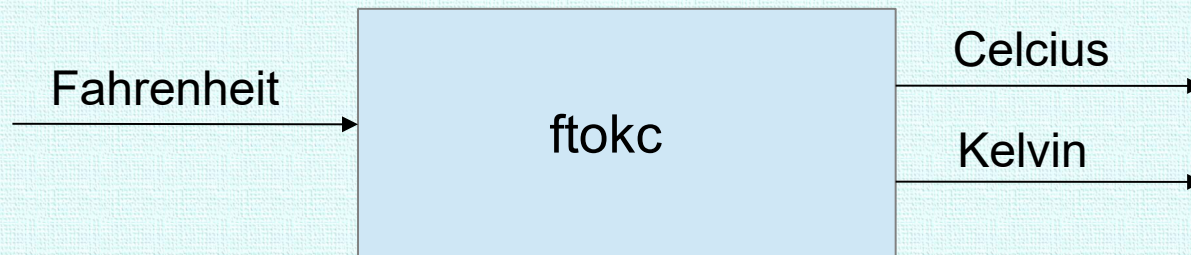
```
function c = fc (f)
%converts f into c.
%Input: f = temperature in Fahrenheit.
%Result: c = temperature in Celcius.
```

```
c = (f - 32) * 5/9;
end
```

```
>> fc (30)
ans =
    -1.1111
>> fc (68)
ans =
    20
```

## Another Function: One Argument, Two Results

- Let's have a similar example of converting degrees Fahrenheit into degrees Kelvin as well as into degrees Celsius.



# Another Function: One Argument, Two Results

```
function [k,c] = ftokc (f)
%converts degrees f into degrees k and c.
%Input: f = temperature in fahrenheit.
%Results: k = temperature in kelvin.
%         c = temperature in celcius.
```

```
c = (f - 32) * 5/9;
k = c + 273.15;
end
```

```
>> [kelvin celcius] = ftokc (80)
celcius =
    26.6667
kelvin =
    299.8167
```

## Another Function: One Argument, Two Results

- Here a script that uses our `ftokc` function (the script is saved as `degrees.m`):

```
far = input ('Enter temperature in degrees Fahrenheit:
');
[kel cel] = ftokc (far);
fprintf ('%.1f degrees Fahrenheit is %.1f degrees
Kelvin.\n', far, kel);
fprintf ('%.1f degrees Fahrenheit is %.1f degrees
Celcius.\n', far, cel);
```

```
>> degrees
```

```
Enter temperature in degrees Fahrenheit: 45
```

```
45.0 degrees Fahrenheit is 280.4 degrees
```

```
Kelvin.
```

```
45.0 degrees Fahrenheit is 7.2 degrees Celcius.
```



# Function: Multiple Arguments, One Result

- Let's now have function with three arguments and one result. The formula for a loan repayment is:
- $M = P * ( J / (1 - (1 + J)^{-N}))$  where
- M: is the monthly payment.
- P: is the principal or amount of the loan
- J: is the monthly interest; the annual interest divided by 100, then divided by 12.
- N: is the number of months of amortization, determined by length in years of loan.
- *The three arguments are P, J, and N. The result is M.*

# Function: Multiple Arguments, One Result

```
function m = loan (p, j, n)
%loan repayment function
%Inputs: p = principal.
%        j = yearly interest as a percentage.
%        n = length of loan in months.
%Result: m = monthly payment.
```

The arguments below must match the number and order of these three parameters!!!

```
j = j / 100 / 12; %compute monthly interest
m = p * ( j / (1 - (1 + j)^ (-n)))
end
```

```
>> format bank
>> loan (100000, 6, 360)
ans =
    599.55
```

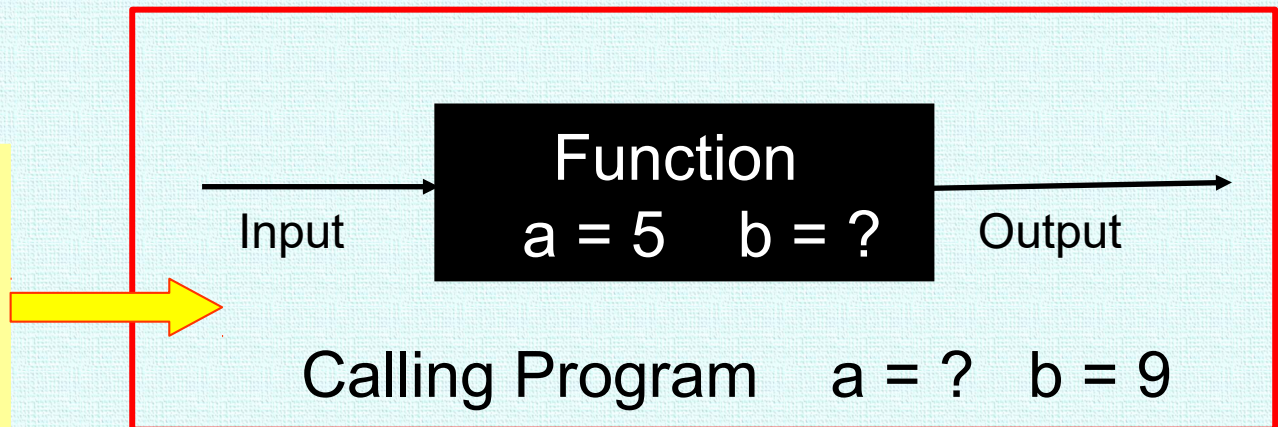
The three arguments  
\$100,000 loan  
6% interest per year  
30 years

# Local Variables inside functions

Variables declared inside a function are *local* to that function, meaning they can only be used inside that function.

- The calling program can't see (access) any of the variables inside the function.
- The function can't see any variables in the calling program.

a and b from the function are different variables from a and b of the calling program!



# Scope of Variable Names

**Remember!**

**Variables inside and outside functions can have the same names but are still different variables occupying different memory locations!**

**Changing one doesn't change the other!**





# Scope of Variable Values - An example

```
function test( n )  
  
    fprintf( 'On entering  
function n = %d\n', n );  
  
    n = 10;  
  
    fprintf( 'Before leaving  
function n = %d\n', n );  
  
end
```

```
>> n = 5
```

```
n = 5
```

```
>> test(n)
```

```
On entering function n = 5
```

```
Before leaving function n =  
10
```

```
>> n
```

```
n = 5
```



n is still 5 after the function call

# Local vs. Global Variables

- You may see in MATLAB literature mentions of global variables. These variables are visible to both the calling code and the called function.
- The use of global variables is not considered good practice. As a result, using global variables on tests/exam or assignments will be considered an error.



# Anonymous Functions

- *Anonymous functions* or *inline functions* are functions defined without using a separate function file. They are useful to define short, one-line, functions that are used many times in one program but do not need to be reused in other programs.
- They are specified within the main script file itself, not in a separate file.
- We saw a form of those anonymous functions when we used the `fplot` command earlier.

# Anonymous Functions

`name = @ (arglist) expr`

The name of the anonymous function.  
(the function handle)

The @ symbol.

A list of input arguments (independent variables).

Mathematical expression.

Example:

`cube = @ (x) x^3`

The mathematical expression must contain the variable(s) from the list of arguments (*arglist*).

# Anonymous Functions - Examples

```
>> triple = @(n) 3 * n;
```

```
>> triple (4)
```

```
ans = 12
```

```
>> x = 1.5;
```

```
>> y = triple(x)
```

```
y = 4.5000
```

```
>> km = @(miles) 1.60934 * miles;
```

```
>> km (50)
```

```
ans = 80.47
```

```
>> km (triple (30));
```

```
ans = 144.84
```



# Anonymous Function - Example 1

Let's create a function named **perimeter** that calculates the sum of all three sides of a right-angled triangle given the length of the other smaller sides (saved as `perimeter.m`). That function uses the anonymous function **hypo** that we create as well and calculates the length of the third (longest) side (the hypotenuse).

```
function p = perimeter (a, b)
%perimeter of a right-angled triangle.
%Inputs: a, b = two shorter sides of the triangle.
%Result: p = the length of the perimeter.
hypo = @(x,y) sqrt(x^2 + y^2);
p = a + b + hypo (a,b);
end
```

```
>> perimeter (3,4)
ans = 12
```

# Anonymous Function - Example 2

Let's look at this other example. Try it in MATLAB!

```
function root = posnegroot(a, b, c)
root = @(a,b,c,sign)(-b + sign*sqrt(b^2 - 4*a*c))/(2*a);
% where sign would be +1 or -1
rootsign = input('Type + for positive root, - for
negative root: ','s');
if rootsign == '+'
    fprintf('Positive root is: %6.3f\n',root(a,b,c,1));
else
    fprintf('Negative root is: %6.3f\n',root(a,b,c,-1));
end
end
```

quite a bit shorter than  
the long formula!



```
>> postnegroot(3,5,2);
Type + for positive root, - for negative root: +
Positive root is -0.667
>>
```

# End of Lesson