Scripts

Scripts

In order to organise a succession of instructions and reuse a previous work, one uses *scripts*

FirstScript.m

```
clear all
close all
clc

% This is a Matlab script, that allows the instructions
% to be ordered and saved
a = 23;
b = 12*a;
```

A Matlab® script is a file with a *.m* extension that contains all your ordered instructions

Scripts

Create your first script:

1. Click on New → Script
2. Write within the file

```
% This is my first script
a = 23;
b = 12*a;
```

3. Save the script with the name *"1_FirstScript.m"*

   What is happening?

Create your first script:

1. Click on New $\rightarrow$ Script
2. Write within the file

```
% This is my first script
a = 23;
b = 12*a;
```

3. Save the script with the name *"1_FirstScript.m"*

   What is happening?

$\rightarrow$ A script's name should always begin with a letter

$\rightarrow$ The extension should also always be ".m", otherwise Maltab® won't recognise it

Scripts

Run your first script:

4. Save the script with the name *"L1_FirstScript.m"*
5. Run the script by clicking on the run button
6. Look at the variables present in the workspace, and check the output of `whos`

Scripts

Run your first script:

4. Save the script with the name *"L1_FirstScript.m"*
5. Run the script by clicking on the run button
6. Look at the variables present in the workspace, and check the output of `whos`

→ Nothing is prompted during the run (use of semicolons)

→ The two variables `a` and `b` appear in the workspace and have the desired values

Scripts

Create and call your first script from the prompt:

7. Run the script from command line by typing `L1_FirstScript`
8. Type `L1_Fi` and press TAB
9. Close L1_FirstScript by clicking on the cross on the top right of the built-in text editor
10. Type `edit L1_FirstScript` in the prompt
11. Type `edit L1_SecondScript` in the prompt

Scripts

Create and call your first script from the prompt:

7. Run the script from command line by typing `L1_FirstScript`
8. Type `L1_Fi` and press TAB
9. Close L1_FirstScript by clicking on the cross on the top right of the built-in text editor
10. Type `edit L1_FirstScript` in the prompt
11. Type `edit L1_SecondScript` in the prompt

→ Tabbing after having written the first letters of a script's name in the prompt *autofills* its name

→ The command `edit` opens or creates a new script, automatically with a ".m" extension

Comments on the top of the file

Always give indications on how to use your code to any potential reader (usually yourself)

- Summarises the purpose of the code
- Specifies the variables taken in input and their format
- Specifies the variables given in output and their format
- Further useful information for using the script

When written at the very top of the file, those comments are accessible externally through the commands

```
>> help L1_FirstScript
>> L1_FirstScript + F1
```

Comments through the file

Give indications on how you though your code as you are writing it, in particular in technical areas where the code understanding is not straightforward

- Eases the understanding of the code for an external reader
- Eases the maintenance of the code

Types of comments

```
% single line comment

%% Describes a code block
```

```
%{
multi line
comment
%}
```

Scripts and documentation

Best practice

- Always start a script with a comment block describing its usage (documentation)
- Then add `clear all; close all; clc;` to clear the prompt and previous workspace variables
- Separate the parts of your code having distinct aims with a block separator `%% A comment` that describes the main goal of the block below the separator
- Comment the technical parts thorough your code
- Write in a legible way and use coherent indentation

Think your code to be efficient and write with a Matlab® spirit

A dream code

```
%{
    This script is a demonstration of a good practice when coding,
    which is fully commented.
    Input: No input
    Output: Variables a and b will be saved in the workspace
                a: integer, the square root of five
                b: integer, the golden ratio
%}

% Clearing the space
clear all; close all; clc


%% Initialisation

% Computing the required quantities
a = sqrt(5);            % Assigning to a the computed value of sqrt(5)
c = 2;                  % Storing the denominator for the purpose of the example

%% Actual computation

% Creating the golden number
b = (1+a)/c
```

Note that Matlab® helps you with *syntax highlighting*

Functions

Create your own functions

A *function* is a group of instructions that executes a given task

- it takes none, one or many *arguments* (input variables)
- it performs the tasks according to the instructions
- it returns none, one or many *output variables* to the script or prompt that calls it

In Matlab®, the function should be named as the containing file, and its structure is as follows

```
function [output1, output2] = TheFunction(arg1, arg2)
    % The help of the function
    The main content of the function
end
```

Functions

Create your first function:

1. Click on New → Function
2. Change the content to:

```
function y = L1_power8(x)
    % Compute y=x^8 (comment displayed in the help)
    y=x^8;
end
```

3. Save it to `L1_power8.m`
4. Type in command line `L1_power8(2)`

Functions

Create your first function:

1. Click on New → Function
2. Change the content to:

```
function y = L1_power8(x)
    % Compute y=x^8 (comment displayed in the help)
    y=x^8;
end
```

3. Save it to `L1_power8.m`
4. Type in command line `L1_power8(2)`

→ The instructions within the function are executed on the value $x = 2$ and the result is printed in the prompt

→ No variable `y` in the workspace, but `ans` contains 256

Functions

Create your first function with multiple arguments:

1. Create a function `L1_ThreeArguments` as follows

   ```
   function [y1, y2] = L1_ThreeArguments(x1,x2,x3)
       % example function with more than one input
       y1=x1^2;
       y2=x2+x3;
   end
   ```

2. Save it to `L1_ThreeArguments.m`
3. Call the function from the prompt with `L1_ThreeArguments(2,3,4)`. What does `ans` value?

Functions

Create your first function with multiple arguments:

1. Create a function `L1_ThreeArguments` as follows

```
function [y1, y2] = L1_ThreeArguments(x1,x2,x3)
    % example function with more than one input
    y1=x1^2;
    y2=x2+x3;
end
```

2. Save it to `L1_ThreeArguments.m`
3. Call the function from the prompt with `L1_ThreeArguments(2,3,4)`. What does `ans` value?

→ The variable ans only contains the returned value for `y1`, the return value for `y2` is not automatically given

Functions

Call your first function with multiple arguments:

4. Store the output of the function by

   `[a, b] = L1_ThreeArguments(2,3,4)`

5. Observe the values of `a` and `b` if you simply write

   `a, b = L1_ThreeArguments(2,3,4)`

6. Retrieve only some outputs among all of those returned

   `a = L1_ThreeArguments(2,3,4)`

   `[a, ~] = L1_ThreeArguments(2,3,4)`

   `[~, b] = L1_ThreeArguments(2,3,4)`

Functions

Call your first function with multiple arguments:

4. Store the output of the function by

   `[a, b] = L1_ThreeArguments(2,3,4)`

5. Observe the values of `a` and `b` if you simply write

   `a, b = L1_ThreeArguments(2,3,4)`

6. Retrieve only some outputs among all of those returned

   `a = L1_ThreeArguments(2,3,4)`

   `[a, ~] = L1_ThreeArguments(2,3,4)`

   `[~, b] = L1_ThreeArguments(2,3,4)`

→ Warning: using `a, b = L1_ThreeArguments(2,3,4)` assigns the first returned value (`y1`) to both `a` and `b`

Functions

Accessing the number of input and output variables

Within a function, the number of input and output arguments specified in its declaration are accessible with the keywords

`nargin`
`nargout`

From outside the function, use the command `nargin(@function)`

```
>> nargin(@sin)
>> nargin(@L1_ThreeArguments)
>> nargout(@L1_ThreeArguments)
```

Note: Functions may be called with less inputs, but not with more

Multiples functions in a file

Defining multiple functions in a single file is possible. However, only the function whose name is matching the file name can be called from the command line or from other scripts

Example: *Content of the file "L1_power8_2.m"*

```
function y = L1_power8_2(x)
    y=mypower8(x);
end

function y = mypower8(x)
    y=x^8;
end
```

The prompt can only execute `L1_power8_2(2)`, not `mypower8(2)`

Scoping and local variables

The interface between the function and the workspace (or script) is done only through the input and output arguments

```
function y = power8_local(x)
    % Power eight function with a local variable
    y=x^8      % Is sent back to the workspace
    myVar = 5; % Does not appear in the workspace
end
```

Any variable that does not appear in the function declaration is a *local variable* and stays within the *scope* of the function

→ Local variables can be named the same in different functions

Global variables

When a variable needs to be shared between different scripts, functions, workspace scopes, one can declare a global variable by

```
global global_var
```

It can then be called everywhere, specifying its global nature

```
function y = power8_global(x)
    % Power eight function with a global variable
    y=x^8;           % Is sent back to the workspace

    global my_global_var % Generates a global variable
    my_global_var=2*y;   % appearing in the workspace
end
```

Note: Using global variables is usually a *bad* practice: prefer arguments

Anonymous functions

Whenever the definition of the function is simple (typically when it takes a single line), it is convenient to use *Anonymous functions*

```
>> f = @(x,y) x^y
>> f(2,3)
```

- The right hand side is an anonymous function
- The variable `f` is a *function handle*

Note: To witness the type of the variable `f`, use `class(f)`

Remark: The old *inline* syntax (e.g. `g = inline('sin(2*pi*f + theta)', 'f', 'theta')`) is depreciated and will be cleared

Closures

Let an anonymous function be defined from a parameter a. Then, if this parameter changes later in the code, the function does not automatically change. This is due to *closures*

```
>> clear all;
>> a=2;
>> f = @(x) x+a;
>> f(2) % Gives out 4
>> a=3;
>> f(2) % Still gives out 4
```

Functions

Best practice

- Always write documentation for your functions
- Choose the most efficient and straightforward way to perform a task: do not reinvent the wheel
- Test your functions by themselves as you write it
- Use the minimum (or none) global variables
- Use variable names that are meaningful and coherent with each other (even for local variables)
- Indent carefully the code blocks

Tips and tricks

Matlab® hacks

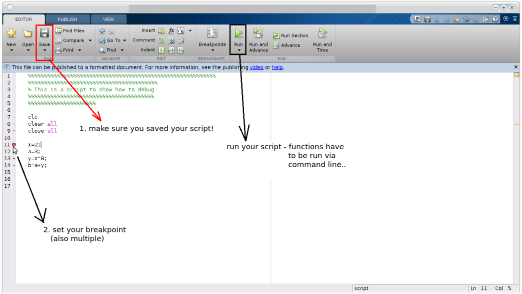
- The keyword `return` leaves a script or function
- Three dots **...** splits an equation into multiple lines

```
function y = power8(x)
   y=... % very long expression over multiple lines
     x^8;
end
```

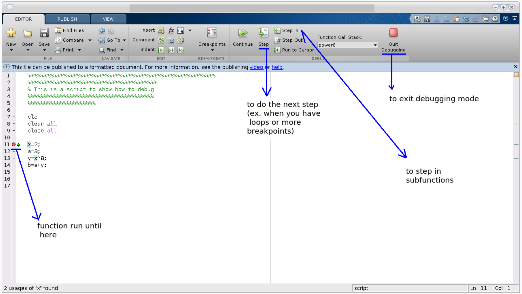
User interaction

- The instruction `disp(x)` displays the value of a variable or expression of `x` in the prompt
- The instruction `error('my message')` raises an error message to the user and exits the function or script
- The keyword `pause` suspends the run until a key is pressed

Debugging



1. make sure you saved your script!
run your script - functions have to be run via command line..
2. set your breakpoint (also multiple)



to do the next step (ex. when you have loops or more breakpoints)
to exit debugging mode
to step in subfunctions
function run until here

Exercises

Exercises: Command Line

Exercise

1. Compute the number of hours and seconds of a year
2. Use the help tool of Matlab to display the square root (`sqrt`)
3. Find the function for computing n-roots and compute $\sqrt[3]{10}$
4. Compute the multiplicative inverse of $1 + 2i$

Exercises: Basic operations

Exercise

5. Overwrite the variables `i,j` and observe the output of

```
>> 3i
>> 3*i
```

6. Determine for which number of the following expressions there exist $(a_1, a_2, a_3) \in \mathbb{Z}^3$ such that

$$(a_1 \text{ op}_1\ a_2) \text{ op}_2\ a_3 \neq a_1 \text{ op}_1\ (a_2 \text{ op}_2\ a_3)$$

   a) $(\text{op}_1, \text{op}_2) = (-, -)$
   b) $(\text{op}_1, \text{op}_2) = (-, +)$
   c) $(\text{op}_1, \text{op}_2) = (+, \&)$ (1+0)&0 1+(0&0)
   d) $(\text{op}_1, \text{op}_2) = (\&, |)$
   e) $(\text{op}_1, \text{op}_2) = (*, ==)$
   f) $(\text{op}_1, \text{op}_2) = (\&, >=)$

Exercises: Basic operations

Exercise

7. Add as many (simple) **brackets** as possible, without changing the meaning of the expression. Then, type the initial expression and your final suggestion in Matlab.

   a) `2 + 4 * - 1 / 3 / 4`
   b) `2 + 4 == 5 | 3 + 5 ^ - 1 & 2`
   c) `- 3 < 7 < 5`
   d) `sin ( n + 1 ) / ( n + 2 )`

Exercises: Basic operations

Exercise

8. Which value has `ans` after each of the following commands?

```
>> 23;
>> ans^2;
>> x = sqrt(ans);
>> x^2 - ans;
>> x^2 - ans;
>> ans - x^2;
```

9. Think about how complex numbers could be rewritten from polar coordinates (angle in radian and radius) to Cartesian coordinates (real and imaginary part). Then, take radius `r=0.2` and the angle `p=1.32` and write the corresponding complex number.

Exercise

10. Try to implement the following function:

```
function y = noinput()
   x=2;
   z=3;
   y=x+z;
end
```

Exercises: Functions

Exercise

11. Write a function, that gets as input the integer a and b and gives as output both the **integer division** a/b and **remainder**.

12. Write a function `curry(f,g,x)`, which takes as input two anonymous functions and a parameter and gives as output $f(g(x))$.