

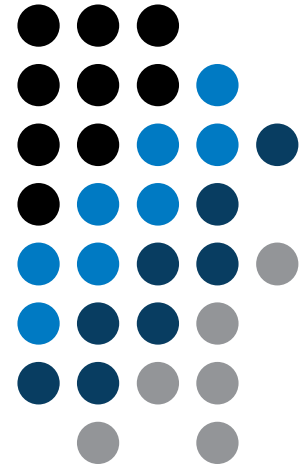
B0B17MTB – Matlab

Part #8



Miloslav Čapek
miloslav.capek@fel.cvut.cz
Viktor Adler, Pavel Valtr, Filip Kozák

Department of Electromagnetic Field
B2-634, Prague



Learning how to ...

Strings

eval, feval

```
HmARLrknhjQfbOQnBcKjKE_FhnPOAYreP_hF]lcMR\D'
o]EUJr[maXEq`HTm[\WJMO[\UnPaOMRi[^LFarFJAjYX:
Pcop^pUCOBLVEGMLlgRT^[QkNoTcNBp[b_frekrfHQBC:
moWfoioWjrSIj^qYMBn_QYUE^l\Omhg^\O\rYcyFKMED:
SVqIm\Qm\XiSg\gcKjLC_Nfyh[^LSOkq`mrahUYDiRkr'
T^LaSYUQNgMqoNLMMLVj_JirHkLUQVQEBCKYNU^CmkEL
WhA\VCWj_foQfLV[aXJLUAfDV\VEODERaYTQFSSYhck
TOIGAfZegNJDVdq\C^N\WFSgncqGaT]JTRRSFZiRYF]Z:
DejRGbjbGSbZqNLSGEeSTPOMXrTpIofk_FWaCBooZlSm
fcbO^_iAKri`cinbB\[lJoqQ`[WRQETLYdGjOjYaWUBo:
bVIcoS`mY`XFFFwo`oDEpAI fj_ZpfdflqrnOCjIBg\Q]:
jDO\_UMUTEG_akYPICLS]]g^FaDSofDfMLAGKKnNEhb_:
YUeOingQdB_FCCBp[f^ePKYfibtDUC^OU^PHrFQBoSr\
l\AZdcmdoAiBZafN_mahYUldjAE\kNg`emgKCHdGLWXE
g[DJAqjWrhYgKjQeHeCdGr^NVoZDaWHg[EnlCamRbWWA.
[reT^]ZHOZHU^ixbfJ_gVVYKjZFSjGaedFpV]EYHPGRb
YBSRNNfGiPraBgcoDcek\kCfblQZWIKC[Ln\EkCHKGRB:
LFEJc\[p`dVMoigDnap\PEVSkRCrRUTF^HSodMfQSYKO:
eqg[W`PWbjPaZHFFlbjp`Z\r`kYAM\FXIQFVdgoFQm[N:
YcZOAObHLl_aDKg`DaZpBeTcdFCaZ[eNlfqISEoi eh]S:
^KMaQ[GWrTDO\fpY`fcGnS[rpiViWTdLlLOC\phMcAgQ:
B^eadHFYTOJpTG\B\TgIX^EYgGdjZARqHgSO\UoRFMHi:
RncBYbUH]pprjallgIDZEVPSrlpMCjc^K[CVJQokMSEh:
mAcOjOTpjmGRd`jLPKBCOBOfD^AkDYIVlaqTUGnbIPN:
```

Characters in Matlab

- string = array (a vector or a matrix or a cell) of characters
 - Try to avoid diacritics (accent) in Matlab

- string is created using apostrophes

```
>> st = 'Hello, world!'
```

- strings are outputs of some functions (e.g. `>> char(65)`)
- each character in a string is an element of an array and requires 2 B
 - datatype `char`
- when an apostrophe is required to be part of a string, it is to be typed as two quote characters:

```
>> pt = 'That''s it!'
```

Strings in Matlab

- characters treated as strings (from **R2016b**): `string`

```
>> str = string(1);  
>> str + 1 % = "11"  
>> chr = char(1);  
>> chr + 1 % = 2
```

- unlike `char`, `string` does not treat numbers as ASCII or Unicode
- `string` can be created by double quotes (from **R2017a**)

```
>> str = "a"  
>> whos
```

- in the following both `char` and `string` are considered to be strings

Strings – principles

- in the case string has more than one line, it has to have same number of columns

```
>> st = ['george'; 'pepi  ']
```

- otherwise (usually) strings are stored as `cell` datatype:

```
pt = {'george', 'pepi', 'and all others', 'including accents ěščř'}
```

- whether a given variable is of type `char` is tested this way:

```
>> ischar(st)
>> iscellstr(pt)
```

Strings - type conversion

- quite often, it is required to convert from a number code to a string and vice versa, e.g.

- double \rightarrow char
- char \rightarrow double
- char \rightarrow uint16

```
>> tx = char([65:70])  
  
>> B = double(tx)  
  
>> C = uint16(tx)  
  
>> whos
```

- operations with strings are similar to operations with numerical arrays
 - holds true for indexing in the first place!

```
>> S1 = 'test'; S2 = '_b5';  
>> S3 = [S1 S2]  
>> size(S3), size(S3')  
>> S4 = [S3(3:5) 'end']
```

Strings

200 s ↑

- create an arbitrary string
 - find out its length
 - try to convert the string into double type
 - try to index selected parts of the string

- questions???

Strings – number conversion #1

- conversion of number in a string (char) to number (double):
 - conversion of multiple numbers (function `str2num`):

```
>> str2num('[1 2 3 pi]')
>> str2num('[1, 2;3 4]')
```

```
>> str2num('[1 2 3 pi]')
ans =
    1.0000    2.0000    3.0000    3.1416
```

- conversion of a single number to double (`str2double`):

```
>> str2double('1 +1j')
>> str2double('-0.5453')
```

```
>> str2num('[1, 2;3 4]')
ans =
     1     2
     3     4
```

- pay attention to possible errors that should be treated in the code

```
>> str2num('1a')
ans =
    []
```

```
>> str2num('1+1j')
```

```
>> str2num('1 +1j')
```

```
>> str2double('[1 2 3 pi]')
ans =
    NaN
```

```
>> str2num('1+1j')
ans =
    1.0000 + 1.0000i

>> str2num('1 +1j')
ans =
    1.0000 + 0.0000i    0.0000 + 1.0000i
```


Strings – number conversion #2

- quite often it is needed to convert numerical result back to a string

```
>> num2str(pi)
>> num2str(pi, 10)
```

```
>> disp(['the value of pi is: ' num2str(pi, 5)]);
```

- for listing purposes it is advantageous to use the function `sprintf`
 - it enables to control output format in a better way

```
>> st = sprintf('the value of pi is: %0.5f\n', pi);
>> st
```

Strings – other conversions

- among others there are other functions available

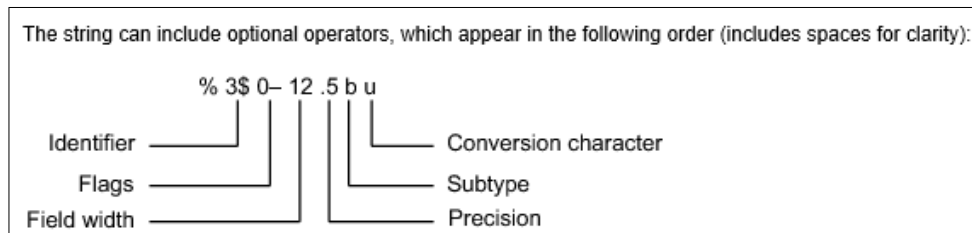
Function	Description
<code>int2str</code>	convert integer to text; in the case the input parameter is not an integer its value it is rounded first
<code>mat2str</code>	converts matrix to string
<code>hex2num, num2hex</code>	converts hexadecimal number of type <code>char</code> to a number (and vice versa)

- e.g.

```
>> mat2str(magic(3))
```

Strings – formatting

- function `sprintf` generates a string with given formatting
 - for more see `>> doc sprintf`
 - alternatively, `disp(sprintf(...))`



- function `fprintf` writes string
 - on a screen (`fid = 1` or `2`)
 - in a file (`fid` to be obtained e.g. using function `fopen`, see later)

```
>> st = sprintf('the value of pi is: %2.3e\n\n', pi);
>> fprintf(st) % or directly fprintf('...', pi);
```

```
>> fprintf(fid, st)
```

- create following strings using `sprintf` help:

- I.

```
ans =  
Value of pi is: 3.14159, value of 5*pi is: 15.70796  
>>
```

- i.e. both numbers are displayed with five digits accuracy

- II.

```
ans =  
This is 50%  
  
>>
```

- i.e. display percent sign, the expression contains 3 line spacings

- III.

```
ans =  
This is a measurement set: test_A  
>>
```

- i.e. insert variable into the string, value of which is `'test_A'` string

- think about the differences between `disp` a `fprintf` (`sprintf`)
 - describe the differences
 - what function do you use in a particular situation?

- function `fprintf` (`sprintf`)
 - it is faster with putting strings together
 - it offers significantly better formatting options
 - it enables to work with functions like `warning`, `error`, ...
 - standard function for file writing

Type conversion (general) – a note

- Matlab determines data types by itself
 - and also performs type conversion if needed
- single / double precision: `single()` / `double()`
- if, however, a particular data type is required that was not assigned on creation of a variable, this variable can be type-converted :
 - function `cast`: performs type conversion, values are truncated as the case may be
 - function `typecast`: performs type conversion and keeps the size of the original variable from the memory point of view as well as the bit value
 - see Matlab documentation for more

Upper case / lower case characters

- `lower` converts all letters in strings to lower case

```
>> lower('All will be LOWERCASE')  
% ans =  
% all will be lowercase
```

- `upper` converts all letters in strings to upper case

```
>> str = 'all will be upper case';  
>> str = upper(str)  
% str =  
% ALL WILL BE UPPER CASE
```

- support of characters from Latin 1 character set on PCs
- other platforms: ISO Latin-1 (ISO 8859-1)
- ⇒ supports Czech accents

Strings – searching

- `strfind` finds a given string inside another
 - returns indexes (positions)
 - searches for multiple occurrences
 - is CaSe sEnSiTiVe
 - enables to search for spaces etc.

```
>> lookFor = 'o';  
>> res = strfind('this book', lookFor);  
res =  
      7      8
```


Strings – comparing

- two strings can be compared using function `strcmp`
 - the function is often used inside `if-else` / `switch-case` statements
 - the result is either `true` or `false`
 - it is possible to compare string vs. cell of strings or cell vs. cell

```
>> strcmp('tel', 'A')
>> strcmp('tel', 'tel')
>> strcmp('test', {'test', 'A', '3', 6, 'test'})
>> strcmp({'A', 'B'; 'C', 'D'}, {'A', 'F'; 'C', 'C'})
```

$$\left(\begin{array}{|c|c|} \hline \text{A} & \text{B} \\ \hline \text{C} & \text{D} \\ \hline \end{array} \right) == \left(\begin{array}{|c|c|} \hline \text{A} & \text{F} \\ \hline \text{C} & \text{C} \\ \hline \end{array} \right) = \begin{array}{|c|c|} \hline \mathbf{1} & \mathbf{0} \\ \hline \mathbf{1} & \mathbf{0} \\ \hline \end{array}$$

Strings – joining

- strings can be joined together using function `strjoin`
 - it is applicable to variables of type `cell`
 - separator is optional (implicitly a space character)

```
>> cl = {'A', 'B', 'C', 'D'}
>> strjoin(cl)
>> strjoin(cl, ',')
```

- `fullfile` connects individual folders into a file path
 - back slash (\) is inserted between individual items (Win)

```
>> folder1 = 'Matlab';
>> folder2 = 'project_one';
>> file     = 'run_process.m';
>> fpath = fullfile(folder1, folder2, file);
```

- replace invalid separator depending on platform
- will be used for exporting and work with GUI

```
>> cl = {'A', 'B', 'C', 'D'}
cl =
    'A'    'B'    'C'    'D'
>> strjoin(cl)
ans =
A B C D
>> strjoin(cl, ',')
ans =
A,B,C,D
```

```
fpath =
Matlab\project_one\run_process.m
```

Strings – string separation

- function `deblank` removes excess space characters from end of string
- function `strtrim` removes space characters from beginning and end of string
- if a string is to be split, function `strtok` is used
 - separator can be chosen arbitrarily


```
>> this_str = 'some few little little small words'
```

1 2 3 4 5 6




```
>> [token, remain] = strtok(this_str, ' ');
```

first separated
word



rest of string



Strings – string separation

- function `regexp` enables to search a string using regular expressions
 - syntax of the function is a bit complicated but its capabilities are vast!!
 - **Ex.:** search for all words beginning with 'wh' with vowels **a** or **e** after and containing 2 characters:

```
>> that_str = 'what which where whose';  
>> regexp(that_str, 'wh[ae]..', 'match')
```

- **Ex.:** search indexes (positions) where words containing **a** or **o** begin and end

```
>> that_str = 'what which where whose';  
>> [from, to] = regexp(that_str, '\w*[ao]\w*')
```

- for more details see `>> doc regexp` → Input Arguments
- in combination with above mentioned functions, typical tokenizer can be created

Strings

600 s ↑

- try out following commands and try in advance to estimate what happens ...

```
>> str2num('4.126e7')
>> str2num('4.126A')
>> D = '[5 7 9]';
>> str2num(D)
>> str2double(D)
>> int2str(pi + 5.7)
>> A = magic(3);
>> mat2str(A)
>> disp([15 pi 20-5i]);
>> disp(D);
>> B = 'MaTLaB';
>> lower(B)
```

```
>> C = 'cik cak cet ';
>> strfind(C, 'cak')
>> deblank(C)
>> [tok remain] = strtok(C, ' ')
>> [st se] = regexp(C, 'c[aeiou]k')
>> [st se] = regexp(C, 'c[ei][kt]')
>> regexp(C, '[d-k]')
>> fprintf('Result is %3.7f', pi);
>> fprintf(1, 'Enter\n\n');
```

```
>> disp([' Result: ' num2str(A(2, 3)) 'mm']);
>> fprintf(1, '% 6.3f%% (per cent)\n', 19.21568);
>> fprintf('Will be: %3.7f V\n', 1e4*(1:3)*pi);
>> fprintf('A=%3.0f, B=%2.0f, C=%1.1f\n', magic(3));
>> fprintf('%3.3e + %3.3f = %3.3f\n', 5.13, 13, 5+13);
>> fprintf(2, '%s a %s\n\n', B, C([1:3 5:7]));
```

Strings – comparing

300 s ↑

- function to compare strings (CaSe SeNsItIvE) is called `strcmp`
 - try to find a similar function that is case insensitive

```
>> strcmpi(string1, string2)
```

- try to find a function that is analogical to the above one (i.e. case insensitive), but compares first `n` characters only

```
>> strncmpi(string1, string2)
```

- think about alternatives to the `strcmp` function

```
>> isequal(string1, string2)
>> all(string1 == string2)
```

- remove all blank spaces from the following string
 - try to recollect logical indexing
 - or use an arbitrary Matlab function

```
>> s = 'this is a big book'
```

- utilization of position of blank space in ASCII table

- write a script/function that splits following sentence into individual words using `strtok`
 - display number of occurrence of string `'is'`
 - list the words individually including position of the word within the sentence (use `fprintf`)

```
clear; clc;  
sen    = 'This-sentence-is-for-testing-purposes-only.';  
...  
...  
...  
...  
...  
...  
...  
...  
...  
...  
...  
...  
...
```


- write a script/function that splits following sentence into individual words
- the problem can be solved in a more elegant way using function `textscan`
 - solution, however, is not complete (word order is missing)

eval – string as a command

- motivation:

```
>> st = 'sqrt(abs(sin(x).*cos(y)))';  
>> x = 0:0.01:2*pi;  
>> y = -x;  
>> fxy = eval(st);  
>> plot(x, fxy);
```

i.e. there is a string containing executable terms

- its execution is carried out by function `eval`
- applicable mainly when working with GUI (execution of commands entered by user, processing callback functions etc.)
- `eval` has certain disadvantages, therefore its usage is a matter of consideration:
 - block of code with `eval` is not compiled (slow down)
 - text inside the string can overwrite anything
 - syntax inside the string is not checked, it is more difficult to understand
- see function help for cases where it is possible to replace `eval`
 - e.g. storing files with serial number (`data1.mat`, `data2.mat`, ...)

- in some cases it is needed not only to carry out a command in form of a string but also to store the result of the command for later use
- function `evalc` („*eval with capture*“) serves this purpose

```
>> CMD = evalc(['var = ' num2str(pi)]);
>> CMD

CMD =

var =

    3.1416

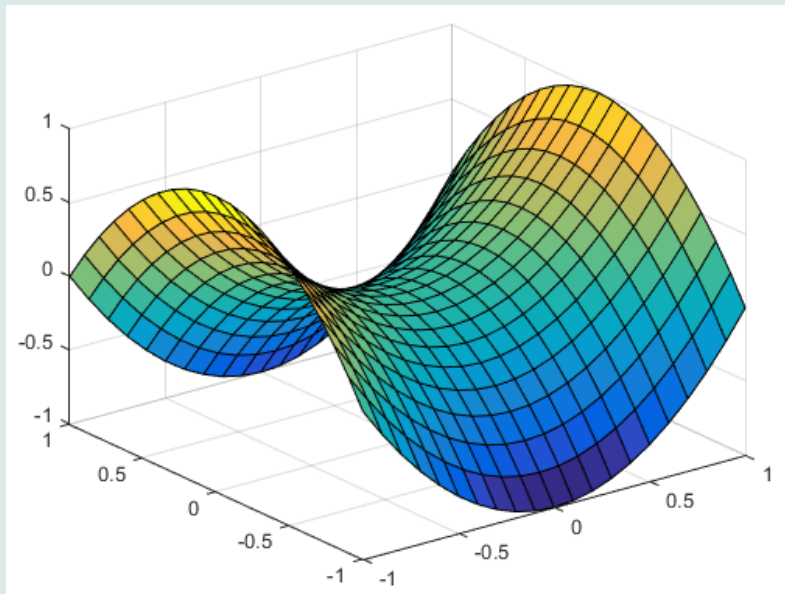
>> whos
Name      Size      Bytes  Class  Attributes

CMD      1x20      40     char
var      1x1       8      double
```

feval – evaluation of a handle function

- the function is used to evaluate handle functions
 - simply speaking, where `eval` evaluates a string there `feval` evaluates function represented by its handle
 - consider this task:

$$f(x, y) = x^2 + y^2, \quad x, y \in \langle -1, 1 \rangle$$



```
>> hFcn    = @(x,y) x.^2 - y.^2;
>> x      = -1:0.1:1;
>> y      = x;
>> [X, Y] = meshgrid(x, y);
```

```
>> fxy     = hFcn(X, Y);
>> surf(X, Y, fxy);
```

```
>> fxy     = feval(hFcn, X, Y);
>> surf(X, Y, fxy);
```

Function exist

- the function finds out whether the given word corresponds to existing
 - (=1) variable in Matlab Workspace
 - (=5) built-in function
 - (=7) directory
 - (=3) mex/dll function/library
 - (=6) p-file
 - (=2) m-file known to Matlab (including user functions, if visible to Matlab)
 - (=4) mdl-file
 - (=8) class(in the order of priority, return value in bracket)

```
>> type = exist('sin')      % type = 5
>> exist('task1', 'var')   % is the file task1 ...
>> exist('task1', 'dir')   % a variable / ...
>> exist('task1', 'file')  % directory / file?
```

What does your m-file depend on?

- in the case you compile your code, send it to colleagues etc., it is suitable to test whether they have all files and functions required
- `function matlab.codetools.requiredFilesAndProducts`
 - return user files and products necessary for evaluation of function/script
 - does not return files which are part of required products
- Ex.: dependencies of Newton's Method script

```
>> [fList, plist] = ...
matlab.codetools.requiredFilesAndProducts('newton_method_start.m')

fList =

    'D:\MTB\newton_method_start.m'    'D:\MTB\optim_fcn.m'

plist =

    Name: 'MATLAB'
    Version: '8.6'
    ProductNumber: 1
    Certain: 1
```

How to create a function – tips

- how to indicate that the given function / script is running?
 - try several possibilities below...

```
fprintf('START\n      ');  
for n = 1:100  
    fprintf(1, '\b\b\b\b%3.0f%%', n);  
    pause(0.05);  
end  
fprintf('\nEND\n');
```

```
T = ['/ ' '- ' '\ '];  
fprintf(2, 'START\n\n');  
for n = 1:100  
    fprintf(1, '\b%c', T(mod(n, 3)+1));  
    pause(0.05);  
end  
fprintf('\b');  
fprintf(2, 'END\n');
```

```
fprintf(2, 'START\n');  
for n = 1:100  
    fprintf(1, '*');  
    pause(0.05);  
end  
fprintf(1, '\n');  
fprintf(2, 'END\n');
```

- later we will see graphical options as well!

Matlab – file suffix

suffix	description
.fig	Matlab figure
.m	script / function / class
.mat	binary data file
.mdl, .slx	Simulink model
.mdl, .slxp	Simulink protected model
.mexa64, .mexmaci64, .mexw32, .mexw64	mex libraries
.mlappinstall	APP soubor – installer
.mlpkginstall	support package – installer
.mltbx	toolbox file – installer
.mn	MuPAD notebook
.mu	MuPAD code
.p	protected Matlab code

Discussed functions

<code>char, uint16, ...</code>	type conversion / creation of variables of given type	•
<code>single, double</code>	single / double precision	
<code>ischar, iscellstr</code>	test if input is character array / cell array of strings	
<code>int2str, mat2str, hex2num, num2hex</code>	conversion (integers – strings, hexadecimal – IEEE double)	
<code>str2double</code>	string to double	
<code>sprintf, fprintf</code>	String formatting, write to text file	•
<code>cast, typecast</code>	type conversion (not keeping / keeping underlying size)	
<code>lower, upper</code>	convert string to lowercase / uppercase	
<code>strfind, strcmp, strjoin, fullfile</code>	search, compare, join strings	•
<code>deblank, strtrim, strtok</code>	remove blank spaces, remove leading and trailing space, split string	•
<code>regexp, textscan</code>	search string (including regular expressions)	•
<code>eval, feval</code>	evaluate string / evaluate handle function	•
<code>exist</code>	check existence of variable	

Exercise #1, #2

450 s ↑

- find out how many spaces there are in the phrase „*how are you?*“
 - look in this lecture / help and find out a suitable function
 - utilize logical indexing
- convert following string to lowercase and find number of characters

```
>> st = 'MATLAB is CaSe sEnSiTiVe!!!';
```

Exercise #3

300 s ↑

- create a function to calculate volume, surface area and space diagonal of following bodies: cuboid, cylinder
 - the main function `main.m` contains verification of input variables (type, size) and checking whether user wants to calculate cuboid (parameters `'cuboid'`, `a`, `b`, `c`) or cylinder (`'cylinder'`, `r`, `h`)
- sub-functions `cuboid()` and `cylinder1()` calculate required parameters

```
function [V, S, u] = main(gType, a, b, c)
% decision making
% call functions
end

function [V, S, u] = cuboid(a, b, c)
% ... code
end

function [V, S, u] = cylinder1(r, h)
% ... code
end
```

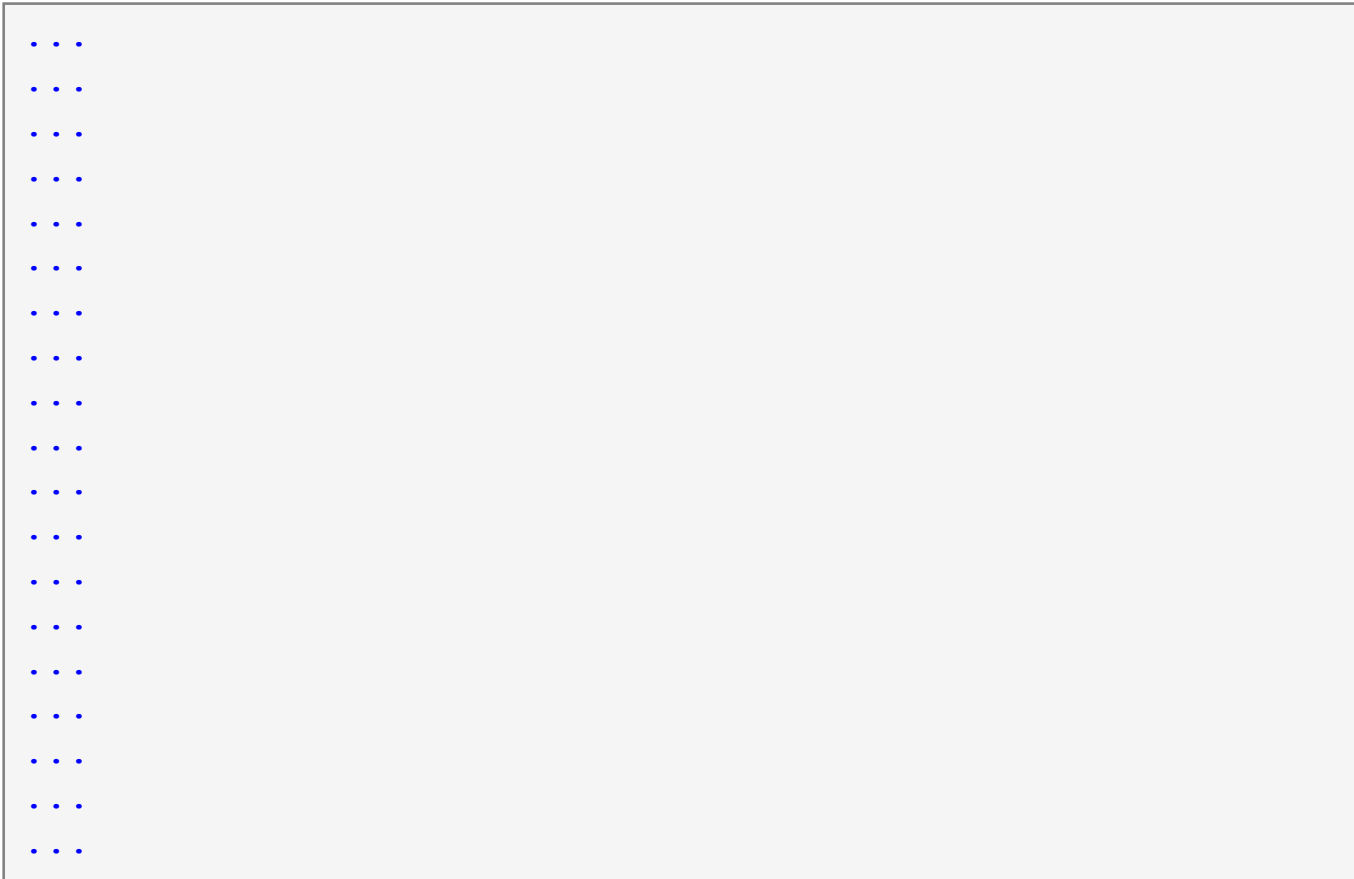

Exercise #4

600 s ↑

- create so called tokenizer (text analyzer), that
 - reads a text input `str` entered by user using function `input`
 - reads separator `sep` (attention, space requires some care!!)
 - split `str` in individual parts depending on `sep`
 - store individual parts separately in a variable of type `cell`
 - analyze how many vowels (a/e/i/y/o/u) each individual word contains, store this number and display it together with list of all individual words
 - all commands in the whole script / function have to be terminated with a semicolon!

Exercise #4

- create a tokenizer (text analyzer)
 - solution using `strtok`



Exercise #4

- improved solution using `strsplit`

```

...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...
...

```

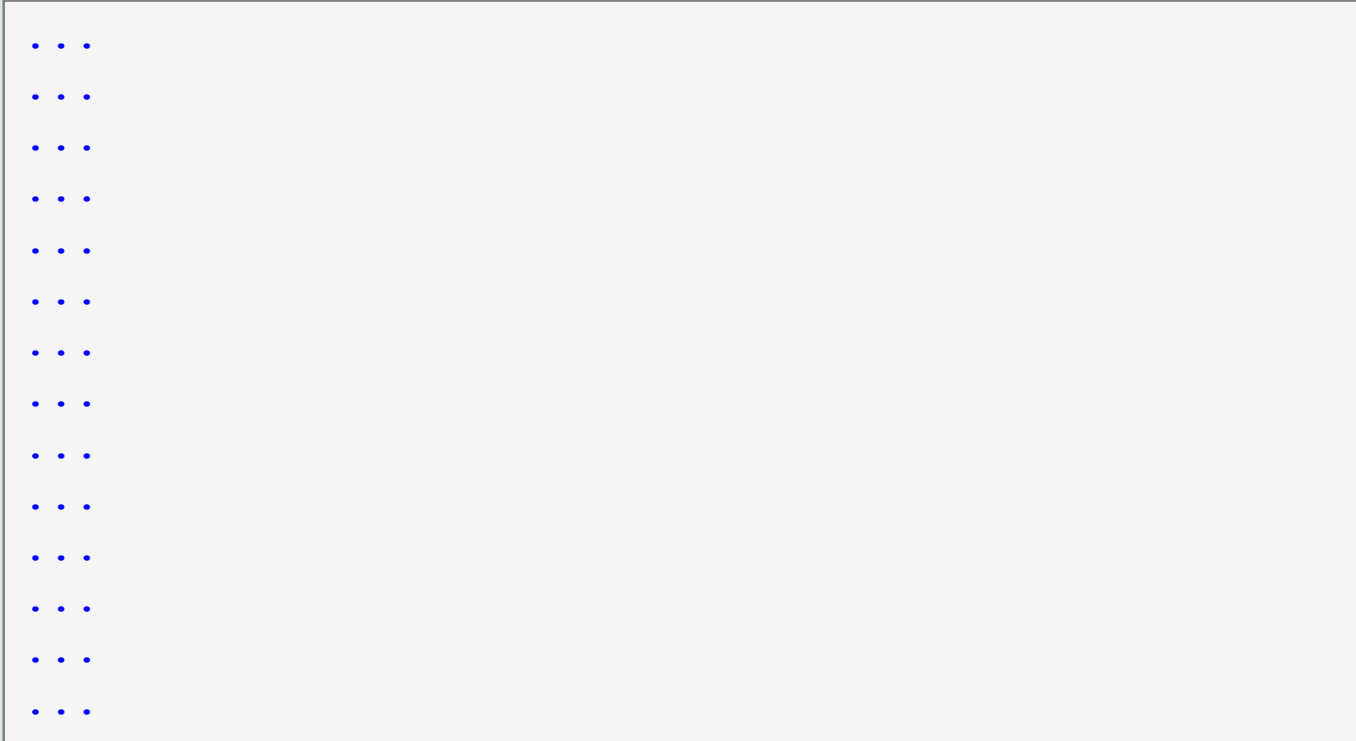
Exercise #5

600 s ↑

- try to create simple unit convertor, length x is given in `'mm'`, `'cm'`, `'in'`, `'inch'` (variable `units`), length in inches can be marked as `'in'` or `'inch'`. Length will be transformed into [mm] according to entered unit string.
 - what decision making construct are you going to use?
 - add a statement from which unit the length was converted and what the result is

```
x      = 15;  
units = 'in';  
% add the rest
```


Exercise #5



Unit conversion – more elegant way

- use data type `struct` and its properties
 - individual arrays in the structure can be indexed using variables of type `char`

```
function result = convertLength(in_val, in_unit, out_unit)

% supported units for conversion
conversion.in    = 1e4/254; % en.wikipedia.org/wiki/Imperial_units
conversion.inch  = conversion.in;
conversion.mm    = 1e3;
conversion.cm    = 1e2;
conversion.m     = 1;

% are the units supported?
if ~isfield(conversion, in_unit)
    error('convertor:nonExistentUnit', ['Unknown unit: ' in_unit]);
end

% calculation
result = in_val * conversion.(out_unit) / conversion.(in_unit);
```

Thank you!



ver. 11.1 (15/04/2019)
Miloslav Čapek, Pavel Valtr
miloslav.capek@fel.cvut.cz
Pavel.Valtr@fel.cvut.cz

Apart from educational purposes at CTU, this document may be reproduced,
stored or transmitted only with the prior permission of the authors.
Document created as part of B0B17MTB course.

