Further data types

Types of data

In Matlab®, the nature of any data is given by its *typology*

Logical values *(0 or 1)*	`logical`
Floating-point	32 bit 8bytes (64 bit) `single`, `double` numbers
Integers	`int8`, `int16`, `int32`, `int64` [-128,127] [-32768,32767]
Unsigned integers	`uint8`, `uint16`, `uint32`, `uint64` [0,256]
Character *(16 bit)*	`char`
Sequence of characters	`string`
Varying data type	`cell`
Matrix *(named row and columns)*	`table`
Data structure with fields	`struct`
Pointer to a function	`function_handle`
Other data types *(not seen here)*	`object`, `map`, `graphics`
	`handle`, `time series`, ...

Typology of data

To check the default data type assigned by Matlab® to your variable, use the command `class`

```
>> a= 1==0;
>> class(a)
>> b=rand(5);
>> class(b)
```

```
>> a= [1.5,2.4,3.0];
>> class(a)
>> b=[1,2,3];
>> class(b)
```

To change the type of any variable, convert it with the function named as the wished type (see the above table for a list)

```
>> a=1.5; b=int8(a)    % Converts a float to an integer
>> b, class(b)         % The float has been floored
>> char(1:1000)        % Converts to a vector of chars
```

Combination of different data typologies

Combining data of different typologies through operators may be possible. Matlab® converts the data into the type it thinks the most suitable. However, unexpected result can occur

```
>> class(b*2.7)
>> class(b*single(2.7))
```

```
>> 2*"2"
>> 2*'2'
```

Similarly, a data conversion may involve several typology changes, leading to hidden typologies combination that may also yield errors

```
>> a="3"; b=int8(a)   % Leads to an error
>> int8('3')          % Does not map to 3
>> int8('2.3')        % Gives out an array
```

Combination of different data typologies

Combining data of different typologies through operators may be possible. Matlab® converts the data into the type it thinks the most suitable. However, unexpected result can occur

```
>> class(b*2.7)
>> class(b*single(2.7))
```

```
>> 2*"2"
>> 2*'2'
```

Similarly, a data conversion may involve several typology changes, leading to hidden typologies combination that may also yield errors

```
>> a="3"; b=int8(a)   % Leads to an error
>> int8('3')          % Does not map to 3
>> int8('2.3')        % Gives out an array
```

Characters

A way to define variables containing non-numeric data is to use characters of type `char`

```
>> c2 = 'q', c1 = '@'
```

A sequence of characters is a character array *(formerly character string)*, which is a $1 \times n$ matrix of type `char`

```
>> s='hello'
>> class(s)
```

```
>> size(s)
>> s(2)
```

The concatenation of character strings is done as for vectors by

```
>> s2 = [s ' world']
>> s2(1)
```

```
>> s=['hallo', 'world'];
>> s(1), s(2)
```

Characters and strings

Strings

A string is a container for text, defined by double quotes `"`

```
>> s = "hello"
>> s
>> s(1)
```

A string array is defined as follows

```
>> s = ["hello", "world"];
>> s(1)
>> s(2)
```

Conversion between characters' array and string

To convert a string into a character array or an array of characters into a string, use the functions named as the desired type

```
>> z = char(s)
```

```
>> t = string(z)
```

Conversion between strings numbers

To convert a number into a string and *vis-versa*, there exists in-built conversion function

```
>> num2str(1.234)
>> str2double('1.234')
```

A default format is used in the string creation, while the conversion to a number is done up to machine precision

Strings' formatting

Informing the user of the results leads to create meaningful sentences or export structured data using *string formatting*

```
>> sprintf('The number is %f or %d.\n', 1.234, 5)
>> sprintf('The number is \t %.1f.', 1.234)
>> sprintf('The number is %f. ', [1,2,3,4])
```

The function `sprintf` also allows you to concatenate sentences. To replace a part of a sentence, use the function `strrep`

```
>> sprintf('This is %d concatenated %s', "string", 1)
>> strrep('A Test', 'Test', 'Toast')
>> strrep('A Test', "Test", 'Toast')
```

A small aside: date and time

Date representation

Operations on date and time are made easy in Matlab® with predefined variables storing the time in a `double`, representing the amount of previous days since the 0. January 0

```
>> now
>> today
```

```
>> now-today
>> date
```

The vector of doubles containing the year/month/day/hour/minute/seconds is given by `datevec`

```
>> datevec(now)
```

Date conversion to string

A conversion from the date's double representation to a human readable string, is given by the commands `datestr`.
The string is automatically formatted

```
>> datestr(now)
>> datestr(today)
>> datestr(0)
```

The reverse operation is given by the command `datenum`

```
>> datenum('2-Mar-1973')
>> datenum('2.3.1973 09:47', 'dd.mm.yyyy HH:MM')
```

Structures' definition

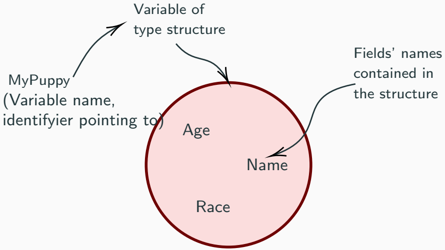
A *structure* is a type of variable that contains different fields, useful to group values that characterise a single entity

MyPuppy

Name = Caramel

Age = 6

Race = Rottweiler



A structure with initial valued fields is defined with the command `struct`. Defining a structure on the fly is also possible

```
>> S1 = struct("Field1","Value1","Field2","Value2")
```

Structures' syntax

The fields can be set and grabbed through `( structname).(fieldname)`

```
>> calib = struct                    % Decalres the structure
>> calib.focal_length = 1.2          % New field + given value
>> calib.date = now                  % Fields of any type
>> calib.visual.color = "Red"        % Nested structure
>> fieldnames(calib)
```

As usual, the size of the structure's array is automatically adjusted

```
>> calib(3).focal_length=1.5
>> size(calib)
```

Containers' Map

Maps' definition and value's access

A `map` is an object from the package `containers` that associates values to unique keys, comparable to `dict` in python

```
>> % Create a Map object that contains rainfall data
>> keySet = {'Jan','Feb','Mar','Apr'};
>> valueSet = [327.2 368.2 197.6 178.4];
>> M = containers.Map(keySet,valueSet)
```

Retrieving the stored data is done through specific methods

```
>> M.keys()
>> M.length()
>> M.values()
```

How much was the rainfall in March?

Containers’ Map

Maps’ definition and value’s access

A `map` is an object from the package `containers` that associates values to unique keys, comparable to `dict` in python

```
>> % Create a Map object that contains rainfall data
>> keySet = {'Jan','Feb','Mar','Apr'};
>> valueSet = [327.2 368.2 197.6 178.4];
>> M = containers.Map(keySet,valueSet)
```

Retrieving the stored data is done through specific methods

```
>> M.keys()
>> M.length()
>> M.values()
```

How much was the rainfall in March?

→ `>> M('Mar')`

Cell arrays' definition

Cells arrays are similar to matrices of type `cell` whose fields can contain any data typology (comparable to `list` is python)

```
>> s={ [1 2 3], 'example'; 2, 1==0 }
>> class(s), size(s)
```

Extracting parts of a cell array is done as usual, but the obtained data type is `cell`. Accessing the content of a cell itself is done with curly brackets `{}`

```
>> a=s(2,1)
>> class(a) % Is a cell
```

```
>> a=s{2,1}
>> class(a)
```

Cell arrays' specificities

Using a comma-separated list in the definition of a cell array is equivalent to separate each field with a comma

```
>> s2={1,2,[3,4]}
>> [s2{:}]
```

Accessing several parts of cell array is done as for any array, but returns two distinct quantities

```
>> s{1, 1:2}
```

What outputs from?

```
>> class(s{1,1:2})
```

Cell arrays' specificities

Using a comma-separated list in the definition of a cell array is equivalent to separate each field with a comma

```
>> s2={1,2,[3,4]}
>> [s2{:}]
```

Accessing several parts of cell array is done as for any array, but returns two distinct quantities

```
>> s{1, 1:2}
```

What outputs from?

```
>> class(s{1,1:2})
```

→ Error: no constructor

Exercise

1. Display in the prompt a string of the form:

   The value of x^2 at 2 is 4.
   The value of x^2 at 2.1 is 4.41.

   for x ranging from 1 to 3.
   Hint: a new line is created with `\n`

2. How many days lasted the first world war (28-Jul-1914, 11-Nov-1918)?

3. What does `mat2cell` do?

matrix

B = mat2cell(A, [1 2], [2 1 1])

row-dim col-dim

cell → B {1×2} {1×1} {1×1}
{2×2} {2×1} {2×1}

$$A = \begin{bmatrix} 2 & 3 & 5 & 7 \\ 11 & 2 & 5 & 7 \\ 6 & 7 & 7 & 8 \end{bmatrix} \rightarrow B: \; \{2, 3\} \; \{5\} \; \{7\} \quad \left\{\begin{matrix} 11 & 2 \\ 6 & 7 \end{matrix}\right\} \; \left\{\begin{matrix} 5 \\ 7 \end{matrix}\right\} \; \left\{\begin{matrix} 7 \\ 8 \end{matrix}\right\}$$

; celldisp(B)

Exercises: Further data types

Exercise

4. Guess the type and value of the following expressions. Check it then in the prompt.

   (c)
   ```
   >> int8(255) + 1;
   >> int32(int8(255) + 1) * 5;
   ```

   (d)
   ```
   >> logical(255) + 1;
   >> int64(double(int64(2)^63)-1) - int64(2)^63
   ```

Functions' inputs and outputs

Keywords controlling the function's arguments

Allowing a function to be called with a various number of arguments requires the use of the keywords `varargin` and `varargout` in the function's header

```
function varargout = FlexibleFunction(varargin)
    ....
end
```

If we pass N arguments to a function and retrieve M outputs, `varargin` and `varargout` are $1 \times N$ and $1 \times M$ **cell arrays**, respectively

- The numbers of arguments are thus not known *a-priori*.
- The keywords `nargin`, `nargout` contain a negative value

Variable functions’ arguments

Variable number of inputs

Define and test the following function on various number and types of input arguments, all stored in the cell array `varargin`

```
function [ ninp ] = varargin_function(varargin)
   % This function displays all the inputs

   ninp=size(varargin,2);

   for i=1:ninp
       disp(varargin{i})
   end
end
```

Variable number of outputs

Define and test the following function on various number and types of outputs, returned in the cell array `varargout`

```
function [varargout] = varargout_function()
    % This returns as many outputs as called
    for i=1:nargout
      if mod(i,2)==0
        varargout{i}=i^2;
      else
        varargout{i}=char(100+i);
      end
    end
end
```

[a, b, c] = varargout_function()
a = 'e'
b = 4
c = 'g'

Variable functions' arguments

Input strings' specificity

If a function takes strings as inputs, those commands are equivalent

```
>> varargin_function a b c
>> varargin_function 'a' 'b' 'c'
>> varargin_function('a' ,'b', 'c')
```

This behaviour will break on any other kind of input data

1. The following is a mistake... why?

```
>> sin pi
```

2. Does this work?

```
>> disp hello
>> disp hello world
```

Variable functions' arguments

Input strings' specificity

If a function takes strings as inputs, those commands are equivalent

```
>> varargin_function a b c
>> varargin_function 'a' 'b' 'c'
>> varargin_function('a' ,'b', 'c')
```

This behaviour will break on any other kind of input data

1. The following is a mistake... why?

```
>> sin pi
```

2. Does this work?

```
>> disp hello
>> disp hello world
```

→ pi is understood as a char, incompatible with the function `sin`

Anonymous function as an argument

If a function takes strings as inputs, those commands are equivalent

```
% Calling the function
>> g = @(t) cos(t)+2
>> x = 2; y = 4;
>> w = Func(g,x,y)
```

```
% Function's definition
function z = Func(f,x,y)
    z = f(x) + y
end
```

It is also possible to pass Matlab®'s predefined functions and use anonymous functions

```
>> g = @(f, t) cos(t)+2*f(t);
>> x = 2;
>> w = g(@sin,x)
```

In-file function as an argument

It is also possible a self-defined function which is stored in an external file as an argument. However, one has to point to the function's identifier by using the symbol `@`

```
% Calling the function
>> x=2; y=4;
>> w=Func(@L1_power8,x,y)
```

```
% Function's definition
function z = Func(f,x,y)
    z = f(x) + y
end
```

Note: Be careful to the *path* where the function is stored, Matlab® may not find it automatically