```
>> Lecture 4
>>
>>              -- Functions
>>
```

Motivation

- A large and complicated problem would be conquered by solving its subproblems.
- So the first step is problem decomposition, that is, separating tasks into smaller self-contained units.
- This is also beneficial to code reuse without copying the codes.
- Note that bugs propagate across the program when you copy and paste the codes.

Function

- A function is a piece of program code that accepts input arguments from the caller, and then returns output arguments to the caller.
- In MATLAB, the syntax of functions is similar to math functions,

$$y = f(x),$$

where x is the input and y is the output.

User-Defined Functions

- We can define a new function as follows:

```
function [outputVar] = function_name(inputVar)
    % What to do.
end
```

- This function should be saved in a file with the function name!
- Note that the input/output variables can be optional.

Example: Addition of Two Numbers

```
function z = myAdd(x, y)
    % Input: x, y (any two numbers).
    % Output: z (sum of x and y).
    z = x + y;
end
```

- It seems bloody trivial.
- The truth is that the plus operator is actually the function **plus**.[1]
- Also true for all the operators like $+$.

[1] See `https://www.mathworks.com/help/matlab/ref/plus.html`.

Variable-length Input Argument List[2] (Optional)

- We can know the number of input arguments for the function executed by **nargin**.
- **varargin** is an input variable in a function definition statement that enables the function to accept any number of input arguments.
  - It must be declared as the last input argument and collects all the inputs from that point onwards.
- The variable **varargout** is a special word similar to **varargin** but for outputs.

[2] See `https://www.mathworks.com/help/matlab/ref/varargin.html`.

Example

```
function ret = myAdd(varargin)

    switch nargin
        case 0
            disp("No input.");
        case 1
            ret = varargin{1};
        case {2, 3}
            ret = sum([varargin{:}]);
        otherwise
            error("Too many inputs.");
    end

end
```

Variable Scope

- Variables in a function are known as local variables, existing only for the function.
- These variables are wiped out when the function finishes its task.
- You may trace the data flow in the program by using the debugger.[3]
  - Let's set some breakpoints!!!

[3] See `https://www.mathworks.com/help/matlab/matlab_prog/debugging-process-and-features.html`.

Example

```
clear; clc;

x = 0;
for i = 1 : 5
    addOne(x);
    disp(x); % output ?
end
```

```
function addOne(x)
    x = x + 1;
end
```

Function Handles & Anonymous Functions

- Anonymous functions are used once and not written in the standard form of functions, for example,

```
f = @(x) x .^ 2 + 1 % f is a function handle.
```

- However, they contain only single statement.
- Besides, we use function handles[4] to handle functions.
- This is also called lambda expressions.
- You can also assign an existing function to a handle, for example,

```
g = @sin
```

[4] You may refer to `https://en.wikipedia.org/wiki/Function_pointer`. The truth is that every function name is an alias of the function address!

More Examples[5,6,7]

```
function y = parabolicFunGen(a, b, c)
    y = @(x) a * x .^ 2 + b * x + c;
end
```

```
function y = getSlope(f, x0)
    eps = 1e-9;
    y = (f(x0 + eps) - f(x0)) / eps;
end
```

```
function y = differentiate(f)
    eps = 1e-9;
    y = @(x) (f(x + eps) - f(x)) / eps;
end
```

[5] Thanks to a lively class discussion (MATLAB244) on August 22, 2014.
[6] Contribution by Ms. Queenie Chang (MAT25108) on March 18, 2015.
[7] Thanks to a lively class discussion (MATLAB260) on September 16, 2015.

Vectorization (Revisited)

- We can apply a function to each element of array by **arrayfun**.[8]

```
B = arrayfun(@(x) 2 * x, A) % Equivalent to 2 * A.
```

- **cellfun** is similar to **arrayfun** but applied to cells.[9]

```
>> data = {"NTU", "CSIE", [], "MATLAB"};
>> isempty(data) % Output 0.
>> cellfun(@isempty, data) % Output 0 0 1 0.
```

[8] See https://www.mathworks.com/help/matlab/ref/arrayfun.html.
[9] See https://www.mathworks.com/help/matlab/ref/cellfun.html.

Error and Error Handling

- You can issue/throw an **error** if you do not allow the callee for some situations.

```
if bad_condition
    error("So wrong."); % Interrupt the normal flow.
end
```

- As an app programmer, you should use a try-catch statement to handle errors.

```
try
    % Normal operations.
catch
    % Handler operations.
end
```

Example: Combinations

- For all nonnegative integers $n \geq k$, $\binom{n}{k}$ is given by

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

- Note that **factorial**(n) returns $n!$.

```
clear; clc;

n = input("n = ? ");
k = input("k = ? ");
y = factorial(n) / (factorial(k) * factorial(n - k))
disp('End of program.');
```

- Try $n = 2, k = 5$.
- However, **factorial**(-3) is not allowed!
- The program is not designed to handle this error, so it is interrupted in Line 5 and does not reach the end of program.
- Add error handling to the program:

```
1  clear; clc;
2
3  n = input("n = ? ");
4  k = input("k = ? ");
5  try
6      y = factorial(n) / (factorial(k) * ...
           factorial(n - k))
7  catch e % capture the thrown exception
8      disp("Error: " + e.message); % show the message
9  end
10 disp("End of program.");
```