Matlab
Matlab

- Matlab, which is short for Matrix Laboratory, is a numerical computing package
- Matlab is a useful tool for solving many modern day engineering and scientific problems (so is Python!)
  - E.g. Matlab has toolboxes for aerospace, process control, robotics, vehicle dynamics etc.

- Why Matlab in ENGG1811?
  - A useful package for engineers
  - To show you that you can be “multilingual” in computer languages
Be computer multilingual

- You will probably see and use many different computer languages over your life
- Good news: Once you’ve mastered one computer language (in your case Python), it is not difficult to pick up a similar computer language

- Transferrable concepts
  - Assignment, for-loops, while-loops and many others
- Python and Matlab differ mainly in their syntax
  - E.g. Python uses ** for exponentiation but Matlab uses ^
  - It can trip you (or even your lecturer!) up sometimes but the good thing about computer languages is that, unlike human languages, you do not have to listen and reply in real-time, so just keep calm and correct the errors
Online/offline Resources

• Documentation at mathworks.com

• Tutorials provided by MathWorks

• The Matlab Onramp interactive tutorial is a good starting point

• Documentation that comes with Matlab. Accessible with the *help button* (more on this later)
Matlab desktop

The Matlab desktop has four main windows

- Select file for editing
- Enter commands
- View current variables
- Review previous commands

Ribbon, or menu + toolbars (depends on version)
Matlab desktop, continued

- Other windows that appear when required are
  - Editor window, where script and function files are edited
  - Figure window, where the results of plot or other display commands are shown
  - Path window, for specifying folders where you keep your Matlab files
  - Help window

- Windows can be docked or undocked as you like
Matlab command window

• This is similar to the Python console in Spyder3
• We will use the Matlab command window to enter some arithmetic expressions
Arithmetic operators and others

- Arithmetic operators:
  - +,-,*,/
  - ^ for exponentiation

- No arithmetic operators for integer division nor remainder
  - Need to use Matlab functions

- % (percentage sign) is the comment sign

- You can use either , (comma) or ; (semi-colon) to separate commands
  - , means to display the result
  - ; means to suppress the display
Note: Precedence is not easy to remember. It can vary from language to language. If in doubt, check, use parentheses or use a simple example to figure out.
Command window (2)

```matlab
>> % Using ; (semi-colon) to suppress the output
>> b = 3*4-2; c = -2^5, d = 5^-2;

    c =  
       -32

>> % Matlab variable names are case-sensitive
>> abc = 20

    abc =  
       20

>> aBc
Undefined function or variable 'aBc'.
```
Some special numbers

Some special numbers

> % Special numbers
> pi
ans =
  3.1416

> i % i is defaulted to square root of -1
ans =
  0.0000 + 1.0000i

> j % j is also defaulted to square root of -1
ans =
  0.0000 + 1.0000i

> (2 + 3*i) * (1 - 2*j)
ans =
  8.0000 - 1.0000i

- Note that if you use pi, i and j as variable names, you will over-write the default value for the Matlab session.

- Some programmers avoid using i and j in their Matlab code. They use ii and jj instead.

Complex numbers are easy to use in Matlab
Character arrays

- In python we enclosed strings using either single or double quotes.
- MATLAB uses single and double quotes to define two different data types.
- Single quotes are used to define character arrays and double quotes are used to define strings. We will be using character arrays only in this module so we will only be using single quotes.
- Let me define a variable `ca` and assign it to a character array.

```plaintext
>> ca = 'We are young and free.'
ca =
   'We are young and free.'
```
Character arrays: concatenation

• To concatenate character arrays together we need to place them inside square brackets in the order we want them to concatenate, separated by commas.

• Example:

```python
>> ca1 = 'The area is '; ca2 = '20'; ca3 = ' square meters.';
>> ca_combined = [ca1, ca2, ca3]
ca_combined =
    'The area is 20 square meters.'
```
Script

- You can put multiple Matlab commands in a file, which is called a script.
- These commands will be executed one by one.
- We will write a script (see the next slide) and run it:
  - We will name the script as script_ex.m.
  - Note the .m extension for files from Matlab.
% Compute the area of a triangle with lengths 6, 6, 4
% Note: It's an isosceles triangle

% Define the lengths
a = 6; b = 4;

% Compute the height
h = sqrt(a^2 - (b/2)^2);
% Calculate the area
area = 0.5 * h * b;
% Print the area on screen
disp(['The area is ',num2str(area)])
% Note: disp(['The area is ', area]) – won’t work
% Will explain 2 slides later later
Math and built-in functions

• Matlab has many built-in functions:
  – E.g. sqrt(), exp(), sin(), sind() – Too many to list
• Look up the documentation when needed
  – There are two ways to do that

• You can search

• You can browse.
  – Click on the documentation button
  – Follow this sequence of clicks in the main panel:
    • MATLAB, Mathematics, Elementary Maths, Trigonometry
• You will see a rich collection of functions
Converting numbers to character arrays

- The `disp()` function in Matlab is similar to `print()` in Python.

```matlab
area = 0.5 * h * b;
disp(['The area is ',num2str(area)])
% Note: disp(['The area is ',area]) – won’t work
```

- The variable area contains a number, need to convert it to a character array
  - Matlab requires explicit conversion
  - Python `print()` does automatic conversion

- Matlab has two functions to convert numbers to character arrays
  - `int2str()` for converting an integer
  - `num2str()` for converting a number
**Side notes on num2str() and int2str()**

- You may ask the name of the function `num2str()` ends with `str` and isn’t it short for string.
- Hmmm, this is the confusing bit.
- In the older versions of MATLAB, character arrays and strings were interchangeable.
- But now, character arrays and strings are two different data types.
- The function `num2str()` was created when character arrays and strings were the same, so it was suitable at that time because it would output a string.
- However, for today’s version, the function `num2str()` converts a number into a character array. Well, that’s a bit of a history.
- Same story for `int2str()`
Comparison and Boolean operators

- Comparisons: >, >=, <, <=, ==, ~=
  - ~= means not equal to (Python uses !=)

- Boolean operators
  - && for and
  - || for or
  - ~ for not

- Matlab uses
  - Logical 1 to denote true
  - Logical 0 to denote false

- Examples in logical_ex.m
  - Note: % to divide code in sections
Selection using if

- Same principle as Python but syntax differs slightly
  - Indentation is optional but **highly recommended** for readability
  - The `end` keyword to signify the completion of the if-block
- Each `if` keyword must have a corresponding `end` keyword

```matlab
if boolexpr_1
    statements_1
elseif boolexpr_2
    statements_2
elseif ...
else
    statements_n
end
```

% example (max of a and b)
```
if a < b
    bigger = b;
else
    bigger = a;
end
```

File: selection_ex.m
Arrays (1)

- Matlab stores data in arrays
  - if the data has one dimension, it’s a vector
  - if it has two dimensions, it’s a matrix
  - An $m \times n$ array has $m$ rows and $n$ columns

```matlab
>> vrow = [ 1, 1.7, -3.5, 123 ]; % row vector 1x4
```

```matlab
>> size(vrow) % size of the array
ans =
   1   4
```

Example in `array_ex.m`
Arrays (2)

- A scalar variable is a 1x1 array

```matlab
>> a = 5.5;  % a scalar is an 1x1 array
```

```matlab
>> size(a)  % size of the array
ans =
    1     1
```

Example in array_ex.m
Rules for entering arrays

- Enter one row at a time, starting from the first row
- rows separated by *semicolons* or end-of-line
- elements within a row separated by *spaces* or *commas*
  - commas preferred for expressions, complex (1-i and 1 –i differ)
- can drop the [ ] for 1x1

Row 1

Then, Row 2

Then, Row 3
Array assignment (1)

- Array literal data is enclosed in square brackets
- Examples:

```matlab
>> vrow = [ 1 1.7 -3.5 123 ]; % row vector 1x4
>> vcol = [ 23; -8; 12.5]; % column vector 3x1
```

- Row vectors and column vectors are different:

<table>
<thead>
<tr>
<th>1</th>
<th>1.7</th>
<th>-3.5</th>
<th>123</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>-8</td>
<td>12.5</td>
<td></td>
</tr>
</tbody>
</table>

vrow is an 1x4 array

Example in `array_ex.m`

vcol is an 3x1 array
Array assignment (2)

- Examples:

```matlab
>> m = [1, 4 ; 2.3, -7 ; 2+5i, 1-2i]; % a 3x2 array
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>-7</td>
</tr>
<tr>
<td>2+5i</td>
<td>1-2i</td>
</tr>
</tbody>
</table>

Example in array_ex.m
Array assignment (3)

• Number of columns must be consistent

$$m1 = [1, 2, 3; 4, 5]$$

Error using vertcat
Dimensions of matrices being concatenated are not consistent.

• Be careful with space for complex numbers or expressions

$$m2 = [1, 4 ; 2.3, -7 ; 2+5i, 1-2i]; \quad \% \text{ OK}$$

$$m3 = [1, 4 ; 2.3, -7 ; 2+5i, 1-2i]; \quad \% \text{ error}$$

Matlab thinks this row has 3 elements

$$a = 5; \ b = 4;$$

$$m4 = [1, 4 ; 2.3, -7 ; 2+5i, a-b]; \quad \% \text{ OK}$$

$$m5 = [1, 4 ; 2.3, -7 ; 2+5i, a-b]; \quad \% \text{ error}$$
Accessing array elements (1)

- Use **round brackets** with subscripts separated by commas
- Index begins with **1**
  - Note: Python uses [ ] and index begins at 0

```matlab
>> m = [1, 4 ; 2.3, -7 ; 2+5i , 1-2i]; % entered earlier
>> m(3,2) % element at row 3, column 2
ans =
   1.0000 - 2.0000i
```

Example in `array_ex.m`
Accessing array elements (2)

- Use round brackets with subscripts separated by commas
  - row and column vectors use one subscript, matrices 2+

\[
\begin{array}{c}
\gg \, v\text{row}(3) \\
\text{ans} = \\
\quad -3.5000
\end{array}
\]

```
vrow
\begin{array}{cccc}
1 & 1.7 & -3.5 & 123 \\
vrow(1) & vrow(2) & vrow(3) & vrow(4) \\
vrow(1,1) & vrow(1,2) & vrow(1,3) & vrow(1,4)
\end{array}
```

Example in `array_ex.m`
Assigning to array elements

>> % m(3,2) – element at row 3, column 2
>> vcol(1) = m(3,2) % replace element
vcol =
    1.0000 – 2.0000i
   -8.0000
   12.5000

- Assigning to non-existent elements extends an array

>> vrow(7) = -1 % intervening elements become zero
vrow =
    1.0000   1.7000  -3.5000   123.0000    0    0  -1.0000

Example in array_ex.m
Transpose operator

- Transpose operator .\(^\prime\) (single quote) can be applied as a suffix to an array expression to exchange rows and columns
  - Important note:
    - \(^\prime\) is transpose
    - \(^\prime\) is transpose and complex conjugate

\[\begin{align*}
\text{>> } g &= [1 \ 2+i]; \quad \text{% row vector} \\
\text{>> } h &= g.^\prime \\
h &= \\
&\begin{bmatrix}
1 \\
2+i
\end{bmatrix}
\text{>> } h &= g' \\
h &= \\
&\begin{bmatrix}
1 \\
2-i
\end{bmatrix}
\end{align*}\]

Example in array_ex.m
Quiz

- The Matlab variable $m$ is a 7x1 array. The Matlab variable $p$ is a 10x13 array.
- You wish to assign the value in the 4-th row of $m$ to the element in row 3 and column 6 of $p$. Which Matlab statement(s) will allow you to do that:
  a) $p(10,13) = m(7,1)$
  b) $p(6,3) = m(1,4)$
  c) $p(6,3) = m(4,1)$
  d) $p(6,3) = m(4)$
  e) $p(3,6) = m(1,4)$
  f) $p(3,6) = m(4,1)$
  g) $p(3,6) = m(4)$
Array functions

• Built-in functions that inform you about arrays
  - `length` (elements in a vector or longest dimension of matrix)
  - `size` (returns an array of all dimensions)

```matlab
>> length(m)
an =
   3
>> size(m)
an =
   3  2
```

• Matrix functions `zeros`, `ones` and `eye` (identity)
  accept row and col sizes, or just one value if square

```matlab
>> eye(4);  % 4x4 identity matrix
>> ones(2,5); % 2x5 matrix filled with 1's.
>> zeros(size(m)); % 3x2 matrix filled with zeroes
```
Colon operator

- Colon operator generates all elements of an arithmetic progression
  
  \[ \text{first} : \text{increment} : \text{last} \]
  
  \[ \text{first} : \text{last} \quad (\text{default increment} = 1) \]

\[
\begin{bmatrix}
\text{seq} = \text{1:2:9} & \% \text{odd integers less than or equal to 9} \\
\text{ans} = \text{1 3 5 7 9} \\
\text{size(seq)} & \% \text{seq is an array} \\
\end{bmatrix}
\]

- Similar to numpy arange() and the :: numpy notation but note the following
  - Numpy ordering is \text{first, last, increment}
  - Numpy does not include \text{last}
Array indexing

- Matlab has a few different methods to index elements
- We will only look at
  - Indexing by subscripts
- There is also linear indexing and Boolean indexing but we won’t cover them
- Detailed examples in `indexing1_ex.m` and `indexing2_ex.m`
Vectorisation

- Matlab and numpy have a number of built in functions that allow you to do mathematical operations without any loops
  - This is called vectorisation

- Again, Matlab and numpy syntax are different
Some useful functions

- **sum**
- **cumsum**: cumulative sum
- **std**: standard deviation
- **var**: variance
- **prod**: product
- **mean**
- **median**
- **max, min**
- **mode**: most frequently occurred value
- **sort**

**Note:**
- These functions work differently for vectors and matrices.
  - The idea is similar, so we use sum as an illustration
- Some functions can take additional inputs or return more than one output. Check the documentation for details.
The sum function (1)

- $\sum$ function (1)
  - $\sum$ (a one dimensional array) is a scalar

Example in `sum_ex.m`

```matlab
v1 = [1, 2, 3]; % row vector
v2 = [-1; -2; 4]; % column vector

>> sum(v1)

6

>> sum(v2)

1
```
The sum function (2)

Example in sum_ex.m

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]

\( m = [1, 2, 3 ; 4, 5, 6]; \)  \( \% \) a 2-by-3 array

\[
\begin{bmatrix}
5 & 7 & 9
\end{bmatrix}
\]

\( \gg \text{sum}(m,1) \)  \( \% \) sum in 1st dimension

\( 5 \quad 7 \quad 9 \)

\( \% \) Note \( \text{sum}(m) \) is the same as \( \text{sum}(m,1) \)

\[
\begin{bmatrix}
15 \\
1
\end{bmatrix}
\]

\( \gg \text{sum}(m,2) \)  \( \% \) sum in 2nd dimension

15

1st dim -> 2nd dim

(1st dim, 2nd dim)
A common use of array is to store data.

Let $m$ be a matrix with 4 rows and 7 columns. The element $m(h,d)$ (in $h$-th row and $d$-th column) is the rainfall in the $h$-th quarter (i.e. a 6-hour duration) of Day $d$.

<table>
<thead>
<tr>
<th>Row 1, 1$^{\text{st}}$ quarter</th>
<th>Day 1</th>
<th>Day 2</th>
<th>Day 3</th>
<th>Day 4</th>
<th>Day 5</th>
<th>Day 6</th>
<th>Day 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 2, 2$^{\text{nd}}$ quarter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 4, 4$^{\text{th}}$ quarter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quiz: The sum function (2)

1) Find the total rainfall for each of the 7 days and store the answer in a vector

```
sum(m,1) or sum(m)
```

2) Find the total rainfall for all the 7 days

```
sum(sum(m))
```

Note: You need to sum up all the elements in the matrix. In Matlab, you need to use a function twice.

3) Find the total rainfall over days 3, 4, 5, 6 and 7

```
sum(sum(m(:,3:7)))
```
Elementwise operation (1)  

- Many maths functions, e.g. `sin()`, `exp()`, `log()`, `abs()`, `sqrt()`, apply to all elements in the array.

```matlab
>> t = [1, 2, 4, 8];  % 1x4 array
>> y = sqrt(t)       % y is also an 1x4 array
y =
    1.0000    1.4142    2.0000    2.6458
```

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1.4142</th>
<th>2</th>
<th>2.6458</th>
</tr>
</thead>
<tbody>
<tr>
<td>y(1)</td>
<td>=sqrt(t(1))</td>
<td>y(2)</td>
<td>=sqrt(t(2))</td>
<td>y(3)</td>
</tr>
</tbody>
</table>

Example in `ele1_ex.m`
Elementwise operation (2)

- Logical comparison too

```matlab
>> a = [1, 2, 4, 8, -1, 5];  % 1x6 array
>> b = (a >= 4)  % Note: ( ) added for readability. They aren’t needed
b =
    0     0     1     1     0     1

>> % Elementwise AND operator is &
>> % Elementwise OR operator is |
>> c = (a >= 4) | (a < 0)
c =
    0     0     1     1     1     1
```

Example in ele1_ex.m
Elementwise arithmetic

- Elementwise addition, subtraction, multiplication and division must be:
  - Between two arrays of the same size, or
  - Between an array and a scalar
- Matlab does not use numpy broadcasting

- The operators are +, -, .* (elementwise multiplication), ./ (elementwise right division), .\ (elementwise left division) and .^ (elementwise exponentiation)

Example in ele2_ex.m

- Note: In Matlab, * is for matrix multiplication and loosely speaking, / and \ are for matrix division
Example: for

- Sum up the elements in a vector $v$
  - You should use `sum()` but this is to illustrate the concept of for-loop

```matlab
v = [4, 3, -1, 8, 7, -2];
total = 0;
for ii = v
    total = total + v;
end
```

Examples in `for_ex.m`
Matlab functions

• You can define your own functions in Matlab
• Let us start with an example
Let me first show you a simple example of a script that does not contain a function.

This script has 4 lines of code.

The first two lines define two variables `edge1` and `edge2`.

Imagine that I have a rectangle whose dimensions are given by the values of `edge1` and `edge2`, then I can compute the area by multiplying the variables `edge1` and `edge2`.

```matlab
1 - edge1 = 7.6;
2 - edge2 = 4.3;
3 - area_of_rect = edge1 * edge2;
4 - disp(['Area is ', num2str(area_of_rect)])
```
Script with function (1)

- The script `calc_rect_area_func` has code from lines 1 to 5 and a function definition from line 7 onwards.
- You can see this script’s resemblance to a python script, except in python, we would define the function first then have code below it to call the function. (next slide)
- In Matlab scripts we have to define the function after all the lines of code outside of a function before the first function definition.

```matlab
1 - edge1 = 7.6;
2 - edge2 = 4.3;
3 - area_of_rect = calc_area_local(edge1,edge2);
4 - % Code line 3 calls the function and gives it the inputs.
5 - disp(['Area is ',num2str(area_of_rect)])
6
7 - function [area] = calc_area_local(length,width)
8 - area = length * width;
9 - end
```
Contrasting with Python

```python
def calc_area(length, width):
    area = length * width
    return area

edge1 = 4
edge2 = 6
area = calc_area(edge1, edge2)
print('Area is', area)
```

↑

In Python, function definition appears before the code calling it.
In this script we call the function in code line 3.

This is exactly the same as how python calls a function.

If you compare the two scripts calc_rect_area (Slide 46) and calc_rect_area_func (below), all we have done is to extract the area formula in line 3 of calc_rect_area and put it into a function in the calc_rect_area_func script.

```matlab
1  edge1 = 7.6;
2  edge2 = 4.3;
3  area_of_rect = calc_area_local(edge1,edge2);
  % Code line 3 calls the function and gives it the inputs.
4  disp(['Area is ',num2str(area_of_rect)])
5
6 function [area] = calc_area_local(length,width)

7       area = length * width;
8
9 end
```
Function (1)

- The function definition line starts with the function keyword, which is equivalent to the def keyword in Python.
- The function keyword should have a corresponding end keyword, which tells Matlab where the end of the function definition is.
- Note that the end keyword is sometimes not necessary for function definition, but for clarity, we recommend that you always use the end keyword.

```
function [area] = calc_area_local(length,width)
    area = length * width;
end
```
Function (2)

- In this example we assign what our function returns into the variable area. The variable area is the output.
- We recommend using the square brackets around the outputs even when there is only one output as it will avoid mistaking it for the function name.
- We specify the inputs enclosed within a pair of round brackets. For this example we have two inputs, length and width.

```
function [area] = calc_area_local(length,width)

    area = length * width;

end
```
function [area] = calc_area_local(length,width)

% This function calculates the area of a rectangle given its length and width
% Inputs:
%   length   Length of the rectangle
%   width    Width of the rectangle
% Outputs:
%   area     Area of the rectangle

    area = length * width;

end
Script and function files

- Matlab has scripts and function files.
- Function file has only function definitions
- Script file has code not inside any function. Script file can also contain functions.

```matlab
% Call calc_area_peri function
edge1 = 7.6;
edge2 = 4.3;
[area_2,peri_2] = calc_area_peri(edge1,edge2);
disp(['Area is ',num2str(area_2)])
disp(['Perimeter is ',num2str(peri_2)])
```

```matlab
function [area,peri] = calc_area_peri(length,width)
    area = calc_area(length,width);
    peri = calc_peri(length,width);
end

% The following is an example of local function
function p = calc_peri(l,w)
    p = 2*(l+w);
end
```
Function files (1)

- Function file names must match the name of the main function in the file exactly, including the casing of the letters in the name has to match too since MATLAB is case sensitive.
- The following picture shows the contents of the file calc_area_peri.m. Note that the file name matches the main function name `calc_area_peri` (which is short for calculates area and perimeter)

```matlab
function [area, peri] = calc_area_peri(length, width)

    area = calc_area(length, width);
    peri = calc_peri(length, width);

end

% The following is an example of local function
function p = calc_peri(l,w)
    p = 2*(l+w);
end
```
Function files (2)

- It is recommended not to have function or script file names the same as built-in Matlab functions because Matlab will throw a warning telling you not to, to avoid potential name conflicts.
- In python if you run a script with just a function in it, nothing will happen but check if the syntax is valid.
- In Matlab if you run a file with just a function in it, an error will appear in the command window telling you that there isn’t enough input arguments.
- This just means Matlab is asking for the input arguments as they are needed in the function.
Calling functions in file

- You need to make a separate script to call the function, as seen in line 4 of the script file run_calc_area_peri.m.
- The main reason why the filename and function name needs to be exactly the same is because when a function is called matlab scans the files in the current directory for it.

```matlab
% Call calc_area_peri function
edge1 = 7.6;
edge2 = 4.3;
[area_2,peri_2] = calc_area_peri(edge1,edge2);
disp(['Area is ',num2str(area_2)])
disp(['Perimeter is ',num2str(peri_2)])
```

```matlab
function [area,peri] = calc_area_peri(length,width)

    area = calc_area(length,width);
    peri = calc_peri(length,width);

end

% The following is an example of local function
function p = calc_peri(l,w)
p = 2*(l+w);
end
```
Input assignments

- We want to point out that like python when we call the function and feed it the inputs the variable names don’t have to be the same variable names in the function definition.
- As long as the order of variables are in the order that is intended by the function definition then the function will work as intended.
- So the value of edge1, 7.6 will be stored in the variable length in our `calc_area_peri` function. Similarly, for edge2.

```matlab
% Call calc_area_peri function
edge1 = 7.6;
edge2 = 4.3;
[area_2,peri_2] = calc_area_peri(edge1,edge2);
disp(['Area is ',num2str(area_2)])
disp(['Perimeter is ',num2str(peri_2)])
```

```matlab
function [area,peri] = calc_area_peri(length,width)
    area = calc_area(length,width);
    peri = calc_peri(length,width);
end
% The following is an example of local function
function p = calc_peri(l,w)
    p = 2*(l+w);
end
```
Local function

• Any function in a file that does not match the filename is called a local function.

• If we switch back to the `calc_rect_area_func` script (see below), the function there called `calc_area_local`, is a local function as the function name does not match the filename.

```matlab
1 - edge1 = 7.6;
2 - edge2 = 4.3;
3 - area_of_rect = calc_area_local(edge1,edge2);
4 - % Code line 3 calls the function and gives it the inputs.
5 - disp(,['Area is ',num2str(area_of_rect)])
6 -
7 - function [area] = calc_area_local(length,width)
8 - area = length * width;
9 - end
```
Global versus local function

- The function with a function name that matches the file name is known as global functions.
- We have been calling global functions as the main function.
- The main difference between a global and local function is that global functions can be called in other scripts or in the command window.
- Whereas, local functions can only be called in the file that it is written in.
- The reason why we have local functions is because if a function or even a script has a subtask that’s not general enough for its own file, we can add a local function in the same file as that function.
- In Python you may have called local functions as helper functions.
- They are of similar concept.
Input assignments

- The function `calc_area_peri` calls another two functions.
- The first function is another global function called `calc_area`, see below.
- It is pretty much the same as the file `calc_rect_area` except this is a function file instead of a script file as it has no lines of code outside of the function.

```matlab
function [area] = calc_area(length,width)
    area = length * width;
end
```

```matlab
function [area,peri] = calc_area_peri(length,width)
    area = calc_area(length,width);
    peri = calc_peri(length,width);
end
```

% The following is an example of local function
```matlab
function p = calc_peri(l,w)
    p = 2*(l+w);
end
```
Calling local functions

• The second function that calc_area_peri calls is a local function called calc_peri. This local function takes two inputs (a length and width) and returns the perimeter of a rectangle.

• We want to note here that the end keyword is particularly important when we have more than one function in a file.

```matlab
function [area, peri] = calc_area_peri(length, width)
    area = calc_area(length, width);
    peri = calc_peri(length, width);
end

% The following is an example of local function
function p = calc_peri(l,w)
    p = 2*(l+w);
end
```
The `calc_area_peri` function returns two outputs.

If we look at the `run_calc_area_peri` script, code line 4, we need to provide two variables to assign the returned values because this function returns 2 outputs.

```matlab
% Call calc_area_peri function
edge1 = 7.6;
edge2 = 4.3;
[area_2,peri_2] = calc_area_peri(edge1,edge2);
disp(['Area is ',num2str(area_2)])
disp(['Perimeter is ',num2str(peri_2)])
```

```matlab
function [area,peri] = calc_area_peri(length,width)
    area = calc_area(length,width);
    peri = calc_peri(length,width);
end

% The following is an example of local function
function p = calc_peri(l,w)
    p = 2*(l+w);
end
```
Note on local functions

• You cannot call local functions outside of the file it is written
• For example, you cannot call the local function `calc_peri()` in `calc_area_peri()` from another file.

```matlab
function [area,peri] = calc_area_peri(length,width)
    area = calc_area(length,width);
    peri = calc_peri(length,width);
end

% The following is an example of local function
function p = calc_peri(l,w)
    p = 2*(l+w);
end
```

Function file `calc_area_peri.m`
**Variable number of inputs and outputs**

- You can write a Matlab function so that it can deal with a variable number of inputs or outputs.

- Example: The sum function can be used with
  - One input, e.g. `sum(m);` or
  - Two inputs, e.g. `sum(m,2)`

- You will need to use variables `nargin` and `nargout`.
  - We won’t be covering these in ENGG1811.
Variable scooping in Matlab

- Matlab functions are passed by value
- By default, all variables are local
- Script and functions have different memory spaces
- Base space is for scripts
- Each function has the same memory space
- If the same variable name is used in different functions, these variables have nothing to do with each other
Scope of the variables

- Separate memory space for each function
- Variables are local

Code: quadruple.m

```matlab
function y = quadruple(x)
    y = 2*double(x);
end

function y = double(x)
    y = 2*x;
end
```

Memory space for function quadruple

<table>
<thead>
<tr>
<th>x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>12</td>
</tr>
</tbody>
</table>

Memory space for function double

<table>
<thead>
<tr>
<th>x</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>6</td>
</tr>
</tbody>
</table>
Let us say the speed \( r \) of an object varies over time \( t \) as follows:

\[
r(t) = 26.2 - 3.1 \exp(-2t) \cos(4t)
\]

You want to find the maximum speed for \( 0 \leq t \leq 2 \), and at what time it occurs.

**Steps:**

1. Define a vector, which we will refer to as \( \text{vec}_t \), whose elements are 0, 0.01, 0.02, 0.03 and so on, up to and including 2.
2. Assuming the elements of \( \text{vec}_t \) are time instances, compute a vector \( \text{vec}_r \) which has the same size as \( \text{vec}_t \) such that the \( i^{\text{th}} \) element of \( \text{vec}_r \) is the speed at the time specified by the \( i^{\text{th}} \) element of \( \text{vec}_t \). You should not use loops for this.
3. Use the Matlab built-in function \text{max} \ to determine the maximum speed.
4. In order to determine the time at which the maximum speed occurs, one method is to use the Matlab function \text{max} \ with 2 output arguments. Look up the documentation to see what the second output is and use that to help you.
<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>%</td>
<td>#</td>
</tr>
<tr>
<td>Division</td>
<td>/, \</td>
<td>/, %</td>
</tr>
<tr>
<td>Exponential</td>
<td>^</td>
<td>**</td>
</tr>
<tr>
<td>Not equals to</td>
<td>~=</td>
<td>!=</td>
</tr>
<tr>
<td>NOT operator</td>
<td>~</td>
<td>not</td>
</tr>
<tr>
<td>AND operator</td>
<td>&amp;, &amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td>OR operator</td>
<td></td>
<td>or</td>
</tr>
</tbody>
</table>
## Comparing MATLAB and python – Decision making

<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>1, true</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>0, false</td>
<td>False</td>
</tr>
<tr>
<td>Else if statement</td>
<td>elseif</td>
<td>elif</td>
</tr>
</tbody>
</table>
## Comparing MATLAB and python - Arrays

<table>
<thead>
<tr>
<th></th>
<th>MATLAB</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Dimensional Arrays</td>
<td>Vector</td>
<td>List</td>
</tr>
<tr>
<td>Two Dimensional Arrays</td>
<td>Matrix</td>
<td>List Of Lists</td>
</tr>
<tr>
<td>Access elements in array using</td>
<td>Round Brackets ()</td>
<td>Square Brackets []</td>
</tr>
<tr>
<td>Index</td>
<td>Starts at 1</td>
<td>Starts at 0</td>
</tr>
<tr>
<td>Dimensions</td>
<td>1 for columns, 2 for rows</td>
<td>1 for columns, 0 for rows</td>
</tr>
</tbody>
</table>
### Comparing MATLAB and Python - Functions

<table>
<thead>
<tr>
<th></th>
<th>Matlab</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword for function</td>
<td>function</td>
<td>def</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary of key concepts

• Arithmetic Operators
• Built-in Functions
• Logical Comparison and Boolean Operators
• Colon Notation

Array
  – Defining, Indexing
  – Elementwise Operators
  – Logical Arrays

• If-Statements
• For-loops
• Functions