

# Lecture #3

Psych 258

Russ Poldrack, Instructor

# Outline

- Structured data types
  - matrices/arrays
  - cell arrays
  - structures & structure arrays
- Functions
- Debugging strategies

# Structured data types

- Many data types need to hold more than a single data value
- These values may be of different types
  - integer
  - floating point
  - string
- Structured data types allow for multiple values of different types to be stored in a single variable

# Arrays (aka matrices)

- An n-dimensional set of values
  - [2 4 5 6], [1:100]
  - must be a single data type within the array
    - except for cell arrays - coming soon
- MATLAB functions can almost always take an array as an argument in place of a single number, and return an array of results

```
>> sin(0)
```

```
ans =  
    0
```

```
>> sin([0 pi/4 pi/2 3*pi/4 pi])
```

```
ans =  
    0    0.7071    1.0000    0.7071    0.0000
```

# Addressing 1D arrays

```
>> x=[5.3 19 6 12.7 8 18 23];
```

```
>> x(1)
```

```
ans =  
    5.3000
```

```
>> x(3)
```

```
ans =  
    6
```

1D column and row arrays  
are indexed using a single  
index value

```
>> x=x'
```

```
x =
```

```
    5.3000  
   19.0000  
    6.0000  
   12.7000  
    8.0000  
   18.0000  
   23.0000
```

```
>> x(1)
```

```
ans =  
    5.3000
```

```
>> x(3)
```

```
ans =  
    6
```

# Accessing N-Dimensional arrays

```
>> d=ceil(rand(3,4)*20)
```

```
d =
```

```
     7     8     9    16
    17    15    14    20
    12    11    13    11
```

```
% the first index denotes
% the row, the second
% denotes the column
```

```
>> d(2,1)
```

```
ans =
```

```
    17
```

```
>> d(3,4)
```

```
ans =
```

```
    11
```

```
>> d=rand(2,2,2)
```

```
d(:,:,1) =
```

```
    0.8801    0.9797
    0.1730    0.2714
```

```
d(:,:,2) =
```

```
    0.2523    0.7373
    0.8757    0.1365
```

```
>> d(1,2,1)
```

```
ans =
```

```
    0.9797
```

# Working with strings

- Strings are stored as arrays of characters

```
>> t=['one','two']  
t =  
onetwo
```

```
>> t=['one';'two']  
t =  
one  
two
```

```
% strings must be of the same size!
```

```
>> t=['one';'two';'three']  
??? Error using ==> vertcat
```

All rows in the bracketed expression must have the same number of columns.

```
>> t=['one  '; 'two  '; 'three']  
t =  
one  
two  
three
```

```
>> t=['one','two'];  
>> t(1)
```

```
ans =  
o
```

```
>> t(4)
```

```
ans =  
t
```

# String processing

- MATLAB has many built-in functions for working with strings (see `help strfun`)
  - `strcmp(string1, string2)`
    - returns 1 if strings are the same, 0 otherwise
  - `newstr=strcat(string1, string2)`
    - concatenate strings
  - `pos=findstr(string1, string2)`
    - finds string pattern within another string
    - returns position of pattern, or [] if not found

# Cell arrays

- Standard arrays require that all elements be the same data type
- Cell arrays allow storage of arbitrary information in an array
  - e.g., strings of different sizes

```
>> c={1 'test'; 'p' 4.5} >> whos
Name      Size      Bytes  Class
c =
ans      1x1        68    cell array
[1]      'test'     266   cell array
'p'      [4.5000]   68    cell array
k        1x1         8     double array
```

```
>> j=c(1)
```

Grand total is 16 elements using 410 bytes

```
j =
```

```
[1]
```

Assigning values: you must use a curly bracket on one side of the assignment statement

```
>> k=c{1}
```

```
k =
```

```
1
```

```
>> t={1 'test' 4.5};
```

```
>> j{1}=1;
```

Retrieving values:

- The curly bracket retrieves the value in its native data format.
- The square bracket retrieves the value in cell array data format

# Converting from cell arrays to other formats

```
>> foo={1 2; 3 4; 5.5 6}
```

```
foo =
```

```
    [ 1]    [2]  
    [ 3]    [4]  
 [5.5000]    [6]
```

```
>> cell2mat(foo)
```

```
ans =
```

```
1.0000    2.0000  
3.0000    4.0000  
5.5000    6.0000
```

```
>> foo2={1 'a';2 'b';3 'c'}
```

```
foo2 =
```

```
    [1]    'a'  
    [2]    'b'  
    [3]    'c'
```

```
>> cell2mat(foo2)
```

```
??? Error using ==> cell2mat at 47
```

```
All contents of the input cell array must be  
of the same data type.
```

```
>> cell2mat(foo2(:,1))
```

```
ans =
```

```
1  
2  
3
```

```
>> cell2mat(foo2(:,2))
```

```
ans =
```

```
a  
b  
c
```

# Structures

- Structures are used to store different data types within a single variable
  - Different fields of the variable are addressed using a field name rather than a cell index
- Structures are most useful for data organized into records with different fields
  - Think of them like a form with different fields for entering information

```
>> file=struct(...  
'Name','Russ Poldrack',...  
'Room',6639,...  
'Building','Franz Hall')
```

```
file =
```

```
    Name: 'Russ Poldrack'  
    Room: 6639  
  Building: 'Franz Hall'
```

```
>> file.Name
```

```
ans =
```

```
Russ Poldrack
```

```
>> exam.date='12-30-02'
```

```
exam =
```

```
    date: '12-30-02'
```

Structures can be created using the struct() function

Structures are accessed using the name.field syntax

New structures can also be created directly by using the name.field syntax in an assignment statement

# Structures and arrays

- Structures can also be created as arrays

```
>> s = struct('type',{'big','little'},'color','red','x',{3 4})
```

```
s =  
1x2 struct array with fields:  
  type  
  color  
  x
```

```
>> s(1)
```

```
ans =  
  type: 'big'  
  color: 'red'  
  x: 3
```

```
>> s(1).x
```

```
ans =  
 3
```

```
>> s.color
```

```
ans =  
red
```

```
ans =  
red
```

```
>> s.type
```

```
ans =  
big
```

```
ans =  
little
```

# Functions versus scripts

- Both functions and scripts are stored in .m files
- A function takes a particular input and returns a particular output to a variable
  - $j = \sin(x)$
- A script is self-contained, in that it does not take explicit input or output arguments
  - process\_data

# Writing functions

```
function y=average(x)
% AVERAGE mean of a vector
% AVERAGE(x), where x is a
% vector, returns the mean
% of the vector elements
```

- function definition line
- H1 line - this is searched by lookup
- help text

```
s=size(x);
if (min(s) > 1 | max(s)==1),
    error('input must be a vector');
end;
```

- function body

```
y=sum(x)/length(x);
```

- the output variable (y in this case) must be defined in the function

```
function y = average(x)
```

input argument  
function name  
output argument

Multiple input/output arguments

```
function [x, y, z] = sphere(theta, phi, rho)
```

No output arguments

```
function printresults(x)
```

# The return statement

- By default, a function will return to the calling program at the last line in the program
- However, sometimes we want to exit the function earlier
- The return statement causes the function to return before the last line
  - Useful in case of errors or bad input values

```

function output = sinc(input,plotflag)
% function output = sinc(input,plotflag)
% this function computes the sinc function of an input
% the input must be numeric

USAGE = 'function output = sinc(input)\n';

% first test the input
if ~exist('input'),           % note that you first must test for existence, before you
    fprintf('%s',USAGE);     % actually use the variable name to test for other things
    output=[]; return;
elseif isempty(input),
    fprintf('%s',USAGE);
    output=[]; return;
elseif ~isnumeric(input),
    fprintf('%s',USAGE);
    output=[]; return;
end;

output = sin(input)./ input;

% test the plot flag
if ~exist('plotflag'),
    plotflag=0;
elseif isempty(plotflag) | ~isnumeric(plotflag),
    plotflag=0;
end;

if plotflag,
    plot(input,output);
end;

return;

```

A function file can contain multiple sub-functions,  
defined below the main function

```
function y=square(x)
% SQUARE returns the square of a value
% if a matrix is provided, it is
% squared in an elementwise manner

y = product(x,x);

function y=product(i,j)
% this subfunction actually does the
% multiplication
y = i .* j;
```

Only the main function  
is visible from outside

the variables within the  
subfunction are only  
accessible within the  
subfunction (i.e., they  
are local variables)

# Variable scope

- The variables in a program remain the MATLAB workspace after the program is run
- The variables defined within a function are, by default, local to that function
  - They can be made to persist in the workspace by defining them as global variables
  - They must be defined as global both in the workspace and within the function

# Variable scope

```
% lecture3_4.m
```

```
j = 1;  
temp = j*5;  
k=temp;
```

```
>> lecture3_4  
>> who
```

```
Your variables are:  
j      k      temp
```

```
function k = lecture3_func(j)
```

```
temp = j*5;  
k=temp;  
return;
```

```
>> k=lecture3_func(1);  
>> who
```

```
Your variables are:  
k
```

# Testing function input

- Input should always be tested
  - to see whether an input variable has been defined, using `exist()`
    - `if ~exist('varname')`
    - `varname` must be in quotes
  - to see whether a defined variable is empty, use `isempty()`
    - `if isempty(varname),`
  - To see whether it is the proper data type, use `ischar()`, `isnumeric()`
  - To see whether an array is one-dimensional
    - `min(size(array))==1`

# Debugging strategies

- Writing the program is usually only a fraction of total development time
- With complex programs, debugging can be quite difficult
  - complex dependencies
  - difficult to apprehend the entire variable space

# Writing understandable code

- Most important key to debugging
  - Include comments as needed
    - don't overcomment!
  - include line breaks
  - use indentation for loops and if statements
  - label each end of a loop or if statement
  - use mnemonic variable names
  - Separate major blocks of code with blank space

```

% lecture3_3.m
load rt; % load existing data from file rt.mat

stdevs_to_trim=2; % define how many stds to trim

% compute mean and stdev of original dataset
% and cutoff for trimming

mean_rt = mean(rt);
std_rt = std(rt);
cutoff = mean_rt + stdevs_to_trim*std_rt;

good_rt_counter=0; % create a counter for good trials

% loop through trials, deciding for each one whether
% it is below the cutoff, keeping it if so

for trial=1:100,
    if rt(trial) < cutoff,
        good_rt_counter = good_rt_counter+1;
        good_rt(good_rt_counter)=rt(trial);
    end; % if rt(trial) ...
end; % for trial ...

% compute # of trimmed trials and trimmed RT

number_of_trimmed_trials = size(rt,2)-good_rt_counter;

trimmed_mean_rt = mean(good_rt);

% print results to screen

fprintf('%d trials were trimmed\n',number_of_trimmed_trials);
fprintf('Original mean was %0.2f\n',mean_rt);
fprintf('Trimmed mean was %0.2f\n',trimmed_mean_rt);

```

- Include necessary comments
- include line breaks
- use indentation for loops and if statements
- label each end of a loop or if statement
- use mnemonic variable names
- Separate major blocks of code with blank space

# When things go wrong

- When the program does not run properly, it's necessary to trace its execution and examine its operation
  - looking at variable values during execution
  - testing for various problematic conditions

# using a DEBUG flag

- Put output statements within if statements conditional on a debug flag
  - this allows you to easily turn them on or off as needed

```
DEBUG=1;
counter=1;

range=[-4*pi:0.2:4*pi];
for x=range,
    sinc(counter)=sin(x)/x;
    if DEBUG,
        fprintf('%0.3f: %0.3f\n',x,t(counter));
    end;
    counter=counter+1;
end;

if DEBUG,
    plot(range,sinc);
end;
```

# try/catch statement

- like an if/else statement where the else statement is executed in case of an error in the try section

```
% lecture3.m
```

```
try,  
    foo=supercalifrag(1);  
catch,  
    fprintf('there was an error!\n');  
    % the error message is returned by lasterr  
    fprintf('The error message was:\n%s\n',lasterr);  
end;
```

```
>> lecture3  
there was an error!  
The error message was:  
Undefined function or variable 'supercalifrag'.
```

# Defensive programming

- Assume that the worst will happen!
  - check all input for correctness
    - existence, data type and bounds
  - use try/catch in places where errors might occur, and then handle the errors gracefully
  - provide a reasonable “catch” or default for if..elseif..else statements
  - Provide diagnostic and understandable error messages

# MATLAB debugging tools

- MATLAB has a set of interactive debugging tools

`dbstop if error` causes matlab to enter the debugger if an error occurs

`dbstop in test.m at 5` causes matlab to enter debugger at line 5 in test.m

`dbstep` executes the next line of code after stopping

`dbcont` continues execution of the program