General MATLAB

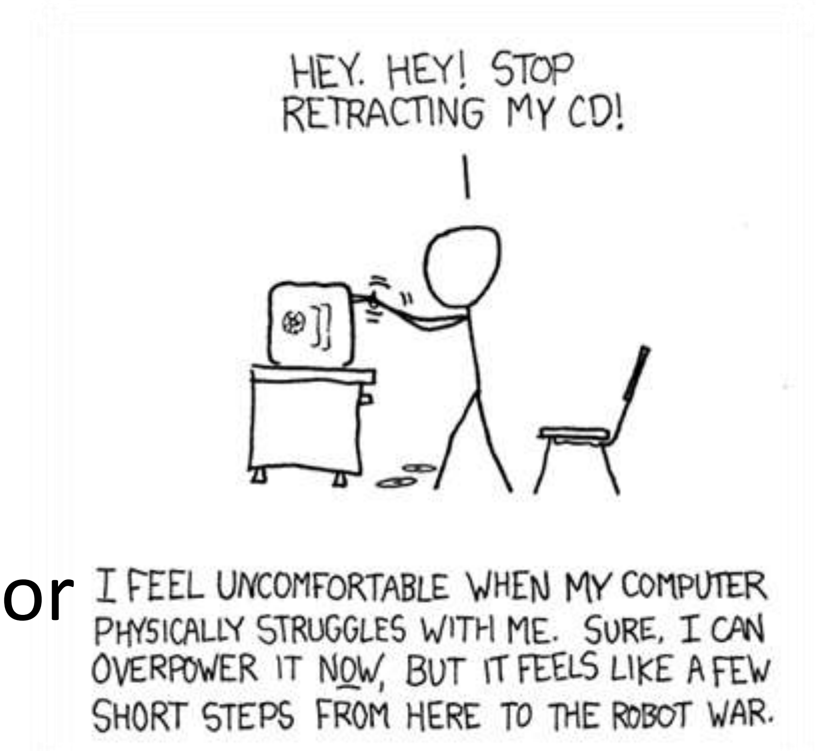
Bending MATLAB to your will

Beyond The Mouse
April 24, 2009
Celso Reyes

from http://www.classicmanga.com/20thcenturyboys/images/spoon-bending.jpg April 23, 2009

Outline

- Answer questions from handout
- Working from the command line
  - Structs
  - Cells
- Writing Scripts in the editor
- Creating and Juggling functions


HEY. HEY! STOP RETRACTING MY CD!
I FEEL UNCOMFORTABLE WHEN MY COMPUTER PHYSICALLY STRUGGLES WITH ME. SURE, I CAN OVERPOWER IT NOW, BUT IT FEELS LIKE A FEW SHORT STEPS FROM HERE TO THE ROBOT WAR.

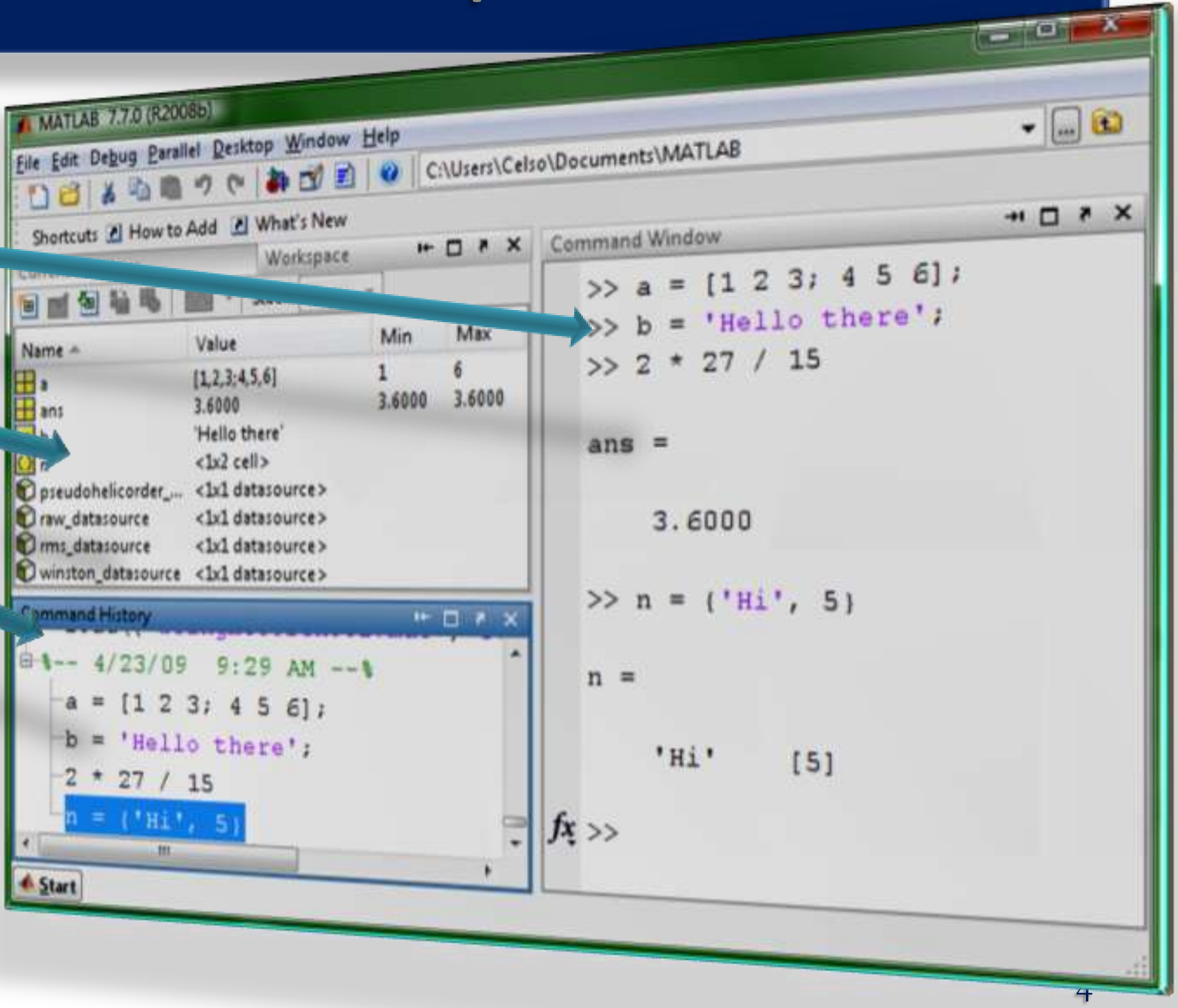
from http://xkcd.com/251 April 22, 2009

MATLAB BASICS (From Handout)

- Basic Data types are Double, char, and logical
- ALL data are Arrays (1x1, 1xn , nxm, nxmxp…)
- Data Initialization
- Accessing data: [ ], ( )
- Indexing tricks: end, colon, and apostrophe

The MATLAB desktop environment

- Command window
- Workspace
- History Window


MATLAB 7.7.0 (R2008b)
File Edit Debug Parallel Desktop Window Help
Shortcuts How to Add What's New
Workspace
Command Window
Command History
Start

C:\Users\Celso\Documents\MATLAB

Name	Value	Min	Max
a	[1,2,3;4,5,6]	1	6
ans	3.6000	3.6000	3.6000
b	'Hello there'		
n	<1x2 cell>		
pseudohelicorder_...	<1x1 datasource>		
raw_datasource	<1x1 datasource>		
rms_datasource	<1x1 datasource>		
winston_datasource	<1x1 datasource>		

```
%-- 4/23/09  9:29 AM --%
a = [1 2 3; 4 5 6];
b = 'Hello there';
2 * 27 / 15
n = {'Hi', 5}
```

```
>> a = [1 2 3; 4 5 6];
>> b = 'Hello there';
>> 2 * 27 / 15

ans =

    3.6000

>> n = {'Hi', 5}

n =

    'Hi'    [5]

fx >>
```

The MATLAB –nodesktop environement

```
>>
```

- Command prompt
- MUCH Faster on slow machines
- Best that most PC's can hope for when SSH'ing into the SUN or LINUX networks.

Programming at the prompt

- All variables are created in the Workspace.
- The history window keeps track of each line you’ve typed and can be used to repeat commands.
  - recently used commands can be repeated through the use of up-arrows, and down-arrows
- After the first few letters of a command have been written, the **TAB** key may be able to auto complete your line.
- OKish for tinkering.

Introducing both **structs** and **cells**

- A struct is a special data type whose data is stored in fields that are accessible by name
  - student.name = ‘joe’
  - student.age = 25;

  equivalent to...
  - student = struct(... ‘name’ , ‘joe’ , ’age’ , 25)

student	(1)	(2)	(3)
.name	‘Jack’	‘Jo’	‘Jake’
.age	21	25	30

- A cell is a container that can hold disparate types of data
  - mycell(1) = {[1 5]}
  - mycell(2,1) = {student}

curly braces tell MATLAB to wrap this value inside a cell.

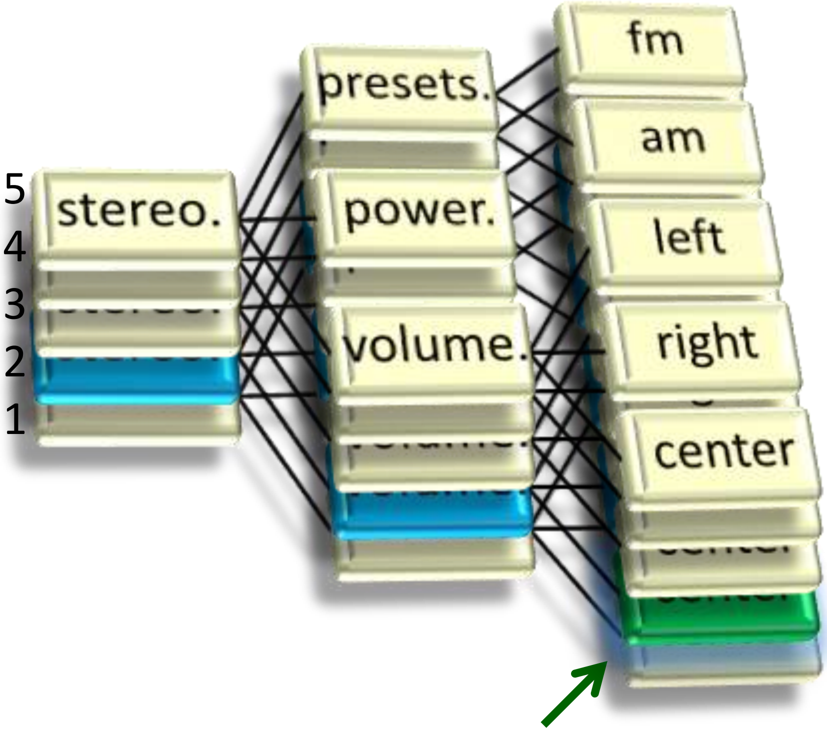
MyCell	{:,1}	(:,2)	{:,3}
{1,:}	[1 5]	‘Ted’	true
{2,:}	21	student	30

Structs

- structs may be nested
- all elements within an array of structs will have same fields.
- field names can be found with **fieldnames()** function.
- If values have same size, you can get all values from a field at once.

  vols = [stereo.volume]

  ** but only 1 level deep!*



stereo(2).volume.center

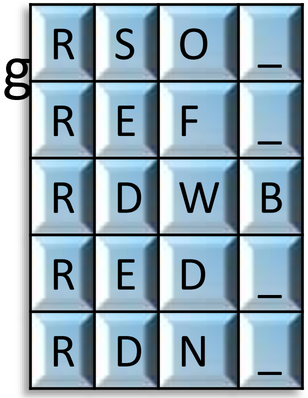
Referencing Cells

- items are put into cells by surrounding the item with curly braces. *e.g.*
  mycell = {item_1, item_2,... , item_{max}}
- each cell element (a cell) can be retrieved with parenthesis. *e.g.*
  mycell(index) = $\text{mycell}_{\text{index}}$
- each cell value is accessed with curly braces. *e.g.*
  mycell{index} = $\text{item}_{\text{index}}$
- cells can provide multiple arguments to a function. e.g.
  funkyfunction(mycell{:})

Cells vs String Arrays

Character array

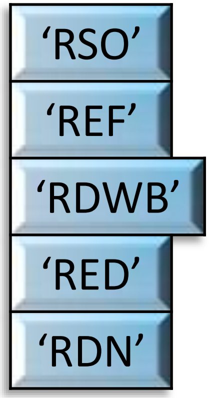
- Each character is an element
- Each string must be the same length, but spaces can be used to pad them to the same length.
- Access each string via (row,:)
- Access columns via (:,col)



R	S	O	_
R	E	F	_
R	D	W	B
R	E	D	_
R	D	N	_

Cell array

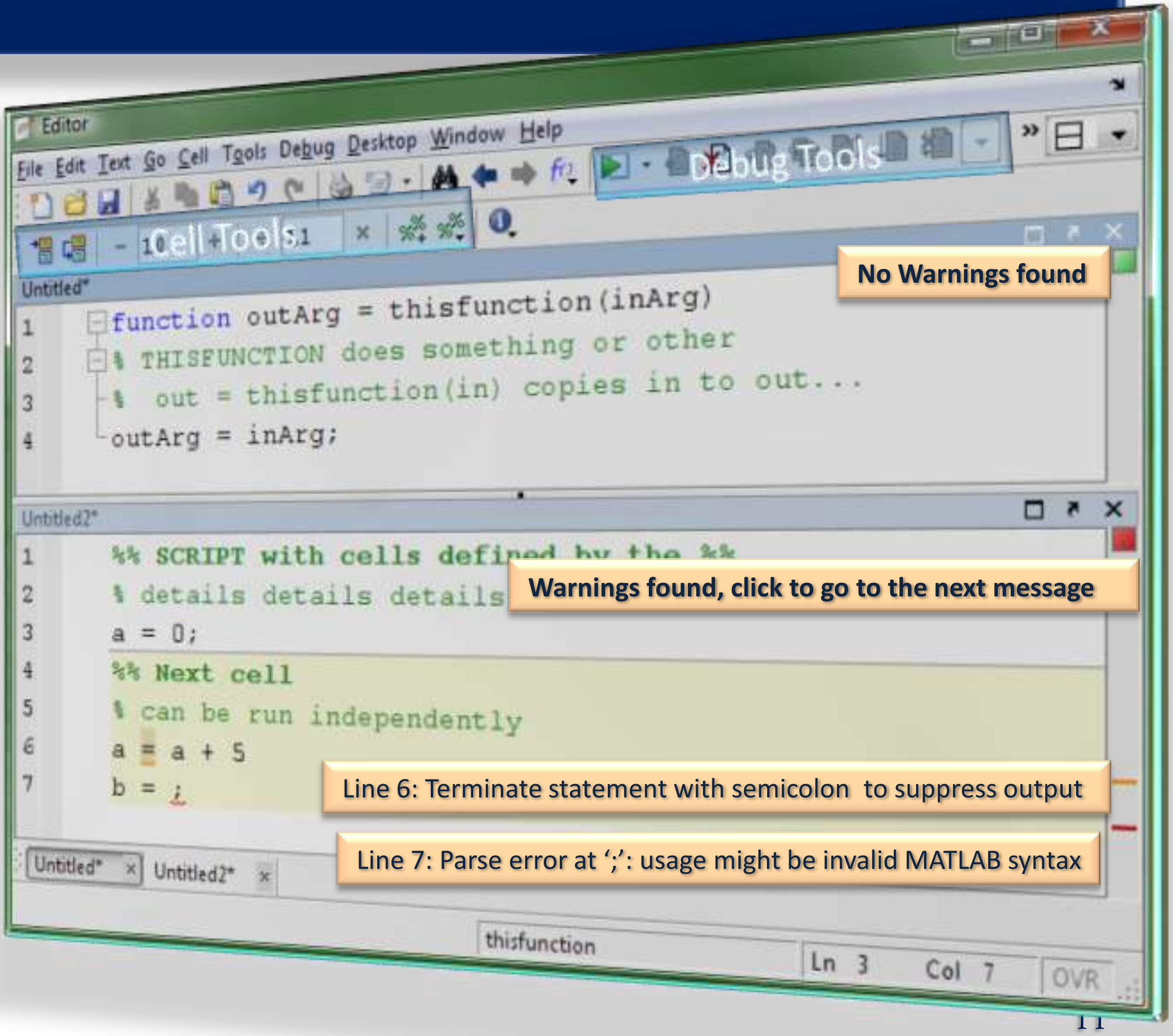
- Each entire string is an element
- Each string can be any length
- Access string via {whichword}



‘RSO’
‘REF’
‘RDWB’
‘RED’
‘RDN’

The MATLAB Editor

- Debug Controls
- Cell Tools
- M-Lint Code Analyzer messages



No Warnings found

```
1  function outArg = thisfunction(inArg)
2  % THISFUNCTION does something or other
3  %   out = thisfunction(in) copies in to out...
4  outArg = inArg;
```

```
1  %% SCRIPT with cells defined by the %%
2  % details details details
3  a = 0;
4  %% Next cell
5  % can be run independently
6  a = a + 5
7  b = ;
```

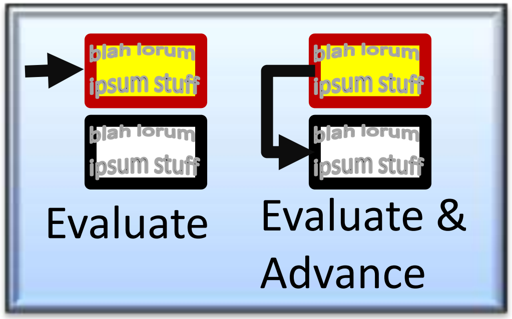
Warnings found, click to go to the next message

Line 6: Terminate statement with semicolon to suppress output

Line 7: Parse error at ';': usage might be invalid MATLAB syntax

Scripting with MATLAB

- Variables used in scripts are created in the workspace. When the script finishes, these variables still exist.
- When the script starts, variables may or may not already exist.
- Sections of the script can be run independently.
  - Each new section starts with %%
  - Comments start with %

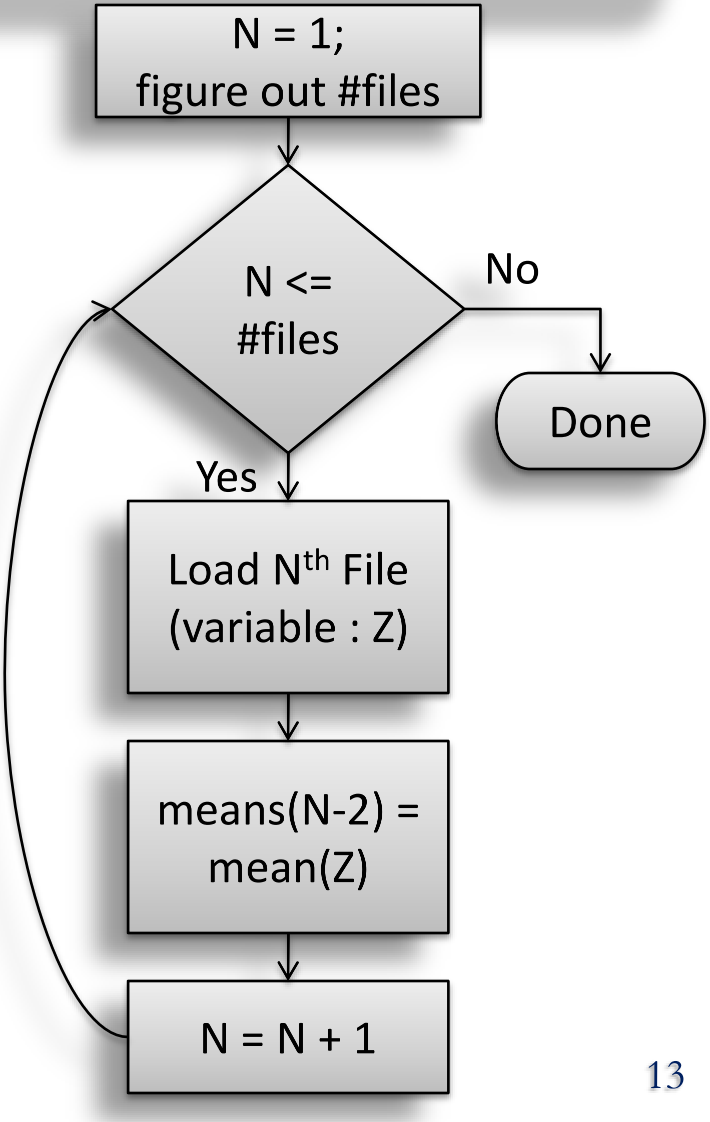


Scripting with MATLAB

```
%% Script grabs mean of each GPS file in a dir

% directory containing preprocessed GPS files
files = dir(‘C:/data/2009/04’);

%% Loop through each file, and get its mean
% we’re skipping the first two files ‘cause they
   are always ‘.’ and ‘..’
for n = 3 : numel(files)
  fileName = fullfile(‘C:/data/2009/04/’, …
  files(n).name); %one file per month
  load(fileName)  %our variable is called “z”
  means(n-2) = mean(z);
end
```



Scripting with MATLAB

```
%% Script grabs mean of each GPS file in a dir

% directory containing preprocessed GPS files
files = dir(‘C:/data/2009/04’);

%% Loop through each file, and get its mean
% we’re skipping the first two files ‘cause they
   are always ‘.’ and ‘..’
for n = 3 : numel(files)
 fileName = fullfile(‘C:/data/2009/04/’, ...
   files(n).name); %one file per month
 load(fileName)  %our variable is called “z”
 means(n-2) = mean(z);
end
```

- Each time you wish to process a different month, you’ll need to change multiple items in the source code.
- If fileName contains a variable called “n”, “means”, or “fileName”, then strange values may pop up
- What happens if this is run for January, then for February?
- There is no direct correlation between means and the file name? what if a file is missing for a day?

Scripting with MATLAB

```
%% Script grabs mean of each GPS file in a dir

% directory containing preprocessed GPS files
myDir = ‘C:/data/2009/04’;
files = dir(fullfile(myDir,‘*.mat’));
means = [];
filenames  = {};

%% Loop through each file, and get its mean

for n = 1 : numel(files)  % one file per month
  fileName = fullfile(myDir, files(n).name);
  gpsFileContents = load(fileName)
  means(n) = mean(gpsFileContentes.z);
  filenames(n) = {fileName};
end
```

Tweaks

- A variable was created to hold the directory value. Now it only needs to be changed in one place.
- Data loaded from file is stored in a specific variable.
- Output variables are cleared ahead of time
- Extraneous files are excluded prior to the loop
- Both the mean and filename are kept.

Creating functions

```
function outputStuff = function_name  (inputStuff)
% FUNCTION_NAME here is the one line summary of the function, used by LOOKFOR
% This is the body of the function where it is explained exactly how to
% call it, what it does to the data, and shows an example of how it should be used.
% All of this shows up when someone types help function_name at the prompt

% Because this line isn’t contiguous with the previous comments, it doesn’t appear
% on the help.  Instead, it is merely a comment internal to the program

outputStuff = inputStuff;  %this is where the actual operations start
```

- A function only knows about variables that are created within it, so there is no need to worry about pre-existing values.
- The comments immediately below the function declaration are displayed when the user asks for **HELP** for a function
- The MATLAB command **lookfor** searches the first comment line

Creating functions

```
function get_gps_means( myDir)
% get_gps_means calculates means for a gps file
% USAGE: get_gps_means(directory);

% directory containing preprocessed GPS files
files = dir( fullfile ( myDir , ’*.mat’) );

%% Loop through each file, and get its mean
for n = 1 : numel(files)  % one file per month
  fileName = fullfile(myDir, files(n).name);
  gpsFileContents = load(fileName)
  means(n) = mean(gpsFileContentes.z);
  filenames(n) = {fileName};
end
```

This code has been moved from a script to a function.

Accepts the directory as an input

Creating functions

```
function [means dates] = get_gps_means(startday, endda
%Figure out which files to grab, they're in directories like
%"C:/DATA/YYYY/MM" in files called gpsDD.mat
dates = fix(datenum(startday)) : fix (datenum(endday))
nDates = numel(dates);
[Y M D] = datevec(dates);
means = nan(1,nDates)

for n = 1 : nDates
 thisfile = {sprintf('C:/DATA/%04d/%02d/gps%02d.mat',...
    Y(n),M(n),D(n))}
 if (exist(thisfile,'file'))
    tmp = load(thisfile);
    if any(strcmp(fieldnames(tmp),'z'))
     means(n) = mean(tmp.z);
    else
      disp(['unable to load file ' thisfile]);
    end
  end
end
```

Now, any arbitrary range of dates can be processed.

Both multiple arguments and return values are present.

Creating functions

The same process, broken into subfunctions makes understanding the main program easier and isolating each behavior.

```
function [means dates] = get_gps_means(startday, endday)
%Figure out which files to grab, they're in directories like
%"C:/DATA/YYYY/MM" in files called gpsDD.mat

dates = fix(datenum(startday)) : fix (datenum(endday))
nDates = numel(dates);
means = nan(1,nDates);

for n=1:nDates
   thisfile = getfilename(date(n));
   means(n) = process(thisfile);
end
```

```
function means = process(filename)
%Load a file, and return the mean of its Z’s
if (exist(thisfile,'file'))
     tmp = load(thisfile);
    if any(strcmp(fieldnames(tmp),'z'))
       means = mean(tmp.z);
    else
        disp(['unable to load file ' thisfile]);
        means = nan;
    end
end
```

```
function fn = getfilename(thisdate)
%Figure out which files to grab based on date
[Y M D] = datevec(thisdate);

thisfile = {sprintf(...
'C:/DATA/%04d/%02d/gps%02d.mat‘,...
Y,M,D}
```

subfunctions

- subfunctions are all written in the same file as, and are written after the *primary* function.
- Subfunctions are only accessible to the functions contained within that one file.

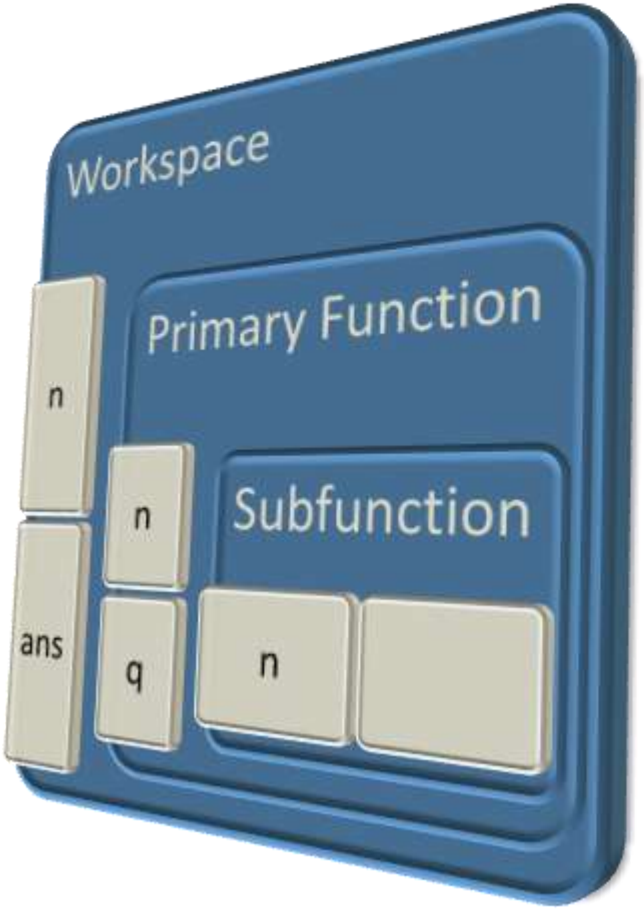
```
function outStuff = primary(inStuff)
% The primary function is first function in the
% M-file.  This function can be invoked from
% outside the M-file.
outStuff = subfunction (inStuff);
outStuff = otherSub(outStuff);

function myStuff = subfunction (myStuff)
% visible only to all functions within this file.
myStuff = myStuff  .* 2;

function outStuff = otherSub(inStuff)
% visible only to all functions within this file.
outStuff = subfunction (inStuff);
outStuff = outStuff + 1;
```

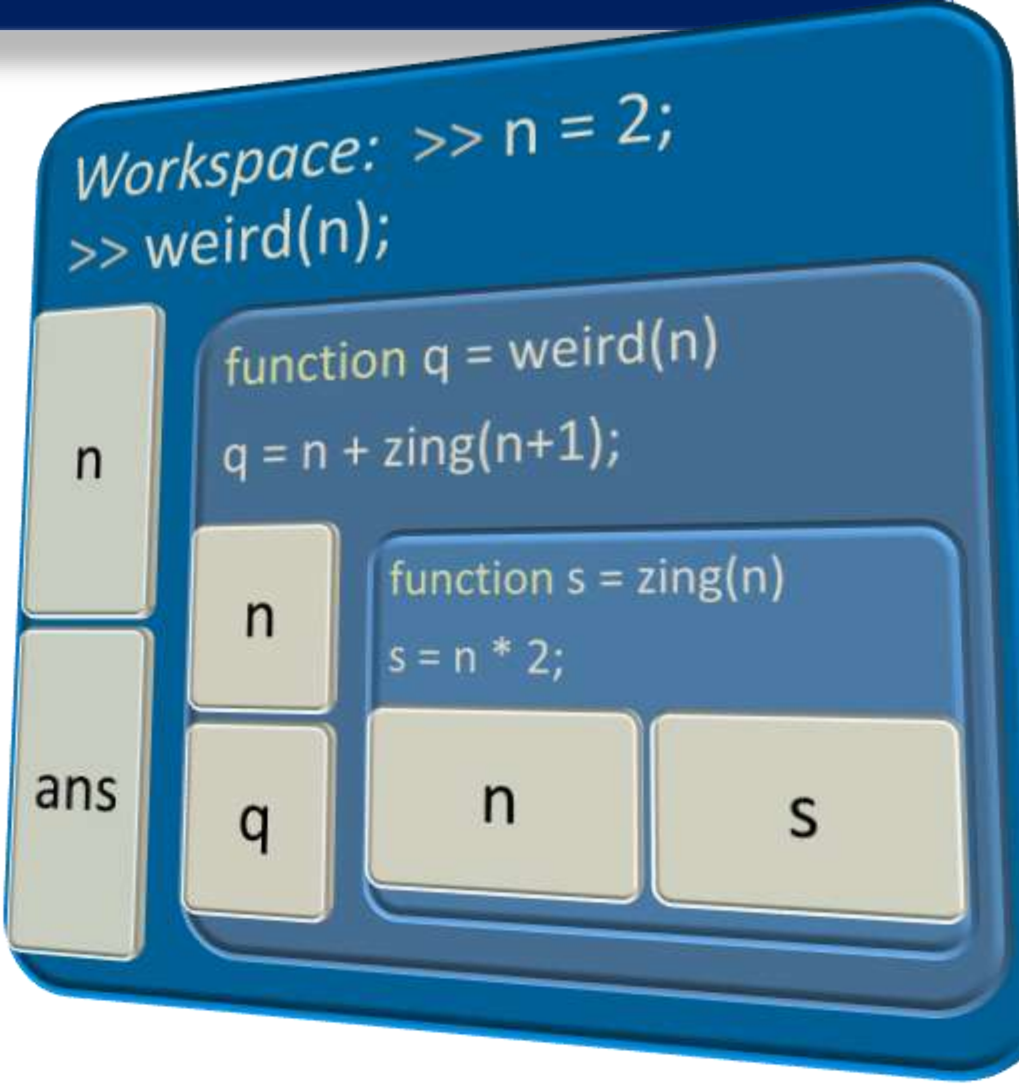
Variable Scope

- SCOPE of a variable is the section of code that has access to it.
  - A variable's scope is usually limited to the function in which it was created. In subfunctions, goes out of scope.
- LIFE of a variable is the entire time it exists, from creation to deletion.
  - A variable can be out of scope, but still exist.



Variable Scope Exercise

- Follow this program to determine scope and lifetime of each of the variables...



```
Workspace: >> n = 2;
>> weird(n);
```

```
function q = weird(n)
q = n + zing(n+1);
```

```
function s = zing(n)
s = n * 2;
```

(Argument lists)

- **Arguments** are the inputs to a function.
  - Enclosed in parenthesis
  - comma separated
  - Number of input arguments can be determined by using **nargin**

[Return Types]

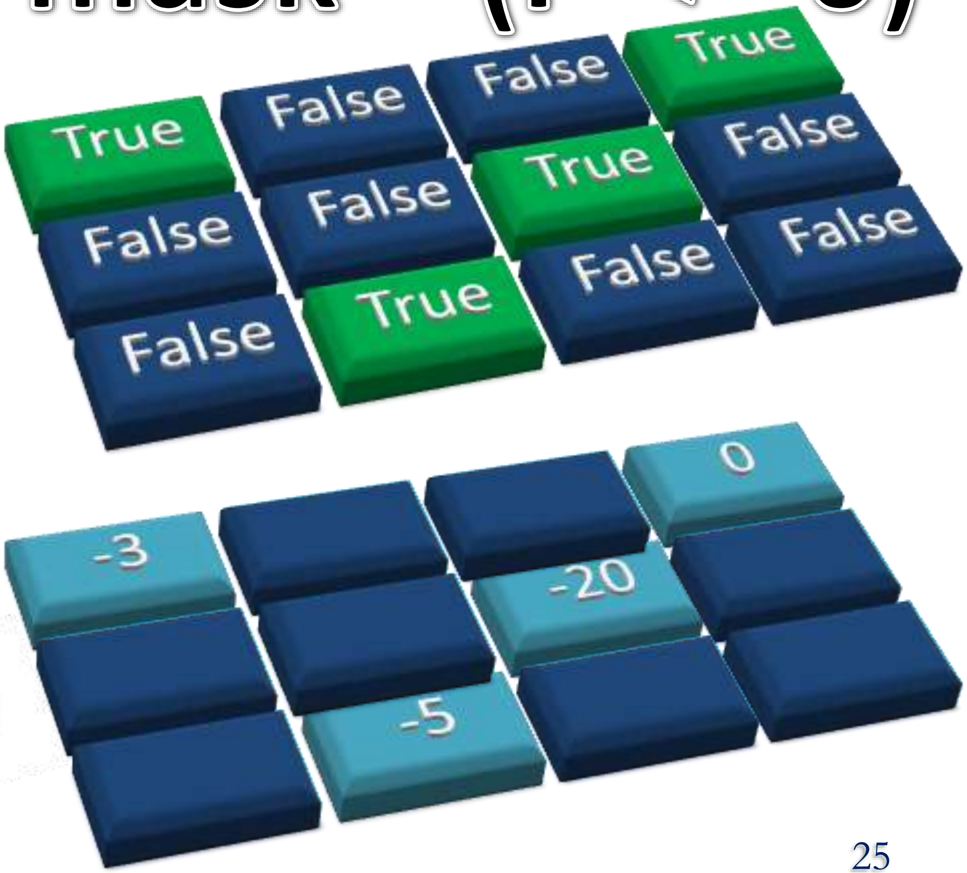
- **Return Types** are the values that a function passes back to the main program
  - Multiple return types are enclosed in square brackets.
  - A program can find out how many variables it was called with by using **nargout**

Masks

- A **mask** is an array of logical values that can overlay another array, allowing you to work with specific values within that array

***mask** is just a variable name, not a specific function*

mask = (P <= 0)



P(mask) == [3;-5;-20;0]

P(mask) =

Finding Stuff (indexing)

- Indexing can be done with either an array of logicals (the same size as the array you’re trying to get information from) or an array of doubles.
  - *logical* – The index array is a **MASK** that tells MATLAB which elements to keep or throw away.
  - *double* – each number represents the **position** within an array of the element of interest.

>> primes = [1 3 5 7 9]

>> [isPrime, loc] =
ismember(3,primes)

isPrime →***true*** and loc→***2***

>> [isPrime, loc] =
ismember(primes,3)

isPrime → ***[F T F F F]***

loc → ***[0 1 0 0 0]***

find(isPrime) → ***2***

Refining your code - Vectorizing

Vectorizing your code an make it run **much** faster

```
% log of numbers from .01 to 10
x = .01;
for k = 1:1001
  y(k) = log10(x);
  x = x + .01;
end
```

```
% log of numbers from .01 to 10
x = .01:.01:10
y = log10(x);
```

```
% append ”.new” to all files in direct
files = dir;
for n = 1:numel (files)
  newfiles(n)=...
     {strcat(files(n).name, ’.new’)}
end
```

```
% append ”.new” to all files in direct
files = dir;
newfiles = strcat({files.name},’.new’)
```

Putting it together: Poker Planning

- Start with a clear vision of what goes in and what goes out.
- List the broad steps required to solve the problem
- each broad step is a perfect candidate for a function.

```
%% Deal cards Example
% 1. Find out how many
  players and how many
  cards each.
% 2. Create a deck
% 3. Shuffle deck
% 4. Deal to each player
% 5. Determine Score
```

Putting it together: Poker Skeleton

- Use your outline to create skeletal functions that serve as place-holders for yet-to-be-created functions

```
function  poker(nplayers, ncards)
 % 1. Find out how #  players and # cards each.
% 2. Create a deck
% 3. Shuffle deck
% 4. Deal to each player
% 5. Determine Score

function deck= create_deck()
disp(‘creating a deck!’)
deck = [];

function deck= shuffle_deck(deck)
disp(‘shuffle shuffle’)

function show_cards(cardlist)
disp(‘showing cards’);

function[cards, deck] = deal_cards(ncards, deck)
 disp(‘dealing’)
cards = [];

function score= get_score()
disp(‘Score!’);
score = 1;
```

Putting it together: Poker Program

```
%%Main function, runs the game
function poker(nplayers, ncards)
% plays a round of poker with itself
% N-card stud, no draw
disp('starting game')
deck = create_deck();
deck = shuffle(deck);
for whichPlayer = 1 : nplayers
  [player(whichPlayer).cards, deck] = deal_cards(ncards,
     deck);
  fprintf('\nPlayer %d:\n',whichPlayer);
  show_cards(player(whichPlayer).cards);
  player(whichPlayer).score =
     get_score(player(whichPlayer).cards);
end
winner = determine_winner(player);
```

```
starting game
creating standard deck
shuffle shuffle...
dealing 5 cards
Player 1:
 Ace of Spades
 Queen of Diamonds
 7 of Diamonds
 8 of Spades
 9 of Spades
* High Card : 14

dealing 5 cards
Player 2:
 10 of Clubs
 Queen of Spades
 5 of Diamonds
 2 of Spades
 10 of Hearts
* Pair!

Winner is player # : 2
```

fin