

# Computer Programming with MATLAB



## Lesson 1: Introduction to MATLAB

by

Akos Ledeczi and Mike Fitzpatrick

# Computer Programming with MATLAB



- ▶ Lead instructor: Mike Fitzpatrick
- ▶ Supported by: Akos Ledeczi and Robert Tairas
- ▶ Vanderbilt University School of Engineering
- ▶ Purpose
  - To teach computer programming to people who have little or no experience with programming
- ▶ Level
  - Introductory college course
- ▶ Length:
  - 8 weeks of video lectures
  - Extra week for final homework submission



VANDERBILT  
UNIVERSITY

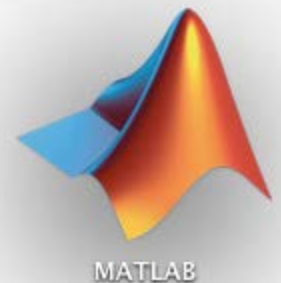
# Programming Languages

- ▶ First high-level language: FORTRAN (1954)
  - To make programming easier, especially to solve numerical problems
- ▶ Today's languages: 1970s–
  - For commercial, “shrink-wrapped”, large, complex programs: C, C++, Java, C#, etc.
  - To make programming easier and to solve numerical problems: MATLAB

# MATLAB History



- ▶ Invented by Prof. Cleve Moler to make programming easy for his students
  - Late 1970s
  - University of New Mexico
- ▶ The MathWorks, Inc. was formed in 1984
  - By Moler and Jack Little
  - One product: MATLAB
- ▶ Today
  - 100 products
  - Over 1,000,000 users
  - Taught in 5,000 universities



# Impact



- ▶ In 2012 the IEEE gave Cleve Moler its annual Computer Pioneer Award
  - “For Improving the quality of mathematical software, making it more accessible, and creating MATLAB”
- ▶ This course is our effort to add to the impact of Moler’s creation by using it to make computer programming more accessible to you.

# It's Computer Science!

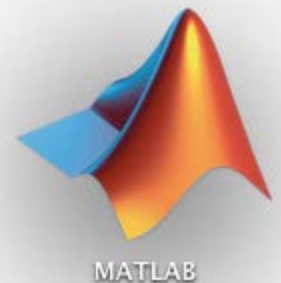


- ▶ We approach MATLAB programming as part of computer science.
- ▶ We use the concepts and vocabulary of computer science to introduce computer programming.
- ▶ This approach gives an idea of how computer scientists think...
- ▶ and makes it easier to learn MATLAB!

# Getting MATLAB



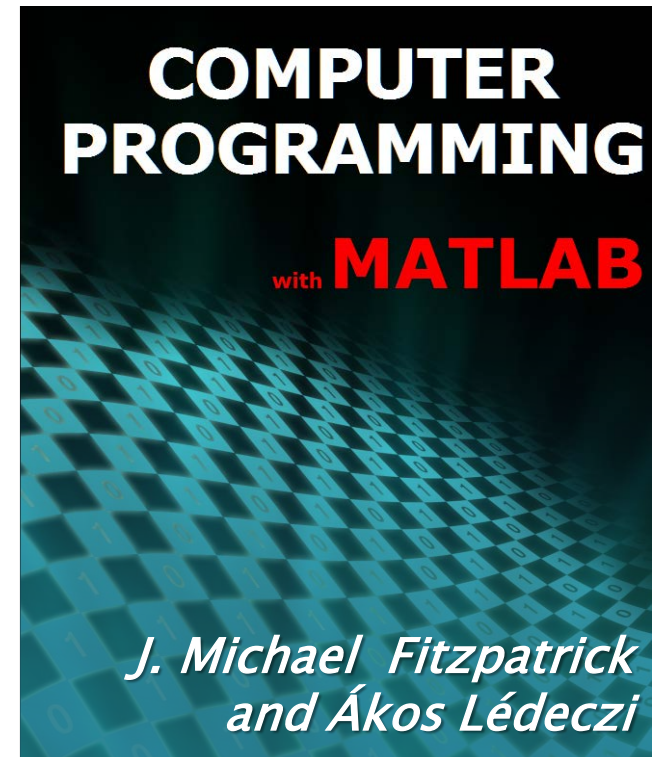
- ▶ [www.mathworks.com](http://www.mathworks.com)
- ▶ Paying for MATLAB
  - Student: \$50
  - MATLAB instructor: Free!
  - Companies: \$100s per user
- ▶ Trying MATLAB for free
  - For everybody: 30 days
  - **For students in this course: free for 12 weeks!**
    - How to get it? Visit class website for info.



# Textbook



- ▶ eBook
- ▶ Title and authors: same as the course!
- ▶ Recommended, but not required.
- ▶ Current version used at Vanderbilt since 2013
- ▶ Previous versions used at Vanderbilt since 2000



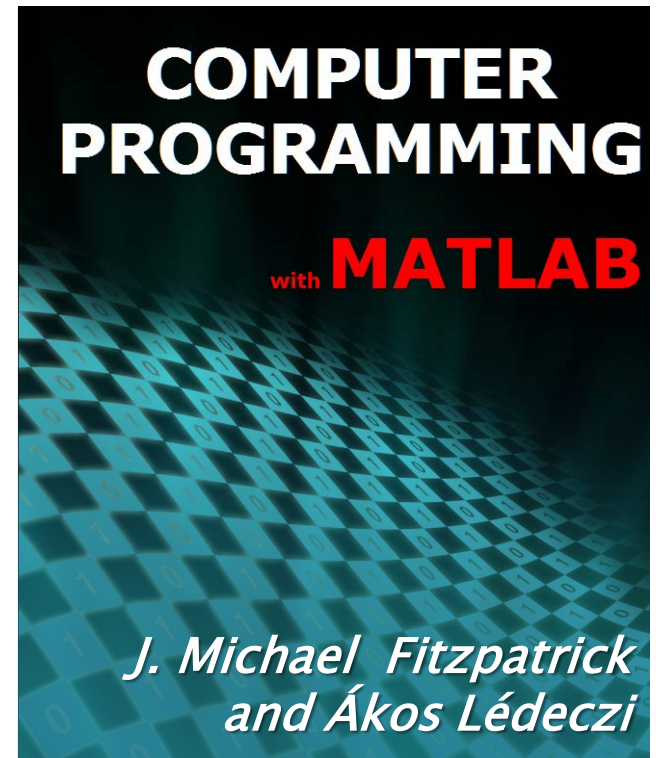




VANDERBILT  
UNIVERSITY

# Getting the Textbook

- ▶ Both Apple (iBook for iPad and Mac) and PDF versions are available for \$9.99 at
  - [cs103.net](http://cs103.net)
- ▶ Three main chapters
  - Chapters 1 and 2 are covered in this course
  - The more advanced concepts in Chapter 3 can be learned on your own from the book after completion of this course



# Computer Programming with MATLAB



## Lesson 2: Matrices and Operators

by

Akos Ledeczi and Mike Fitzpatrick

# Introduction to Arrays and Matrices

## ▶ Array

- Any set of numbers arranged in a rectangular pattern.

### Example—

A page with six rows of four numbers each is a two-dimensional array

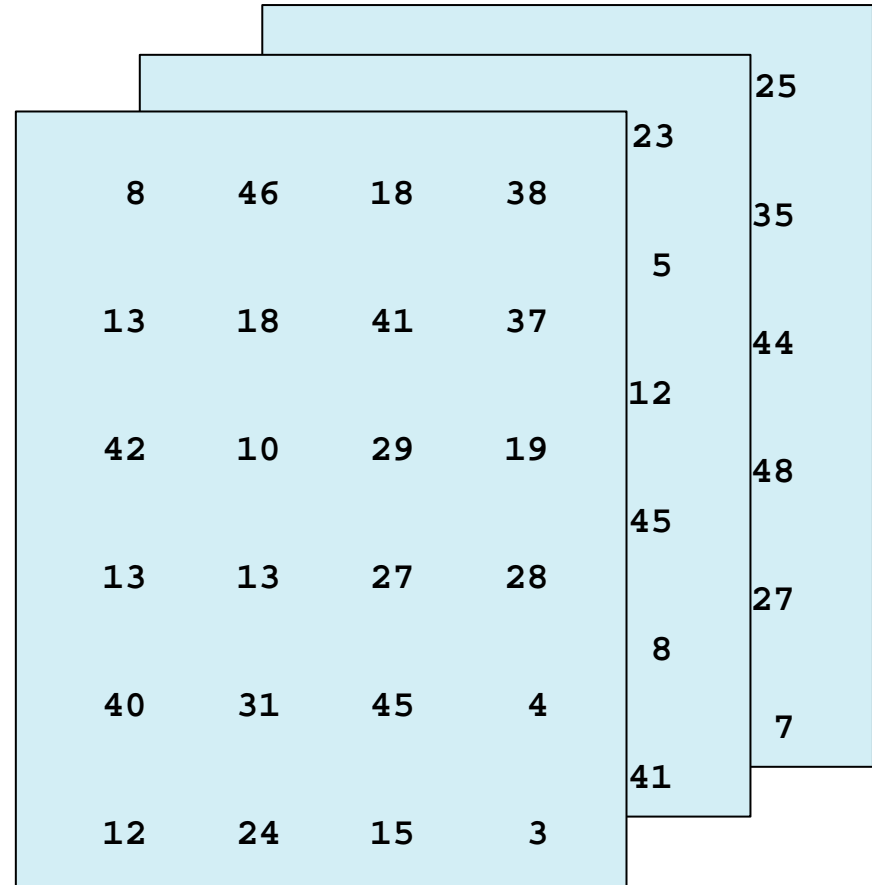
10	14	48	25
24	34	17	35
22	33	29	44
32	8	11	48
35	6	37	27
37	25	13	7

# Introduction to Arrays and Matrices

- ▶ Array
  - Any set of numbers arranged in a rectangular pattern.

Three-dimensional  
Example—

A stack of such  
pages



8	46	18	38	23	25
13	18	41	37	5	35
42	10	29	19	12	44
13	13	27	28	45	48
40	31	45	4	8	27
12	24	15	3	41	7

# Introduction to Arrays and Matrices

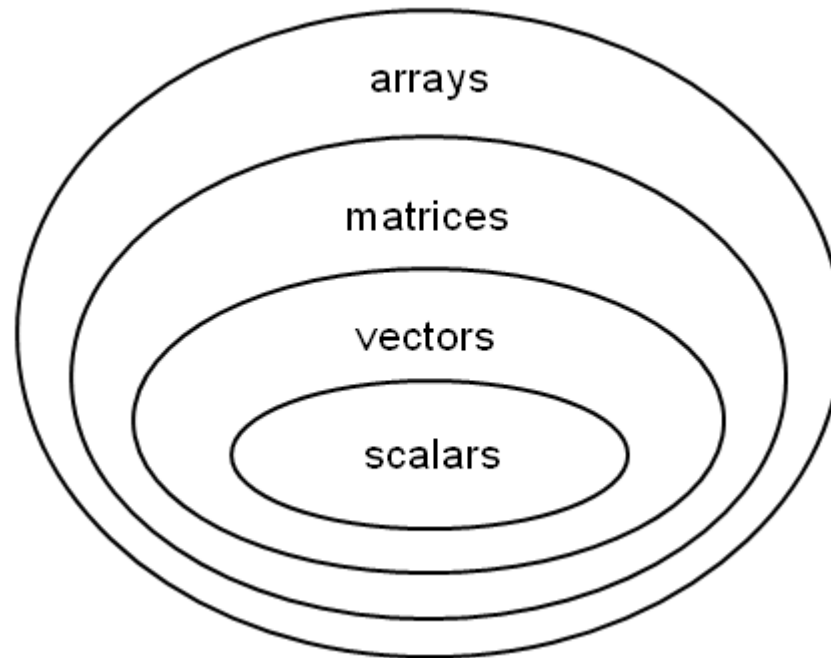


- ▶ Higher dimensions are uncommon
- ▶ The most common have special names:
  - 2D array = “matrix” (plural is “matrices”)
  - 1D array = “vector”
- ▶ Most ingenious part of Cleve Moler’s invention of MATLAB was the way he set it up to deal with matrices.
- ▶ MATLAB stands for “Matrix Laboratory”!

# Arrays and Matrices



VANDERBILT  
UNIVERSITY





VANDERBILT  
UNIVERSITY

# Rows and Columns

```
>> X = [1:4; 5:8; 9:12];
```

1:	1	2	3	4
2:	5	6	7	8
3:	9	10	11	12

rows



VANDERBILT  
UNIVERSITY

# Rows and Columns

```
>> X = [1:4; 5:8; 9:12];
```

1:	2:	3:	4:
1	2	3	4
5	6	7	8
9	10	11	12

columns





VANDERBILT  
UNIVERSITY

# Indexing

```
>> X = [1:4; 5:8; 9:12];
```

```
>> X(2,3)
```

3:

	1	2	3	4
2:	5	6	7	8
	9	10	11	12

```
>> ans =
```

7

# Array Addition

- ▶  $Z = X + Y$  means
  - $Z(m,n) = X(m,n) + Y(m,n)$  for all valid  $m$  and  $n$

$$\begin{array}{rcl}
 Z(1, 1) & = & X(1, 1) + Y(1, 1) \\
 Z(1, 2) & = & X(1, 2) + Y(1, 2) \\
 \dots & & \\
 Z(1, \text{end}) & = & X(1, \text{end}) + Y(1, \text{end})
 \end{array}
 \left. \vphantom{\begin{array}{rcl} Z(1, 1) \\ Z(1, 2) \\ \dots \\ Z(1, \text{end}) \end{array}} \right\} \text{1st row}$$
  

$$\begin{array}{rcl}
 Z(2, 1) & = & X(2, 1) + Y(2, 1) \\
 Z(2, 2) & = & X(2, 2) + Y(2, 2) \\
 \dots & & \\
 Z(2, \text{end}) & = & X(2, \text{end}) + Y(2, \text{end})
 \end{array}
 \left. \vphantom{\begin{array}{rcl} Z(2, 1) \\ Z(2, 2) \\ \dots \\ Z(2, \text{end}) \end{array}} \right\} \text{2nd row}$$
  

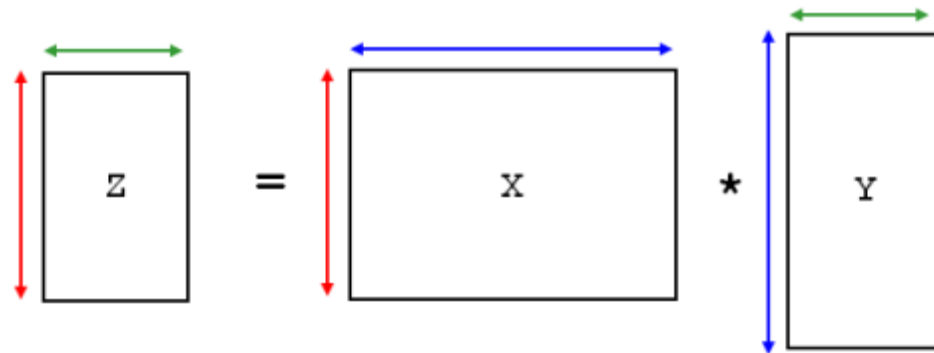
$$\begin{array}{rcl}
 \dots & & \\
 Z(\text{end}, 1) & = & X(\text{end}, 1) + Y(\text{end}, 1) \\
 Z(\text{end}, 2) & = & X(\text{end}, 2) + Y(\text{end}, 2) \\
 \dots & & \\
 Z(\text{end}, \text{end}) & = & X(\text{end}, \text{end}) + Y(\text{end}, \text{end})
 \end{array}
 \left. \vphantom{\begin{array}{rcl} Z(\text{end}, 1) \\ Z(\text{end}, 2) \\ \dots \\ Z(\text{end}, \text{end}) \end{array}} \right\} \text{last row}$$

# Matrix Multiplication

- ▶ Different from Array Multiplication!
- ▶  $Z = X * Y$  means that for all valid  $m$  and  $n$

$$Z(m, n) = \sum_k X(m, k) Y(k, n)$$

- ▶ Not always legal:
  - Inner dimensions of  $X$  and  $Y$  must be the same



# Array Division

- ▶  **$Z = X ./ Y$** 
  - means that for each  $m$  and  $n$ ,  $Z(m,n) = X(m,n)/Y(m,n)$
- ▶  **$Z = X .\ Y$** 
  - means that for each  $m$  and  $n$ ,  $Z(m,n) = Y(m,n)/X(m,n)$
- ▶ Try these out in MATLAB on your own!
- ▶ Matrix division is a complicated concept in linear algebra, so we are not covering it here
  - But you can check out the advanced concepts of the textbook for detailed explanation

# Precedence

- ▶  $x = a + b + c$ 
  - order does not matter with addition
- ▶  $y = c + a * b$  is not the same as
- ▶  $y = (c + a) * b$
- ▶ Multiplication has priority over addition
  - In programming, this is called *precedence*

PRECEDENCE	OPERATOR
0	Parentheses: (...)
1	Exponentiation ^ and Transpose '
2	Unary +, Unary -, and logical negation: ~
3	Multiplication and Division (array and matrix)
4	Addition and Subtraction
5	Colon operator :

Precedence Table

# Associativity

- ▶  $x = a + b + c$
- ▶  $x = a * b * c$ 
  - order does not matter with addition or multiplication
- ▶  $y = a ^ (b ^ c)$  is not the same as
- ▶  $y = (a ^ b) ^ c$
- ▶ In programming, the order in which operators of the same precedence are executed is called *associativity*
- ▶ In MATLAB, it is left to right
- ▶  $y = a ^ b ^ c$  is the same as
- ▶  $y = (a ^ b) ^ c$

# Computer Programming with MATLAB



## Lesson 3: Functions

by

Akos Ledeczki and Mike Fitzpatrick

# Formal Definition

▶ `function` [out\_arg1, out\_arg2, ...] =  
    function\_name (in\_arg1, in\_arg2, ...)

## ▶ Examples

- `function` func
- `function` func(in1)
- `function` func(in1, in2)
- `function` out1 = func
- `function` out1 = func(in1)
- `function` [out1, out2] = func
- `function` [out1, out2] = func(in1, in2)
- ...



# Function names

- ▶ Use meaningful names that tell you something about what your function does
- ▶ Do not use existing names, e.g., plot, sum, sqrt, sin, etc.
  - MATLAB already has these
  - It would get really confusing really fast
  - To check whether a name is already in use, try the built-in exist function. To see how it works, try

```
>> help exist
```

# Advantages of functions

- ▶ Functions allow you to break down large, complex problems to smaller, more manageable pieces
- ▶ Functional decomposition
- ▶ Reusability
- ▶ Generality
  - A function can solve a set of related problems not just a specific one by accepting input arguments.
  - For example, the built-in function `plot` can draw a wide range of figures based on its input

# Helpful Functions

MATLAB provides many hundreds of built-in functions. Here, we provide tables of some of them that might be helpful to you in working problems for this course and later for applying what you learn in this course<sup>1</sup>:

Table 1. Matrix-building functions

FUNCTION	RETURNS AN N-BY-M MATRIX OF
<code>zeros (N,M)</code>	zeros
<code>ones (N,M)</code>	ones
<code>eye (N,M)</code>	zeros except for the diagonal elements that are ones
<code>rand (N,M)</code>	random numbers uniformly distributed in the range from 0 to 1

Table 2. Trigonometric functions

FUNCTION	RETURN
<code>acos (x)</code>	Angle in radians whose cosine equals x
<code>acot (x)</code>	Angle in radians whose cotangent equals x
<code>asin (x)</code>	Angle in radians whose sine equals x
<code>atan (x)</code>	Angle in radians whose tangent equals x
<code>atan2 (y,x)</code>	Four-quadrant angle in radians whose tangent equals y/x
<code>cos (x)</code>	Cosine of x (x in radians)
<code>cot (x)</code>	Cotangent of x (x in radians)
<code>sin (x)</code>	Sine of x (x in radians)
<code>tan (x)</code>	Tangent of x (x in radians)

Table 3. Exponential functions

FUNCTION	RETURNS
<code>exp (x)</code>	$e$ raised to the x power
<code>log (x)</code>	Natural logarithm x
<code>log2 (x)</code>	Base-2 logarithm of x
<code>log10 (x)</code>	Base-10 logarithm of x
<code>sqrt (x)</code>	Square root of x

Table 4. Functions that work on complex numbers

FUNCTION	RETURNS
<code>abs (z)</code>	Absolute value of z
<code>angle (z)</code>	Phase angle of z
<code>conj (z)</code>	Complex conjugate of z
<code>imag (z)</code>	Imaginary part of z
<code>real (z)</code>	Real part of z

---

<sup>1</sup> Tables excerpted from *Computer Programming with MATLAB*, revised edition, by J. Michael Fitzpatrick and Ákos Lédeczi, 2013.

Table 5. Rounding and remainder functions

FUNCTION	RETURNS
<code>fix(x)</code>	Round x towards zero
<code>floor(x)</code>	Round x towards minus infinity
<code>ceil(x)</code>	Round x towards plus infinity
<code>round(x)</code>	Round x towards nearest integer
<code>rem(x, n)</code>	Remainder of $x/n$ (see help for case of noninteger n)
<code>sign(x)</code>	1 if $x > 0$ ; 0 if $x$ equals 0; -1 if $x < 0$

Table 6. Descriptive functions applied to a vector

FUNCTION	RETURNS
<code>length(v)</code>	Number of elements of v
<code>max(v)</code>	Largest element of v
<code>min(v)</code>	Smallest element of v
<code>mean(v)</code>	Mean of v
<code>median(v)</code>	Median element of v
<code>sort(v)</code>	Sorted version of v in ascending order
<code>std(v)</code>	Standard deviation of v
<code>sum(v)</code>	Sum of the elements of v

Table 7. Descriptive functions applied to a two-dimensional matrix

FUNCTION	RETURNS A ROW VECTOR CONSISTING OF
<code>max(M)</code>	Largest element of each column
<code>min(M)</code>	Smallest element of each column
<code>mean(M)</code>	Mean of each column
<code>median(M)</code>	Median of each column
<code>size(M)</code>	Number of rows, number of columns
<code>sort(M)</code>	Sorted version, in ascending order, of each column
<code>std(M)</code>	Standard deviation of each column
<code>sum(M)</code>	Sum of the elements of each column

# Computer Programming with MATLAB



## Lesson 4: Programmer's Toolbox

by

Akos Ledeczki and Mike Fitzpatrick



VANDERBILT  
UNIVERSITY

# Polymorphism

- ▶ Polymorphic: having multiple forms
- ▶ If a function can handle calls when the same input argument has different types: polymorphic function
- ▶ If a function can handle a variable number of input arguments: polymorphic also
- ▶ Many MATLAB functions return an output that has the same shape as the input provided (e.g., `sqrt`)
- ▶ Polymorphism is very powerful



VANDERBILT  
UNIVERSITY

# Random Sequence Initialization

- ▶ When we start MATLAB and call `rand`, it always returns the same exact number: 0.8147 (As an exercise, try it at home!)
- ▶ *Pseudo* random number generator: initialized at startup and it generates the exact same sequence of numbers every time
- ▶ Repeatability for testing the program: good!
- ▶ What if we want “truly” random numbers?
- ▶ Initialize the MATLAB pseudo random number generator with `rng`!

# Debugging

- ▶ It is easy to make mistakes!
- ▶ Syntax errors:
  - MATLAB catches these
- ▶ Semantic errors
  - Some may cause errors that MATLAB can catch
  - Some may cause the wrong result every time
  - Some may only cause problems occasionally (e.g., with a certain combination of inputs):
    - Hard to notice and find the cause of
- ▶ MATLAB has built-in debugger
  - Tool to help find errors (bugs)



# Computer Programming with MATLAB



## Lesson 5: Selection

by

Akos Ledeczki and Mike Fitzpatrick



VANDERBILT  
UNIVERSITY

# Control Flow

- ▶ Sequential control
  - Sequence of commands executed one after the other
- ▶ MATLAB interpreter
  - Part of the MATLAB program that interprets and executes the various commands
  - Sequential control: default
- ▶ Control construct
  - A method by which the interpreter selects the next command to execute
  - Sequential control: default
  - Selection or Branching

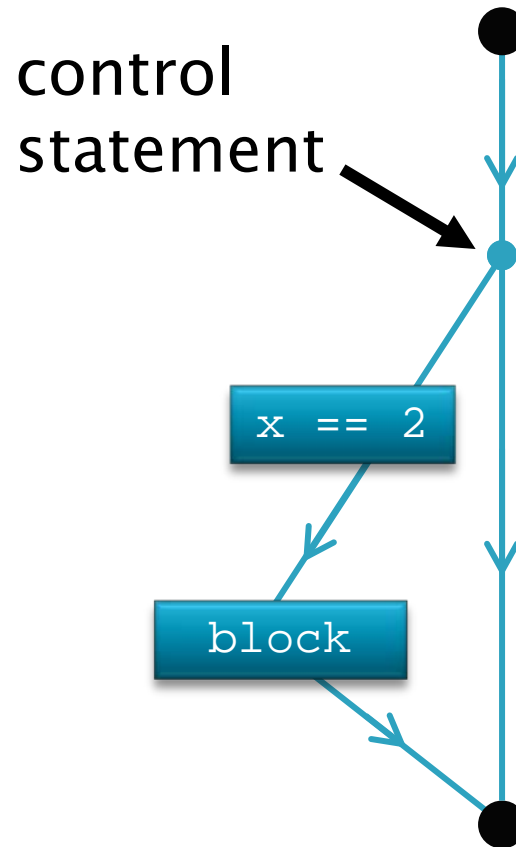
# if-statement

- ▶ Most common selection construct: if-statement
- ▶ Example:

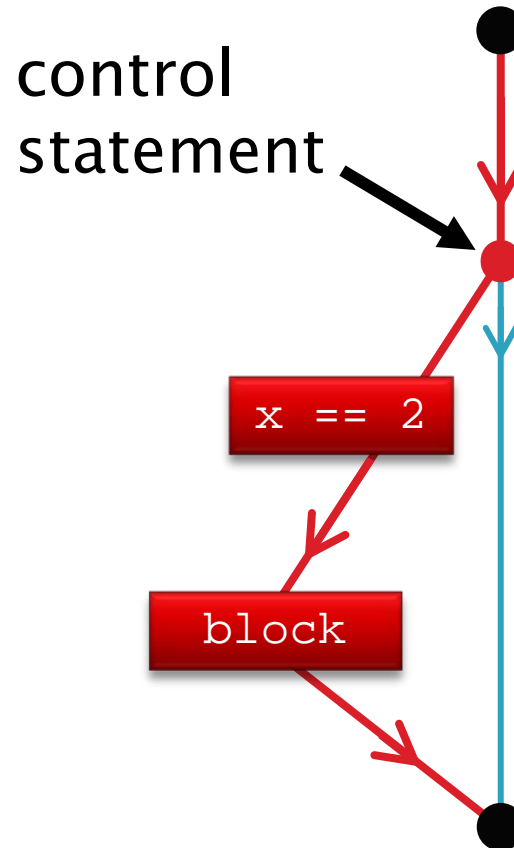
```
function guess_my_number(x)
    if x == 2
        fprintf('Congrats! You guessed my number.\n');
    end
```

- ▶ Begins with control statement
  - **if** keyword followed by a condition
- ▶ Ends with statement: **end**
- ▶ In between: statements to be executed if and only if condition is true

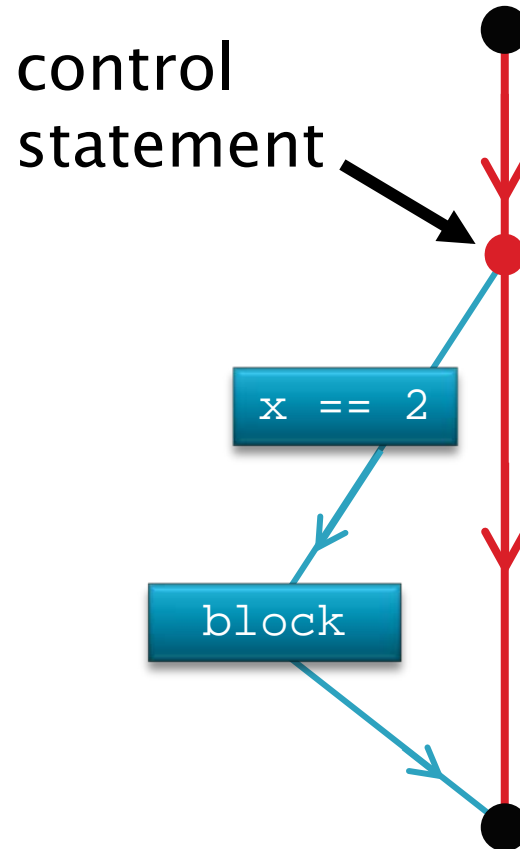
# Schematic of an if-statement



# Condition: **true**



# Condition: **false**

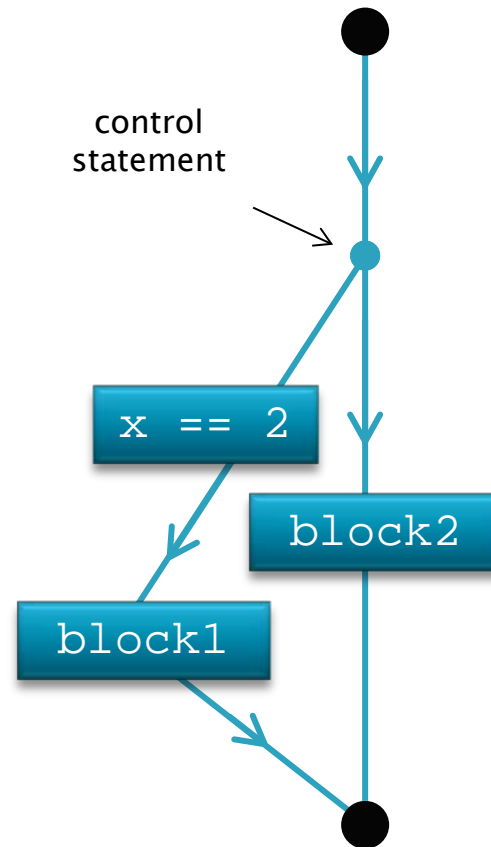


# if-else-statement

- ▶ Executing a different set of statements based on the condition:

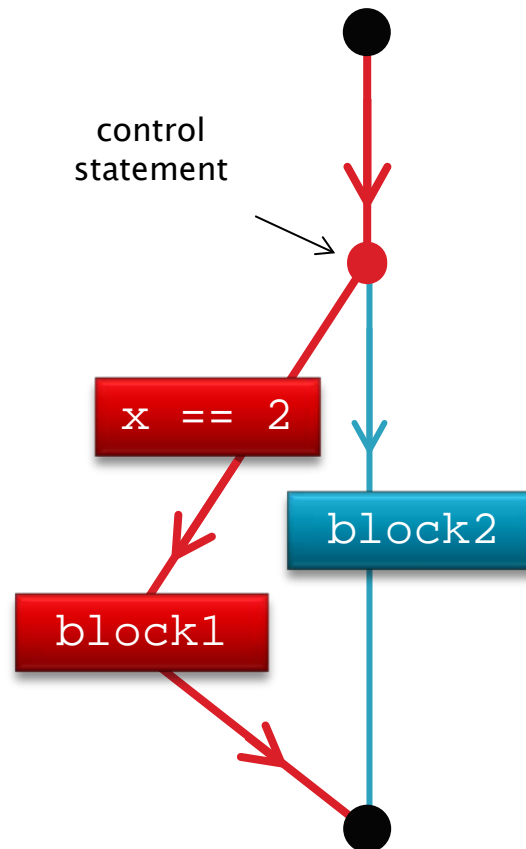
```
function guess_my_number(x)
if x == 2
    fprintf('Congrats! You guessed my number!\n');
else
    fprintf('Not right, but a good guess.\n');
end
```

# if-else-statement

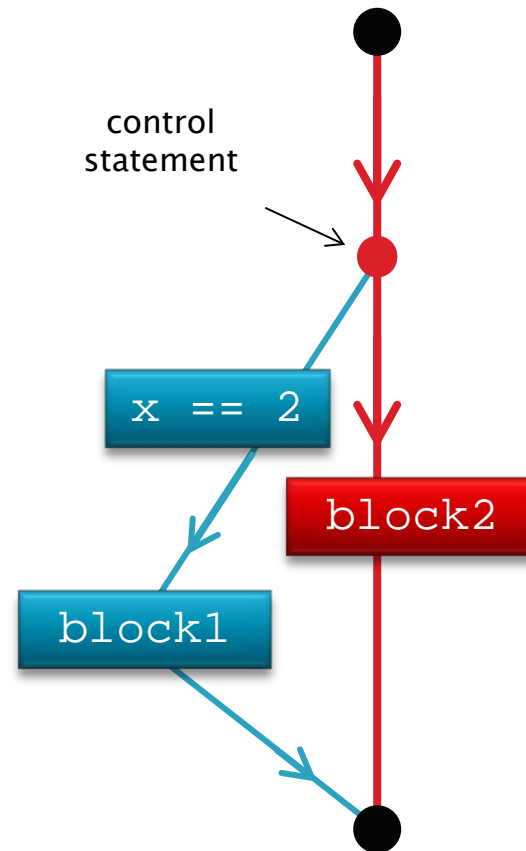




# Condition: **true**



# Condition: **false**





VANDERBILT  
UNIVERSITY

# if-statement summary

▶ if-statement:

```
if conditional  
    block  
end
```

# Relational operators

- ▶ Produces a result that depends on the relation between its two operands
- ▶ It can appear outside if-statements!

OPERATOR	MEANING
==	is equal to
~=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

# Logical operators

- ▶ Logical values:
  - Non-zero: true
  - Zero: false
  - MATLAB returns 1 for true
- ▶ How to combine logical values?
- ▶ Logical operators:

OPERATOR	MEANING
<b>&amp;&amp;</b>	and
<b>  </b>	or
<b>~</b>	not

# Truth table

- ▶ not:
  - flips the value of its (single) operand
- ▶ and:
  - true if and only if both of its operands are true
- ▶ or:
  - false if and only if both of its operands are false

INPUTS		&&	
false	false	0	0
false	true	0	1
true	false	0	1
true	true	1	1

# Truth table

- ▶ not:
  - flips the value of its (single) operand
- ▶ and:
  - true if and only if both of its operands are true
- ▶ or:
  - false if and only if both of its operands are false

INPUTS		&&	
0	0	0	0
0	nonzero	0	1
nonzero	0	0	1
nonzero	nonzero	1	1

# Precedence revisited



VANDERBILT  
UNIVERSITY

PRECEDENCE	OPERATOR
0	Parentheses: (...)
1	Exponentiation $^$ and Transpose $'$
2	Unary $+$ , Unary $-$ , and logical negation: $\sim$
3	Multiplication and Division (array and matrix)
4	Addition and Subtraction
5	Colon operator $:$
6	Relational operators: $<$ , $<=$ , $>$ , $>=$ , $==$ , $\sim=$
7	Element-wise logical "and": $\&$
8	Element-wise logical "or": $ $
9	logical "and": $\&\&$
10	logical "or": $  $





VANDERBILT  
UNIVERSITY

# Precedence revisited

```
>> help precedence
```

1. transpose (.'), power (.^), complex conjugate  
transpose ('), matrix power (^)
2. unary plus (+), unary minus (-), logical negation (~)
3. multiplication (.\*), right division (./), left  
division (.\), matrix multiplication (\*), matrix right  
division (/), matrix left division (\)
4. addition (+), subtraction (-)
5. colon operator (:)
6. less than (<), less than or equal to (<=),  
greater than(>), greater than or equal to (>=),  
equal to (==), not equal to (~=)
7. element-wise logical AND (&)
8. element-wise logical OR (|)
9. short-circuit logical AND (&&)
10. short-circuit logical OR (||)

# Nested if-statements

- ▶ if-statements can contain other if-statements
- ▶ Consider the example with a single if-elseif-else statement:

```
function ultimate_question(x)
if x == 42
    fprintf('Wow! You answered the question.\n');
elseif x < 42
    fprintf('Too small. Try again.\n');
else
    fprintf('Too big. Try again.\n');
end
```



VANDERBILT  
UNIVERSITY

# Nested if-statements

- ▶ Here is a version with nesting:

```
function ultimate_question_nested(x)
    if x == 42
        fprintf('Wow! You answered the question.\n');
    else
        if x < 42
            fprintf('Too small. Try again.\n');
        else
            fprintf('Too big. Try again.\n');
        end
    end
end
```

# Nested if-statements

- ▶ Here is another version with nesting:

```
function ultimate_question_nested2(x)
if x <= 42
    if x == 42
        fprintf('Wow! You answered the question.\n');
    else
        fprintf('Too small. Try again.\n');
    end
else
    fprintf('Too big. Try again.\n');
end
```



VANDERBILT  
UNIVERSITY

# Polymorphic functions

- ▶ Functions that behave differently based on
  - Number of input or output arguments
  - Type of input or output arguments
- ▶ Many built-in functions are polymorphic (sqrt, max, size, plot, etc.)
- ▶ How do we make our functions polymorphic?



VANDERBILT  
UNIVERSITY

# Number of arguments

- ▶ Two built-in functions:
  - **nargin**: returns the number of actual input arguments that the function was called with
  - **nargout**: returns the number of output arguments that the function caller requested

# Example: multiplication table

```
function [table summa] = multable(n, m)
```

- ▶ The function `multable` returns an  $n$ -by- $m$  multiplication table in the output argument `table`
- ▶ Optionally, it can return the sum of all elements in the output argument `summa`
- ▶ If `m` is not provided, it returns an  $n$ -by- $n$  matrix

# Example

```
function [table summa] = multable(n, m)
```

```
if nargin < 2
```

```
    m = n;
```

```
end
```

```
table = (1:n)' * (1:m);
```

```
if nargout == 2
```

```
    summa = sum(table(:));
```

```
end
```





VANDERBILT  
UNIVERSITY

# Robustness

- ▶ A function declaration specifies:
  - Name of the function,
  - Number of input arguments, and
  - Number of output arguments
- ▶ Function code and documentation specify:
  - What the function does, and
  - The type of the arguments
  - What the arguments represent
- ▶ Robustness
  - A function is robust if it handles erroneous input and output arguments, and
  - Provides a meaningful error message

# Example

```
function [table summa] = multable(n, m)

if nargin < 1
    error('must have at least one input argument');
end
if nargin < 2
    m = n;
elseif ~isscalar(m) || m < 1 || m ~= fix(m)
    error('m needs to be a positive integer');
end
if ~isscalar(n) || n < 1 || n ~= fix(n)
    error('n needs to be a positive integer');
end

table = (1:n)' * (1:m);

if nargout == 2
    summa = sum(table(:));
end
```

# Comments

- ▶ Extra text that is not part of the code
- ▶ MATLAB disregards it
- ▶ Anything after a % is a comment until the end of the line
- ▶ Purpose:
  - Extra information for human reader
  - Explain important or complicated parts of the program
  - Provide documentation of your functions
- ▶ Comments right after the function declaration are used by the built-in `help` function

# Example

```
function [table summa] = multable(n, m)

%MULTABLE multiplication table.
% T = MULTABLE(N) returns an N-by-N matrix
% containing the multiplication table for
% the integers 1 through N.
% MULTABLE(N,M) returns an N-by-M matrix.
% Both input arguments must be positive
% integers.
% [T SM] = MULTABLE(...) returns the matrix
% containing the multiplication table in T
% and the sum of all its elements in SM.

if nargin < 1
    error('must have at least one input argument');
end

...
```

>> help multable



VANDERBILT  
UNIVERSITY

# Persistent variable

- ▶ Variables:
  - Local
  - Global
  - Persistent
- ▶ Persistent variable:
  - It's a local variable, but its value persists from one call of the function to the next.
  - Relatively rarely used
  - None of the bad side effects of global variables.

# Computer Programming with MATLAB



## Lesson 6: Loops

by

Akos Ledeczi and Mike Fitzpatrick

# Loops

- ▶ The loop is a new control construct that makes it possible to repeat a block of statements a number of times.
- ▶ We have already used loops without knowing it:  

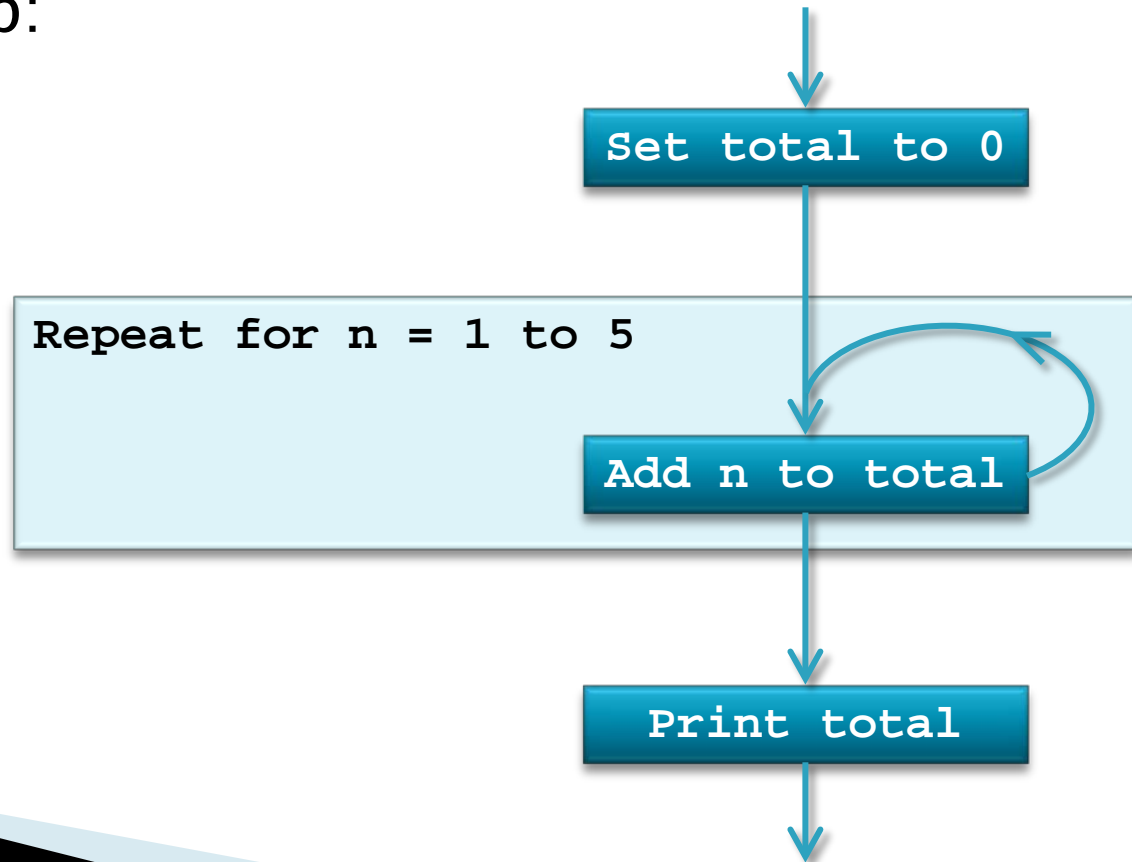
```
>> n = 1:5;  
>> total = sum(n);
```
- ▶ MATLAB uses loops internally both to compute the result of the colon operator and to compute the sum of the elements of the vector **n** above.
- ▶ Implicit loop



VANDERBILT  
UNIVERSITY

# Schematic of a loop

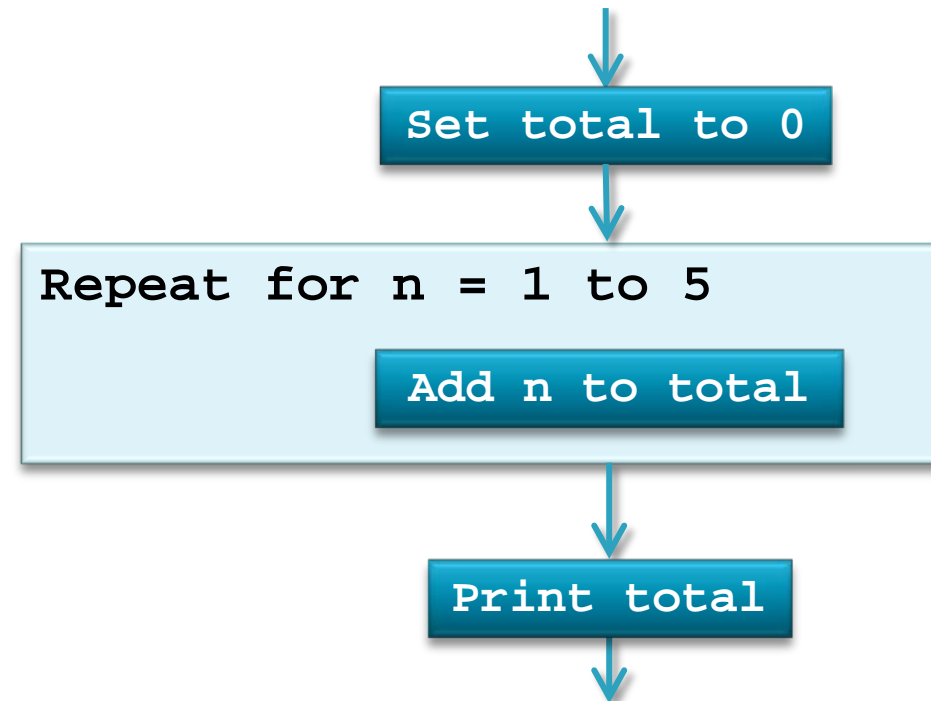
- ▶ Let's compute the sum of 1 through 5 *without* using the built-in sum function!
- ▶ Use a loop:





# Execution of a loop

- ▶ Set total to 0
- ▶ Set n to 1
- ▶ Execute Add n to total (total equals 1)
- ▶ Set n to 2
- ▶ Execute Add n to total (total equals 3)
- ▶ Set n to 3
- ▶ Execute Add n to total (total equals 6)
- ▶ Set n to 4
- ▶ Execute Add n to total (total equals 10)
- ▶ Set n to 5
- ▶ Execute Add n to total (total equals 15)
- ▶ Print total



# for-loop

- ▶ MATLAB implementation using a for-loop:

```
total = 0;  
for n = 1:5  
    total = total + n;  
end  
fprintf('total equals %d\n',total);
```



VANDERBILT  
UNIVERSITY

# Parts of a for-loop

```
total = 0;
loop { for n = 1:5      _____ control statement
      total = total + n; _____ body
      end
      fprintf('total equals %d\n',total);
```

# Colon operator is not required

- ▶ Here is another example:

```
list = rand(1,5); % assigns a row vector of random numbers
for x = list
    if x > 0.5
        fprintf('Random number %f is large.\n',x)
    else
        fprintf('Random number %f is small.\n',x)
    end
end
```

```
Random number 0.141890 is small.
Random number 0.421760 is small.
Random number 0.915740 is large.
Random number 0.792210 is large.
Random number 0.959490 is large.
```

# Example revisited

- ▶ Notice that we do not need the list variable at all:

```
for x = rand(1,5)
    if x > 0.5
        fprintf('Random number %f is large.\n',x)
    else
        fprintf('Random number %f is small.\n',x)
    end
end
```

# Observations

- ▶ The values assigned to the loop index do not have to be
  - integers,
  - regularly spaced, or
  - assigned in increasing order,
- ▶ In fact, they do not have to be scalars either:
  - The loop index will be assigned the columns of the array
- ▶ Any other control construct can be used in the body of the for-loop
  - if-statements
  - other loops
  - etc.



VANDERBILT  
UNIVERSITY

# while-loop

- ▶ for-loops work well when we know the number of necessary iterations before entering the loop
- ▶ Consider this problem:
  - Starting from 1, how many consecutive positive integers do we need to add together to exceed 50?
  - The only way to solve this with a for-loop is to guess a large enough number for the number of iterations and then use a break statement.
  - There is a better solution: a while-loop!



VANDERBILT  
UNIVERSITY

# while-loop example

```
function [n total] = possum(limit)
total = 0;
n = 0;
while total <= limit
    n = n + 1;
    total = total + n;
end
fprintf('sum: %d    count: %d\n', total, n);
```





VANDERBILT  
UNIVERSITY

# while-loop example

```
function [n total] = possum(limit)
total = 0;
n = 0;
while total <= limit
    n = n + 1;
    total = total + n;
end
fprintf('sum: %d    count: %d\n', total, n);
```

```
>> possum(50)
sum: 55    count: 10
```

```
ans =
```

```
10
```

# while-loop example

```
function [n total] = possum(limit)
total = 0;
n = 0;
loop { while total <= limit      _____ control statement
      [ n = n + 1;
        total = total + n; ]    _____ body
      end
      fprintf('sum: %d    count: %d\n', total, n);
```

# General form

```
while conditional  
    block  
end
```

```
if conditional  
    block  
end
```

## ► Difference:

- while condition is evaluated repeatedly
- block is executed repeatedly as long as condition is true



VANDERBILT  
UNIVERSITY

# Logical indexing

- ▶ Problem: given a vector,  $v$ , of scalars, create a second vector,  $w$ , that contains only the non-negative elements of  $v$



VANDERBILT  
UNIVERSITY

# Logical indexing

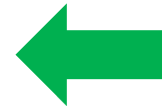
- ▶ Traditional solution:

```
w = [];  
jj = 0;  
for ii = 1:length(v)  
    if v(ii) >= 0  
        jj = jj + 1;  
        w(jj) = v(ii);  
    end  
end
```

# Example revisited

- ▶ MATLAB provides a more elegant solution:

```
w = [];  
for ii = 1:length(v)  
    if v(ii) >= 0  
        w = [w v(ii)];  
    end  
end
```



# Example with logical indexing

- ▶ The ultimate solution needs only a single line:

```
w = v(v >= 0);
```

- ▶ This is an example of logical indexing.
- ▶ To understand why and how this works, we need to introduce logical arrays

# Computer Programming with MATLAB



## Lesson 7: Data Types

by

Akos Ledeczki and Mike Fitzpatrick





VANDERBILT  
UNIVERSITY

# The Limitation of Computers

- ▶ Real numbers in mathematics:
  - Can be infinitely large
  - Have infinitely fine resolution
- ▶ Computers: Finite memory
  - Upper limit on the largest number that can be represented
  - Lower limit on the absolute value of any non-zero number
- ▶ The set of values that can be represented by a MATLAB variable is finite.

# Data Types

- ▶ MATLAB: many different data types
- ▶ A data type is defined by:
  - Set of values
  - Set of operations that can be performed on those values
- ▶ MATLAB:
  - All elements of a given array must be of the same type
  - Elementary type
- ▶ Type of a MATLAB array is defined by
  - Number of dimensions
  - Size in each dimension
  - Elementary type

# Numerical Types

## ▶ double

- Default type in MATLAB
- Floating point representation
  - Example:  $12.34 = 1234 * 10^{-2}$
  - Mantissa and exponent
- 64 bits (8 bytes)

## ▶ single

- 32-bit floating point

## ▶ Integer types

- Signed, unsigned
- 8-, 16-, 32-, 64-bit long

# Range of Values

DATA TYPE	RANGE OF VALUES
int8	$-2^7$ to $2^7-1$
int16	$-2^{15}$ to $2^{15}-1$
int32	$-2^{31}$ to $2^{31}-1$
int64	$-2^{63}$ to $2^{63}-1$
uint8	0 to $2^8-1$
uint16	0 to $2^{16}-1$
uint32	0 to $2^{32}-1$
uint64	0 to $2^{64}-1$
single	$-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$ , Inf, NaN
double	$-1.79 \times 10^{308}$ to $1.79 \times 10^{308}$ , Inf, NaN

# Range of Values

DATA TYPE	RANGE OF VALUES
int8	$-2^7$ to $2^7-1$
int16	$-2^{15}$ to $2^{15}-1$
int32	$-2^{31}$ to $2^{31}-1$
int64	$-2^{63}$ to $2^{63}-1$
uint8	0 to $2^8-1$
uint16	0 to $2^{16}-1$
uint32	0 to $2^{32}-1$
uint64	0 to $2^{64}-1$
single	$-3.4 \times 10^{38}$ to $3.4 \times 10^{38}$ , Inf, NaN
double	$-1.79 \times 10^{308}$ to $1.79 \times 10^{308}$ , Inf, NaN

**Inf :**

“Infinity”

**NaN:**

“Not-a-number”



VANDERBILT  
UNIVERSITY

# Useful functions

## ▶ Type check:

- `class`
- `isa`

```
>> isa(x, 'double')
```

## ▶ Range check:

- `intmax`, `intmin`
- `realmax`, `realmin`

```
>> intmax('uint32')
```

## ▶ Conversion:

- Name of function = name of desired data type
- `int8(x)`, `uint32(x)`, `double(x)`, etc.

# Operators

- ▶ Arithmetic operators
  - Operands of the same data type: No problem
  - Different data types:
    - “mixed-mode arithmetic”
    - Many rules and restrictions
- ▶ Relational operators
  - Different data types are always allowed
  - Result is always of logical type

# Strings

- ▶ Text: string
- ▶ We have used them already:
  - Argument to `fprintf` and other functions
- ▶ String: vector of `char-s`
- ▶ Numerical type
  - Uses an encoding scheme
  - Each character is represented by a number
  - ASCII scheme



# ASCII code



VANDERBILT  
UNIVERSITY

- ▶ American Standard Code for Information Interchange
- ▶ Developed in the 1960's
- ▶ 7 bits, 128 characters:
  - Latin alphabet
  - Digits
  - Punctuation
  - Special characters
- ▶ Newer schemes with far more characters
- ▶ ASCII is a subset of them

(nul)	0	(sp)	32	@	64	`	96
(soh)	1	!	33	A	65	a	97
(stx)	2	"	34	B	66	b	98
(etx)	3	#	35	C	67	c	99
(eot)	4	\$	36	D	68	d	100
(enq)	5	%	37	E	69	e	101
(ack)	6	&	38	F	70	f	102
(bel)	7	'	39	G	71	g	103
(bs)	8	(	40	H	72	h	104
(ht)	9	)	41	I	73	i	105
(nl)	10	*	42	J	74	j	106
(vt)	11	+	43	K	75	k	107
(np)	12	,	44	L	76	l	108
(cr)	13	-	45	M	77	m	109
(so)	14	.	46	N	78	n	110
(si)	15	/	47	O	79	o	111
(dle)	16	0	48	P	80	p	112
(dc1)	17	1	49	Q	81	q	113
(dc2)	18	2	50	R	82	r	114
(dc3)	19	3	51	S	83	s	115
(dc4)	20	4	52	T	84	t	116
(nak)	21	5	53	U	85	u	117
(syn)	22	6	54	V	86	v	118
(etb)	23	7	55	W	87	w	119
(can)	24	8	56	X	88	x	120
(em)	25	9	57	Y	89	y	121
(sub)	26	:	58	Z	90	z	122
(esc)	27	;	59	[	91	{	123
(fs)	28	<	60	\	92		124
(gs)	29	=	61	]	93	}	125
(rs)	30	>	62	^	94	~	126
(us)	31	?	63	_	95	(del)	127

# Exercise

- ▶ Print out the visible characters of the ASCII table:

```
function char_codes
for ii = 33:126
    fprintf('%s',char(ii));
end
fprintf('\n');
```

# String functions



VANDERBILT  
UNIVERSITY

FUNCTION	DESCRIPTION
char	converts type to char
findstr	finds the positions of a substring in a string
ischar	returns 1 if argument is a character array and 0 otherwise
isletter	finds letters in string
isspace	finds spaces, newlines, and tabs in string
isstrprop	finds characters of specified type in string
num2str	converts number to string
length	determines the number of letters in string
lower	converts string to lower case
sprintf	writes formatted data to string (compare with fprintf)
strcmp	compares strings
strcmpi	like strcmp but independent of case
strmatch	search array for rows that begin with specified string
strncmp	like strcmp but compares only first n characters
strncmpi	like strncmp but independent of case
str2num	converts string to number
upper	converts string to upper case

# Structs

- ▶ An array must be homogeneous:
  - It cannot contain elements of multiple types.
- ▶ A struct can be heterogeneous:
  - It can contain multiple types.
- ▶ A struct is different from an array:
  - fields, not elements
  - field names, not indices
  - Fields in the same struct can have different types.
- ▶ Versatility inside:
  - A field of a struct can contain another struct.



VANDERBILT  
UNIVERSITY

# Structs in action

```
>> r.ssn = 12345678
r =
    ssn: 12345678
>> class(r)
ans =
    struct
>> class(r.ssn)
ans =
    double
>> r.name = 'Homer Simpson'
r =
    ssn: 12345678
    name: 'Homer Simpson'
>> r.address.street = '742 Evergreen Terrace'
r =
    ssn: 12345678
    name: 'Homer Simpson'
    address: [1x1 struct]
```



VANDERBILT  
UNIVERSITY

# Structs

- ▶ An array must be homogeneous:
  - It cannot contain elements of multiple types.
- ▶ A struct can be heterogeneous:
  - It can contain multiple types.
- ▶ A struct is different from an array:
  - fields, not elements
  - field names, not indices
  - Fields in the same struct can have different types.
- ▶ **Versatility inside:**
  - A field of a struct can contain another struct.
  - **Structs can hold arrays, and arrays can hold structs.**

# Struct functions

FUNCTION	DESCRIPTION
getfield	returns the value of a field whose name is specified by a string
isfield	true if a struct has a field whose name is specified by a string
isstruct	returns true if argument is of type struct
orderfields	changes the order of the fields in a struct
rmfield	removes from a struct a field whose name is specified by a string
setfield	assigns a value to a field whose name is specified by a string -or- if the field is not present, adds it and assigns it the value
struct	create a struct with fields whose name are specified by strings

# Pointers

- ▶ How to store a page of text?
  - Each line should be a separate string
  - Cannot use an array of chars:
    - Each line would have to have the same length
  - A vector of objects with each referring to one line
- ▶ Pointer
  - Each variable (scalar, vector, array, etc.) is stored in the computer memory.
  - Each memory location has a unique address.
  - A pointer is a variable that stores an address.
  - MATLAB calls a pointer a “cell”.



# Cells

- ▶ MATLAB has a restrictive pointer model
  - Strict rules on what can be done with cells
  - Harder to make mistakes
- ▶ But it is a powerful way to store heterogeneous data
  - Cell arrays
  - Used more frequently than structs
- ▶ New syntax:
  - To access the data a cell points to, use: { }

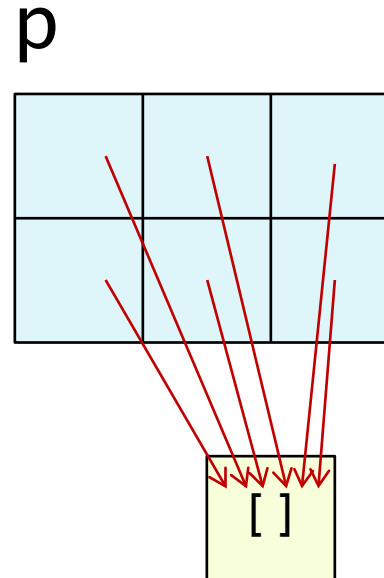
# Cell array example

```
>> p = cell(2,3)
```

```
p =
```

```
    []    []    []  
    []    []    []
```

```
>>
```





VANDERBILT  
UNIVERSITY

# Cell array example

```
>> p = cell(2,3)
```

```
p =
```

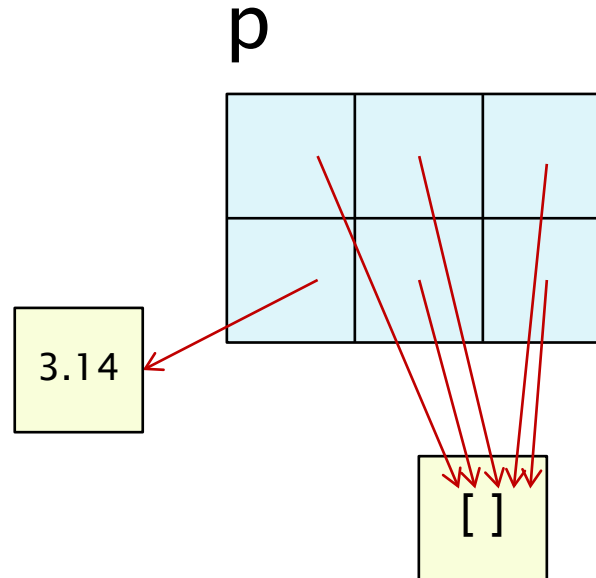
```
    []    []    []  
    []    []    []
```

```
>> p{2,1} = pi
```

```
p =
```

```
    []    []    []  
[3.14]    []    []
```

```
>>
```

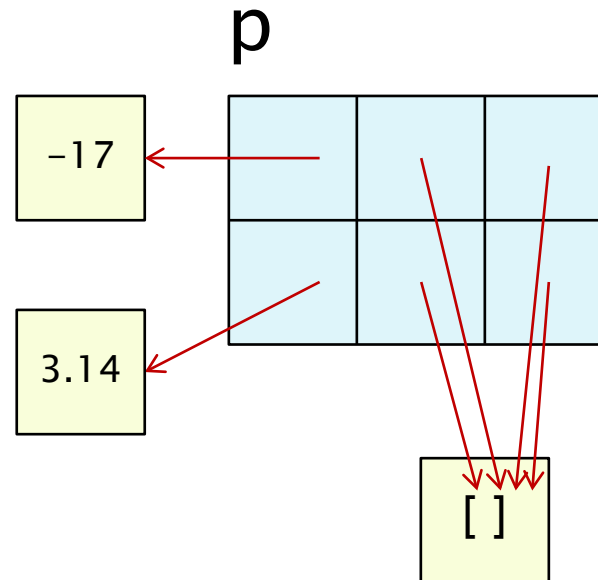




VANDERBILT  
UNIVERSITY

# Cell array example

```
>> p = cell(2,3)
p =
    []    []    []
    []    []    []
>> p{2,1} = pi
p =
    []    []    []
    [3.14] []    []
>> p{1,1} = int8(-17)
p =
    [-17]    []    []
    [3.14]    []    []
>>
```

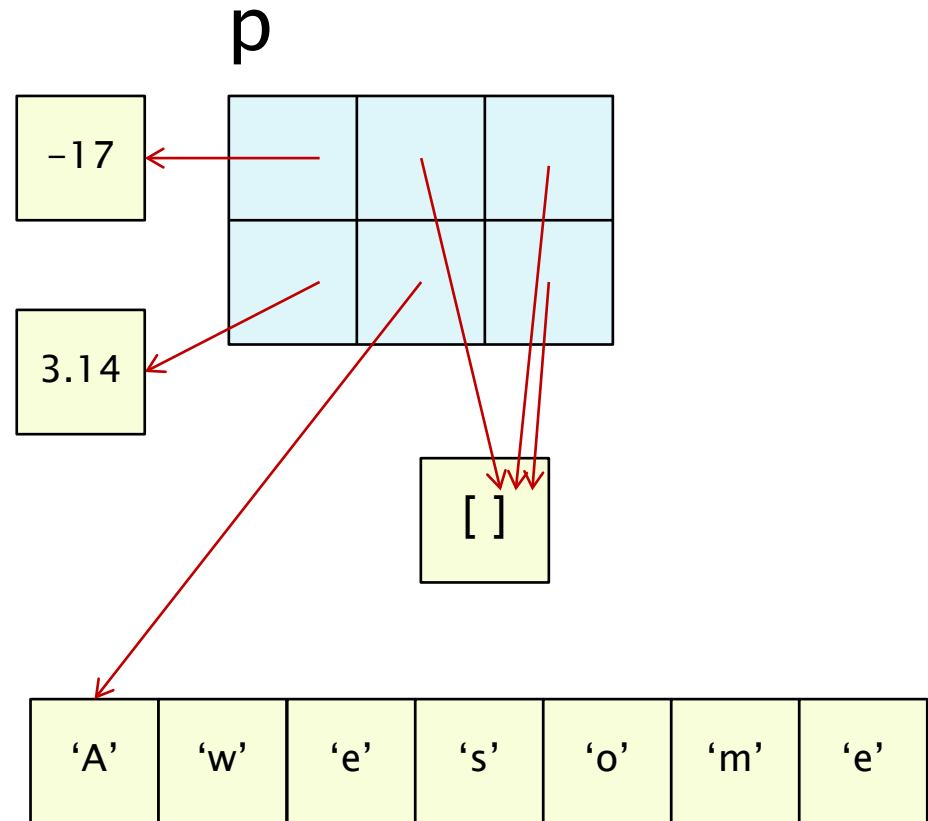




VANDERBILT  
UNIVERSITY

# Cell array example

```
>> p = cell(2,3)
p =
    []    []    []
    []    []    []
>> p{2,1} = pi
p =
    []    []    []
    [3.14] []    []
>> p{1,1} = -17
p =
    [-17] []    []
    [3.14] []    []
>> p{2,2} = 'Awesome'
P =
    [-17] []    []
    [3.14] 'Awesome' []
```





VANDERBILT  
UNIVERSITY

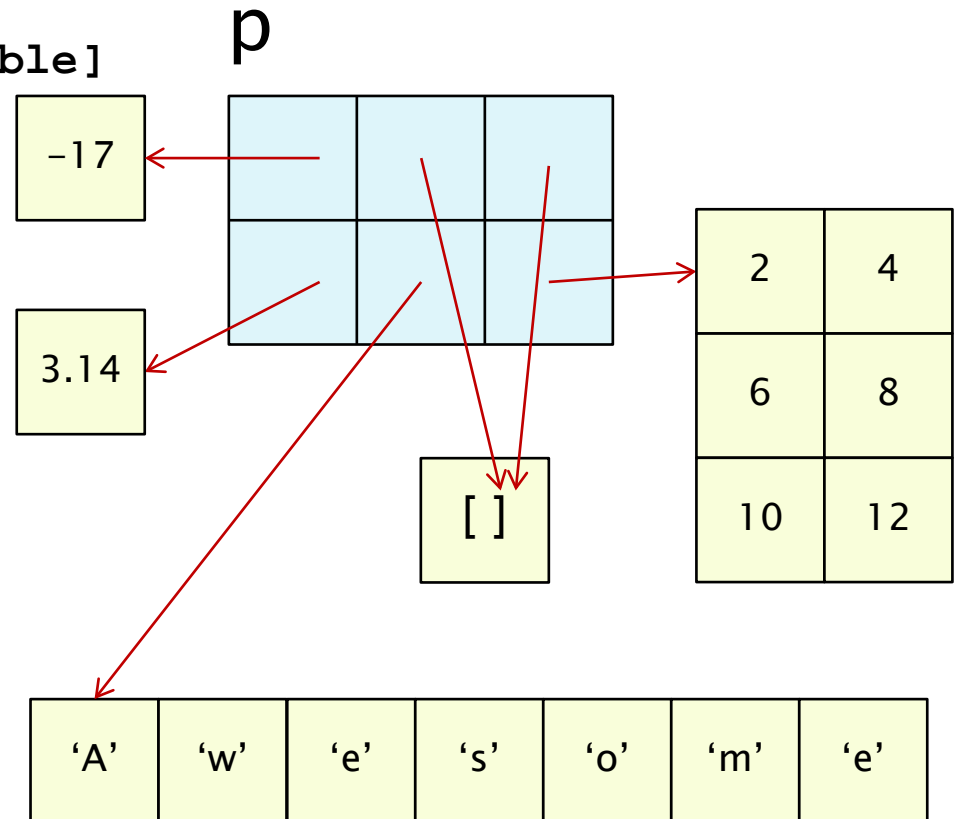
# Cell array example

```
>> p{2,3} = [2 4; 6 8; 10 12]
```

```
P =
```

```
    [-17]    []    []  
    [3.14] 'Awesome' [3x2 double]
```

```
>>
```





VANDERBILT  
UNIVERSITY

# Cell array example

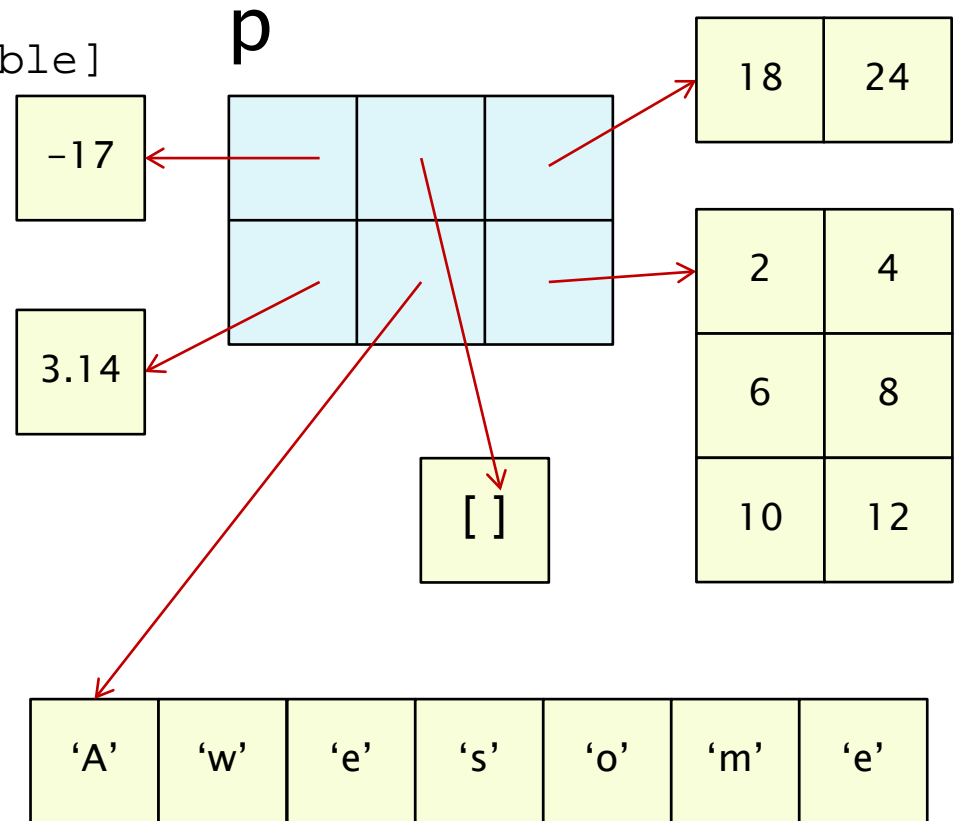
```
>> p{2,3} = [2 4; 6 8; 10 12]
```

```
P =
```

```
    [-17]    []    []  
    [3.14] 'Awesome' [3x2 double]
```

```
>> p{1,3} = sum(p{2,3});
```

```
>>
```





VANDERBILT  
UNIVERSITY

# Cell array example

```
>> p{2,3} = [2 4; 6 8; 10 12]
```

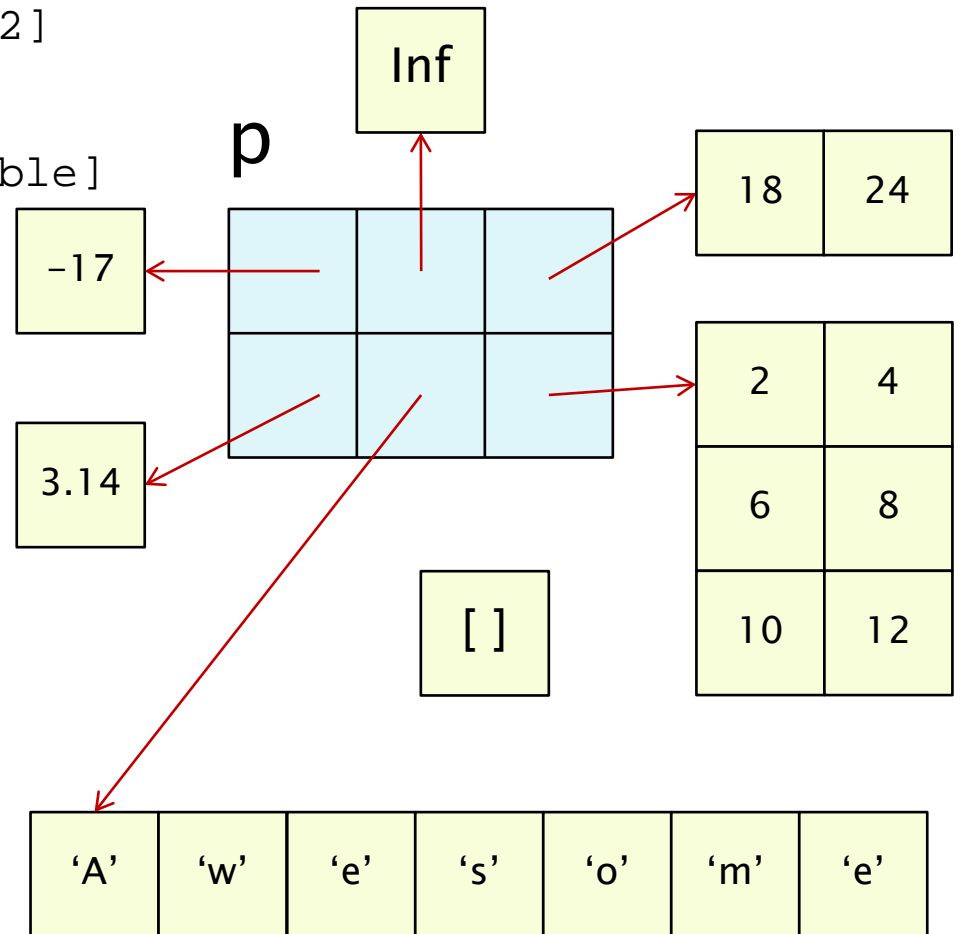
```
P =
```

```
    [-17]    []    []  
    [3.14] 'Awesome' [3x2 double]
```

```
>> p{1,3} = sum(p{2,3});
```

```
>> P{1,2} = 1/0;
```

```
>>
```







VANDERBILT  
UNIVERSITY

# Cell array example

```
>> p{2,3} = [2 4; 6 8; 10 12]
```

```
P =
```

```
    [-17]    []    []  
    [3.14] 'Awesome' [3x2 double]
```

```
>> p{1,3} = sum(p{2,3});
```

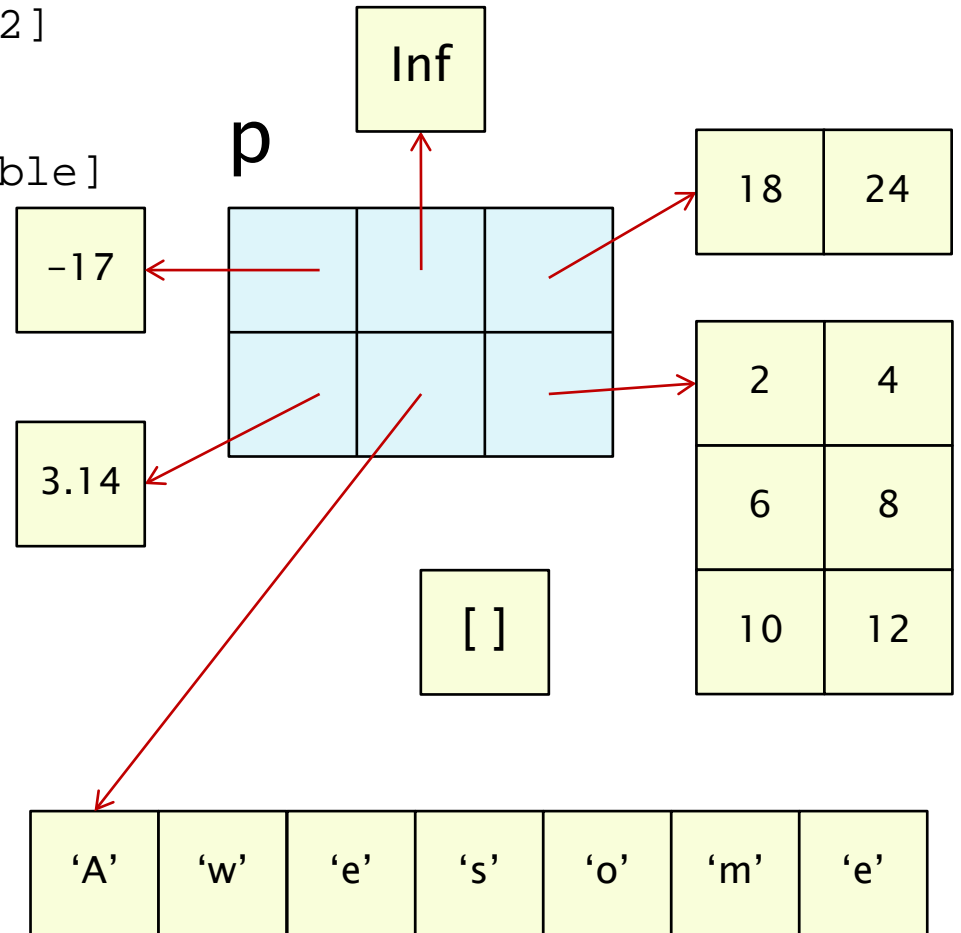
```
>> P{1,2} = 1/0;
```

```
>> class(p)
```

```
ans =
```

```
    cell
```

```
>>
```





VANDERBILT  
UNIVERSITY

# Cell array example

```
>> p{2,3} = [2 4; 6 8; 10 12]
```

```
P =
```

```
    [-17]    []    []  
    [3.14] 'Awesome' [3x2 double]
```

```
>> p{1,3} = sum(p{2,3});
```

```
>> P{1,2} = 1/0;
```

```
>> class(p)
```

```
ans =
```

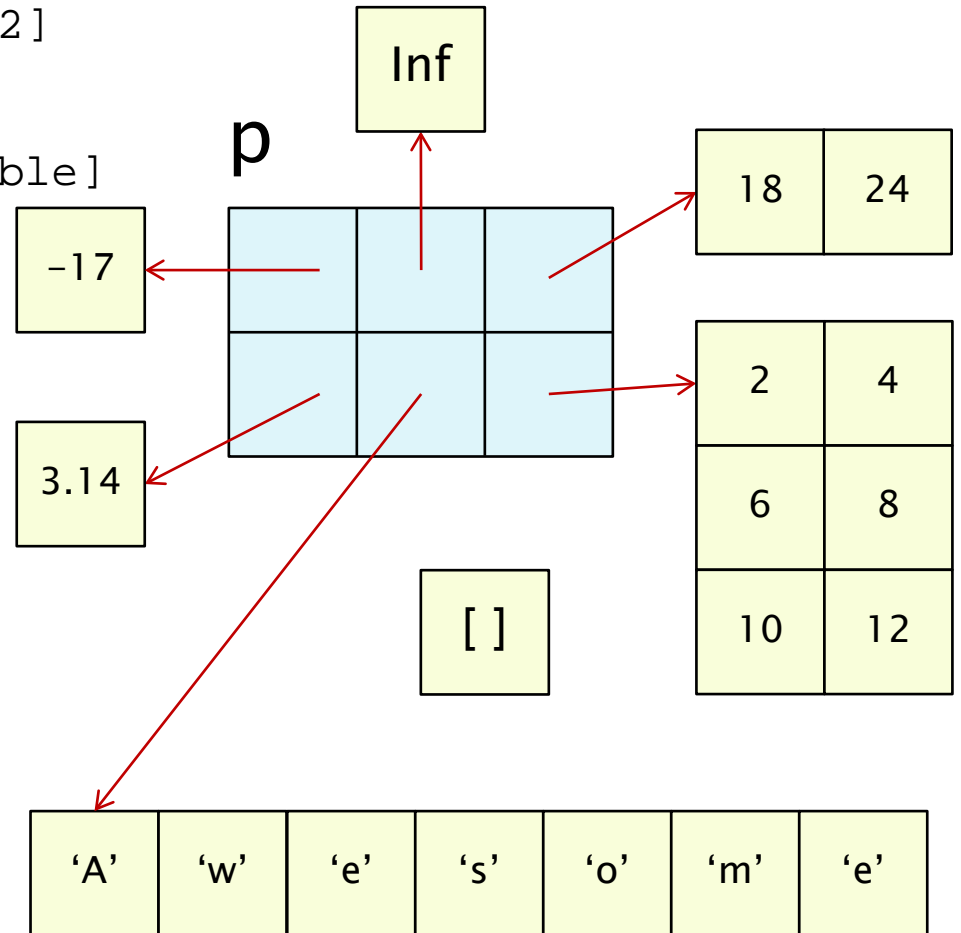
```
    cell
```

```
>> class(p{1,2})
```

```
ans =
```

```
    double
```

```
>>
```





VANDERBILT  
UNIVERSITY

# Cell array example

```
>> p{2,3} = [2 4; 6 8; 10 12]
```

```
P =
```

```
    [-17]    []    []  
    [3.14] 'Awesome' [3x2 double]
```

```
>> p{1,3} = sum(p{2,3});
```

```
>> P{1,2} = 1/0;
```

```
>> class(p)
```

```
ans =
```

```
    cell
```

```
>> class(p{1,2})
```

```
ans =
```

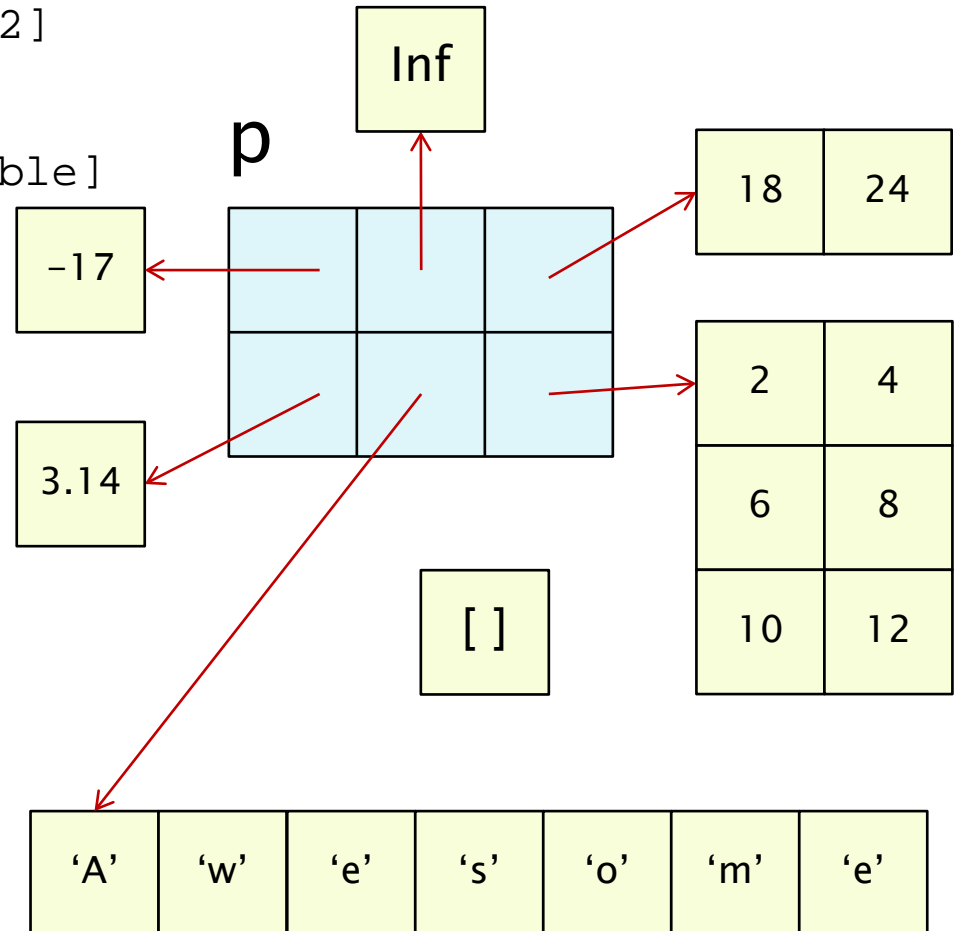
```
    double
```

```
>> class(p(1,2))
```

```
ans =
```

```
    cell
```

```
>>
```



# Cell array example



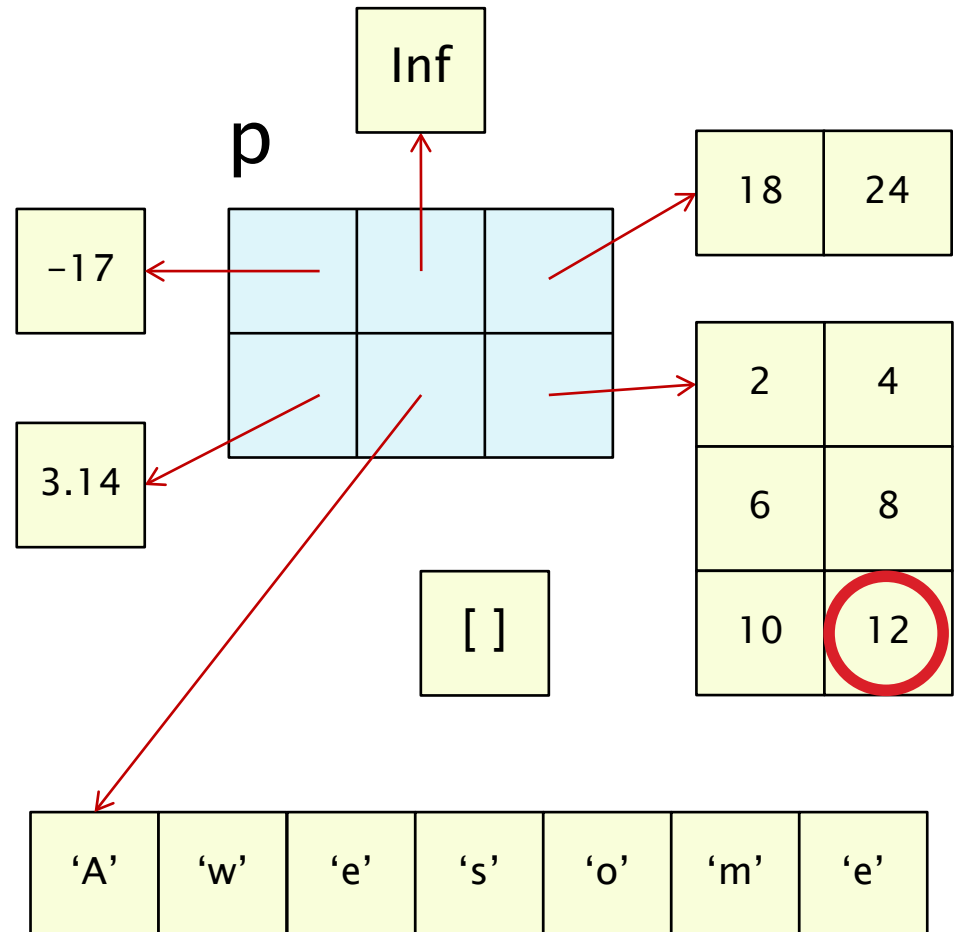
VANDERBILT  
UNIVERSITY

```
>> p{2,3}(3,2)
```

```
ans =
```

```
12
```

```
>>
```



# Cell functions

FUNCTION	DESCRIPTION
cell	create an array of type cell
celldisp	show all the objects pointed at by a cell array
cellfun	apply a function to all the objects pointed at by a cell array
cellplot	show a graphical depiction of the contents of a cell array
cell2struct	convert a cell array into a struct array
deal	copy a value into output arguments
iscell	returns true if argument is of type cell
num2cell	convert a numeric array into a cell array

# Computer Programming with MATLAB



## Lesson 8: File Input/Output

by

Akos Ledeczi and Mike Fitzpatrick

# File Input/Output

## ▶ File:

- Area in permanent storage (disk drive)
- Stores information
- Managed by the operating system
- Can be copied or moved
- Can be accessed by programs

## ▶ File Input/Output (I/O)

- Data exchange between programs and computers
- Data exchange between the physical world and computers
- Saving your work so you can continue with it later

## ▶ MATLAB can handle

- Mat-files and M-files AND text, binary, and Excel files



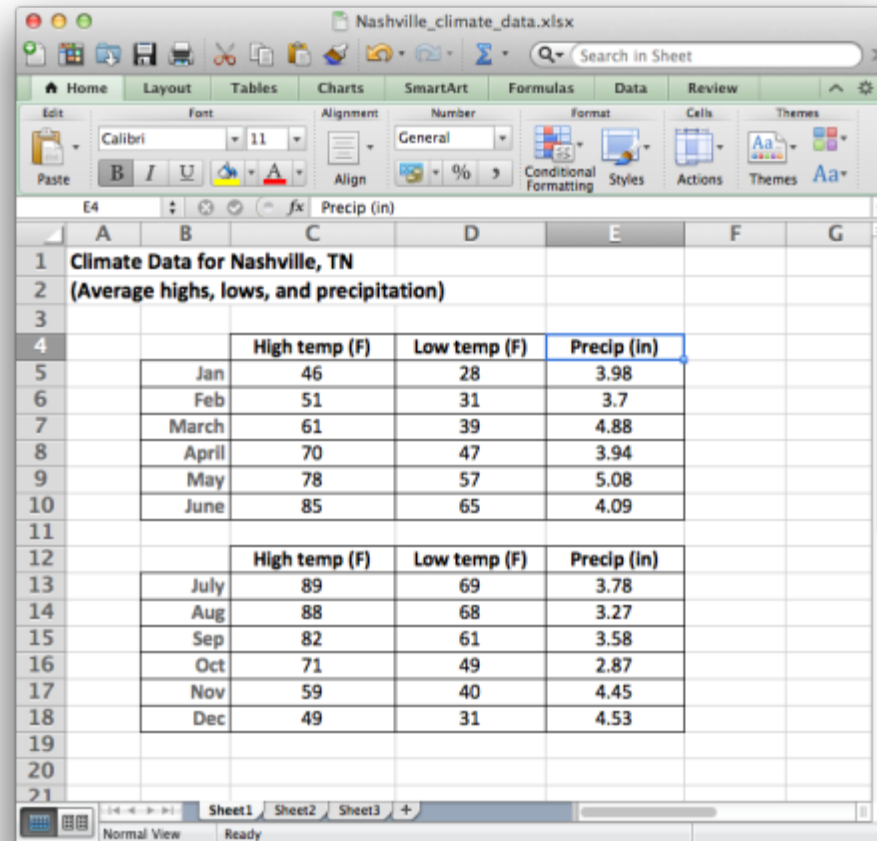
VANDERBILT  
UNIVERSITY

# Excel files

- ▶ Microsoft Excel® is a widely used data-analysis tool
- ▶ Many other programs support reading and writing Excel files
- ▶ MATLAB does too with two built-in functions
  - `xlsread`
  - `xlswrite`



# Reading Excel files



Climate Data for Nashville, TN (Average highs, lows, and precipitation)				
		High temp (F)	Low temp (F)	Precip (in)
Jan		46	28	3.98
Feb		51	31	3.7
March		61	39	4.88
April		70	47	3.94
May		78	57	5.08
June		85	65	4.09
		High temp (F)	Low temp (F)	Precip (in)
July		89	69	3.78
Aug		88	68	3.27
Sep		82	61	3.58
Oct		71	49	2.87
Nov		59	40	4.45
Dec		49	31	4.53

```
>> [num,txt,row] = xlsread('Nashville_climate.xlsx');
```

# Numerical



VANDERBILT  
UNIVERSITY

```
>> num
```

```
num =
```

46	28	3.98
51	31	3.7
61	39	4.88
70	47	3.94
78	57	5.08
85	65	4.09
NaN	NaN	NaN
NaN	NaN	NaN
89	69	3.78
88	68	3.27
82	61	3.58
71	49	2.87
59	40	4.45
49	31	4.53

The screenshot shows an Excel spreadsheet titled "Nashville\_climate\_data.xlsx". The spreadsheet contains climate data for Nashville, TN, including average high and low temperatures and precipitation for each month. The data is organized into two tables, one for the first half of the year (January to June) and one for the second half (July to December). The first table has columns for High temp (F), Low temp (F), and Precip (in). The second table also has columns for High temp (F), Low temp (F), and Precip (in). The data is as follows:

	High temp (F)	Low temp (F)	Precip (in)
Jan	46	28	3.98
Feb	51	31	3.7
March	61	39	4.88
April	70	47	3.94
May	78	57	5.08
June	85	65	4.09
July	89	69	3.78
Aug	88	68	3.27
Sep	82	61	3.58
Oct	71	49	2.87
Nov	59	40	4.45
Dec	49	31	4.53

# Numerical



VANDERBILT  
UNIVERSITY

```
>> num
```

```
num =
```

46	28	3.98
51	31	3.7
61	39	4.88
70	47	3.94
78	57	5.08
85	65	4.09
NaN	NaN	NaN
NaN	NaN	NaN
89	69	3.78
88	68	3.27
82	61	3.58
71	49	2.87
59	40	4.45
49	31	4.53

Nashville\_climate\_data.xlsx

Search in Sheet

Home Layout Tables Charts SmartArt Formulas Data Review

Edit Font Alignment Number Format Cells Themes

Paste B I U A Align General Conditional Formatting Styles Actions Themes Aa

E4 fx Precip (in)

	A	B	C	D	E	F	G
1	Climate Data for Nashville, TN						
2	(Average highs, lows, and precipitation)						
3							
4			High temp (F)	Low temp (F)	Precip (in)		
5		Jan	46	28	3.98		
6		Feb	51	31	3.7		
7		March	61	39	4.88		
8		April	70	47	3.94		
9		May	78	57	5.08		
10		June	85	65	4.09		
11							
12			High temp (F)	Low temp (F)	Precip (in)		
13		July	89	69	3.78		
14		Aug	88	68	3.27		
15		Sept	82	61	3.58		
16		Oct	71	49	2.87		
17		Nov	59	40	4.45		
18		Dec	49	31	4.53		
19							
20							
21							

Sheet1 Sheet2 Sheet3

Normal View Ready

# Numerical



VANDERBILT  
UNIVERSITY

```
>> num
```

```
num =
```

46	28	3.98
51	31	3.7
61	39	4.88
70	47	3.94
78	57	5.08
85	65	4.09
NaN	NaN	NaN
NaN	NaN	NaN
89	69	3.78
88	68	3.27
82	61	3.58
71	49	2.87
59	40	4.45
49	31	4.53

Nashville\_climate\_data.xlsx

Search in Sheet

Home Layout Tables Charts SmartArt Formulas Data Review

Climate Data for Nashville, TN  
(Average highs, lows, and precipitation)

	High temp (F)	Low temp (F)	Precip (in)
Jan	46	28	3.98
Feb	51	31	3.7
Mar	61	39	4.88
Apr	70	47	3.94
May	78	57	5.08
Jun	85	65	4.09
Jul	89	69	3.78
Aug	88	68	3.27
Sep	82	61	3.58
Oct	71	49	2.87
Nov	59	40	4.45
Dec	49	31	4.53

# Text



VANDERBILT  
UNIVERSITY

```
>> txt =
```

```
txt =
```

```
[1x30 char] ''      ''      ''      ''
[1x40 char] ''      ''      ''      ''
''          ''      ''      ''      ''
''          ''      'High temp (F)' 'Low temp (F)' 'Precip (in)'
```

''	'Jan'	''	''	''
''	'Feb'	''	''	''
''	'March'	''	''	''
''	'April'	''	''	''
''	'May'	''	''	''
''	'June'	''	''	''
''	''	''	''	''
''	''	'High temp (F)'	'Low temp (F)'	'Precip (in)'
''	'July'	''	''	''
''	'Aug'	''	''	''
''	'Sep'	''	''	''
''	'Oct'	''	''	''
''	'Nov'	''	''	''
''	'Dec'	''	''	''

# All data: cell array



VANDERBILT  
UNIVERSITY

```
>> raw
```

```
raw =
```

```
[1x30 char] [ NaN] [ NaN]          [ NaN]          [ NaN]
[1x40 char] [ NaN] [ NaN]          [ NaN]          [ NaN]
[ NaN]      [ NaN] [ NaN]          [ NaN]          [ NaN]
[ NaN]      [ NaN] 'High temp (F)' 'Low temp (F)' 'Precip (in)'
```

[ NaN]	'Jan'	[ 46]	[ 28]	[ 3.98]
[ NaN]	'Feb'	[ 51]	[ 31]	[ 3.7]
[ NaN]	'March'	[ 61]	[ 39]	[ 4.88]
[ NaN]	'April'	[ 70]	[ 47]	[ 3.94]
[ NaN]	'May'	[ 78]	[ 57]	[ 5.08]
[ NaN]	'June'	[ 85]	[ 65]	[ 4.09]
[ NaN]	[ NaN]	[ NaN]	[ NaN]	[ NaN]
[ NaN]	[ NaN]	'High temp (F)'	'Low temp (F)'	'Precip (in)'
[ NaN]	'July'	[ 89]	[ 69]	[ 3.78]
[ NaN]	'Aug'	[ 88]	[ 68]	[ 3.27]
[ NaN]	'Sep'	[ 82]	[ 61]	[ 3.58]
[ NaN]	'Oct'	[ 71]	[ 49]	[ 2.87]
[ NaN]	'Nov'	[ 59]	[ 40]	[ 4.45]
[ NaN]	'Dec'	[ 49]	[ 31]	[ 4.53]

# Text files

- ▶ Text files contain characters
- ▶ They use an encoding scheme:
  - ASCII or
  - Any one of many other schemes
  - MATLAB takes care of encoding and decoding
- ▶ Before using a text file, we need to open it
- ▶ Once done with the file, we need to close it

# Opening text files

- ▶ Opening: `fid = fopen(filename, permission)`
- ▶ Closing: `fclose(fid)`
- ▶ `fid`: Unique file identifier for accessing file
- ▶ Permission: what we want to do with the file—
  - read, write, overwrite, append, etc.

2ND ARGUMENT	PERMISSION
'rt'	open text file for reading
'wt'	open text file for writing; discard existing contents
'at'	open or create text file for writing; append data to end of file
'r+t'	open (do not create) text file for reading and writing
'w+t'	open or create text file for reading and writing; discard existing contents
'a+t'	open or create text file for reading and writing; append data to end of file



# Reading text files

- ▶ One line at a time
- ▶ `type` prints a text file in the command window
- ▶ Let's re-implement it:

```
function view_text_file(filename)
fid = fopen(filename,'rt');
if fid < 0
    error('error opening file %s\n\n', filename);
end

% Read file as a set of strings, one string per line:
oneline = fgets(fid);
while ischar(oneline)
    fprintf('%s',oneline) % display one line
    oneline = fgets(fid);
end
fprintf('\n');
fclose(fid);
```



VANDERBILT  
UNIVERSITY

# Reading text files

- ▶ Reading lines into string variables is easy
- ▶ Parsing these strings to get numerical data is much harder
- ▶ Not covered
- ▶ Binary files are more suited for numerical data

# Binary files

- ▶ Binary file = “not a text file”
- ▶ Many different ways to represent numbers
- ▶ All we need to know are their types.
- ▶ Binary files need to be
  - Opened with **fopen**
  - Closed with **fclose**

2ND ARGUMENT	PERMISSION
'r'	open binary file for reading
'w'	open binary file for writing; discard existing contents
'a'	open or create binary file for writing; append data to end of file
'r+'	open (do not create) binary file for reading and writing
'w+'	open or create binary file for reading and writing; discard existing contents
'a+'	open or create binary file for reading and writing; append data to end of file

# Writing binary files

- ▶ Data type is important
- ▶ Example: write a double array into a binary file

```
function write_array_bin(A,filename)
fid = fopen(filename,'w+');
if fid < 0
    error('error opening file %s\n', filename);
end

fwrite(fid,A,'double');

fclose(fid);
```

# Reading binary files

- ▶ Example: read a double array from a binary file

```
function A = read_bin_file(filename,data_type)
fid = fopen(filename,'r');
if fid < 0
    error('error opening file %s\n',filename);
end

A = fread(fid,inf,data_type);

fclose(fid);
```