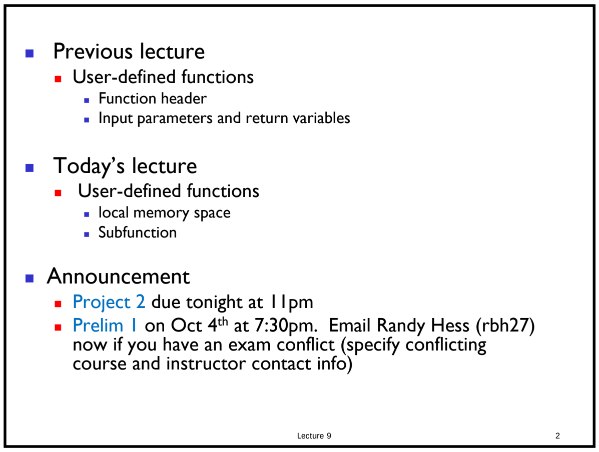
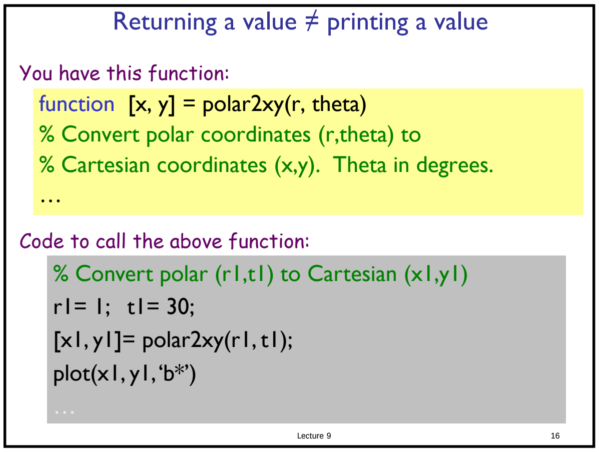


- Previous lecture
  - User-defined functions
    - Function header
    - Input parameters and return variables
- Today's lecture
  - User-defined functions
    - local memory space
    - Subfunction
- Announcement
  - Project 2 due tonight at 11pm
  - Prelim 1 on Oct 4th at 7:30pm. Email Randy Hess (rbh27) now if you have an exam conflict (specify conflicting course and instructor contact info)



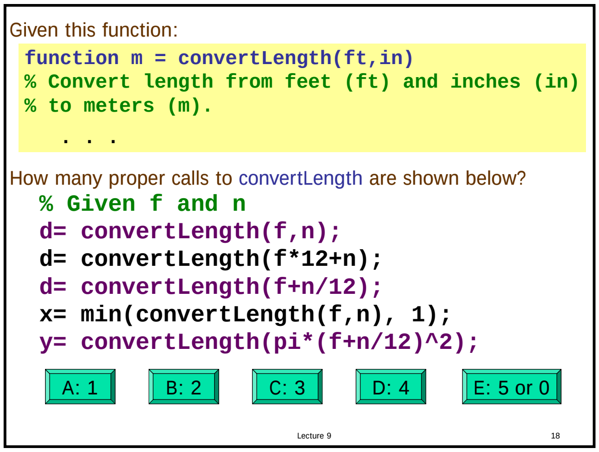
Returning a value $\neq$ printing a value

You have this function:

```
function  [x, y] = polar2xy(r, theta)
% Convert polar coordinates (r,theta) to
% Cartesian coordinates (x,y).  Theta in degrees.
…
```

Code to call the above function:

```
% Convert polar (r1,t1) to Cartesian (x1,y1)
r1= 1;  t1= 30;
[x1,y1]= polar2xy(r1,t1);
plot(x1,y1,'b*')
…
```



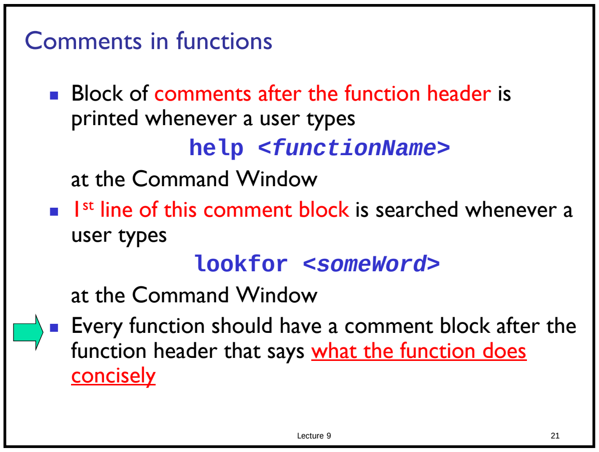
Given this function:

```
function m = convertLength(ft,in)
% Convert length from feet (ft) and inches (in)
% to meters (m).
   . . .
```

How many proper calls to convertLength are shown below?

```
% Given f and n
d= convertLength(f,n);
d= convertLength(f*12+n);
d= convertLength(f+n/12);
x= min(convertLength(f,n), 1);
y= convertLength(pi*(f+n/12)^2);
```

A: 1 B: 2 C: 3 D: 4 E: 5 or 0



Comments in functions

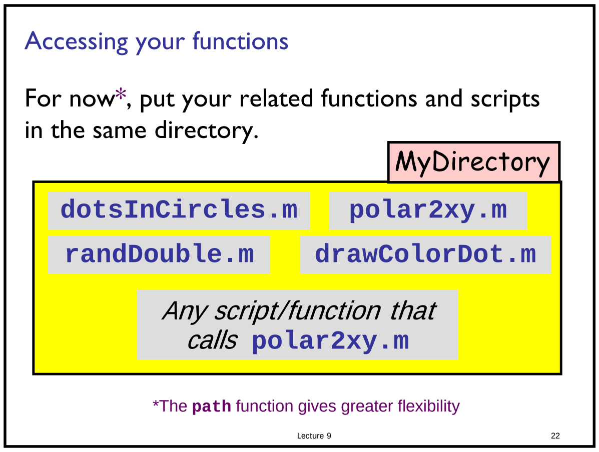
- Block of comments after the function header is printed whenever a user types

  `help <functionName>`

  at the Command Window
- 1st line of this comment block is searched whenever a user types

  `lookfor <someWord>`

  at the Command Window
- Every function should have a comment block after the function header that says what the function does concisely



Accessing your functions

For now*, put your related functions and scripts in the same directory.

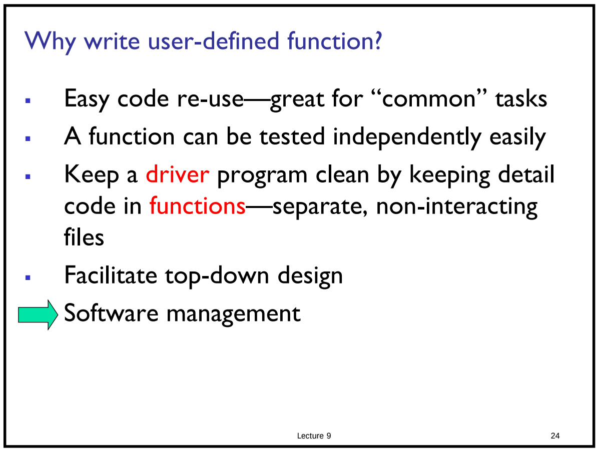
MyDirectory

dotsInCircles.m polar2xy.m

randDouble.m drawColorDot.m

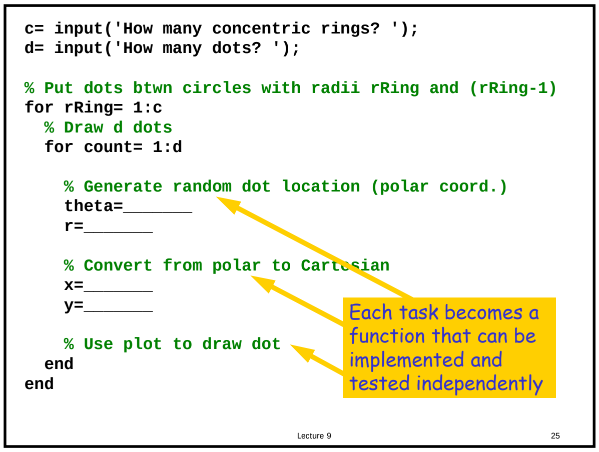
Any script/function that calls polar2xy.m

*The path function gives greater flexibility



Why write user-defined function?

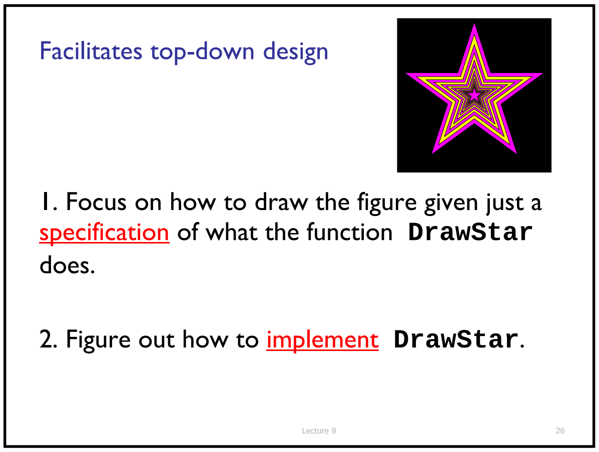
- Easy code re-use—great for "common" tasks
- A function can be tested independently easily
- Keep a driver program clean by keeping detail code in functions—separate, non-interacting files
- Facilitate top-down design
- Software management



```
c= input('How many concentric rings? ');
d= input('How many dots? ');

% Put dots btwn circles with radii rRing and (rRing-1)
for rRing= 1:c
  % Draw d dots
  for count= 1:d

    % Generate random dot location (polar coord.)
    theta=_______
    r=_______

    % Convert from polar to Cartesian
    x=_______
    y=_______

    % Use plot to draw dot
  end
end
```

Each task becomes a function that can be implemented and tested independently

Lecture 9 25

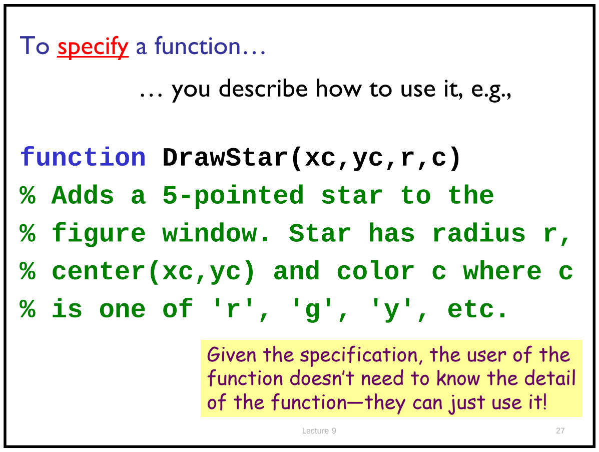


Facilitates top-down design

1. Focus on how to draw the figure given just a specification of what the function **DrawStar** does.

2. Figure out how to implement **DrawStar**.

Lecture 9 26



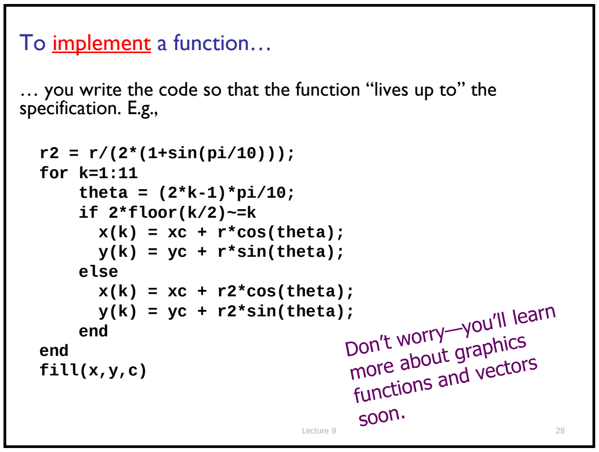
To specify a function…

… you describe how to use it, e.g.,

```
function DrawStar(xc,yc,r,c)
% Adds a 5-pointed star to the
% figure window. Star has radius r,
% center(xc,yc) and color c where c
% is one of 'r', 'g', 'y', etc.
```

Given the specification, the user of the function doesn't need to know the detail of the function—they can just use it!

Lecture 9 27



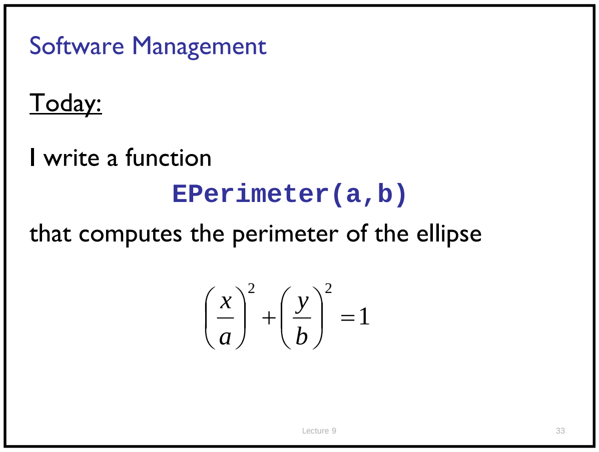
To implement a function…

… you write the code so that the function "lives up to" the specification. E.g.,

```
r2 = r/(2*(1+sin(pi/10)));
for k=1:11
    theta = (2*k-1)*pi/10;
    if 2*floor(k/2)~=k
      x(k) = xc + r*cos(theta);
      y(k) = yc + r*sin(theta);
    else
      x(k) = xc + r2*cos(theta);
      y(k) = yc + r2*sin(theta);
    end
end
fill(x,y,c)
```

Don't worry—you'll learn more about graphics functions and vectors soon.

Lecture 9 28



Software Management

Today:

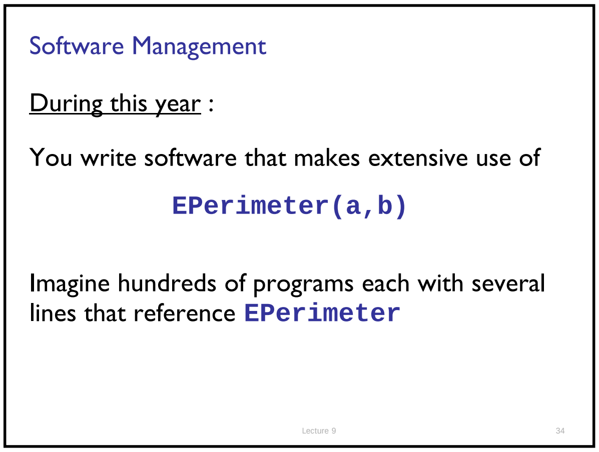
I write a function

EPerimeter(a,b)

that computes the perimeter of the ellipse

$$\left(\frac{x}{a}\right)^2+\left(\frac{y}{b}\right)^2=1$$

Lecture 9 33



Software Management

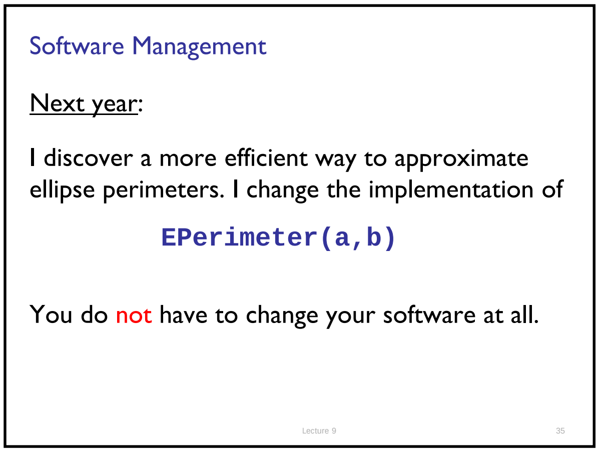
During this year :

You write software that makes extensive use of

EPerimeter(a,b)

Imagine hundreds of programs each with several lines that reference **EPerimeter**

Lecture 9 34



Software Management

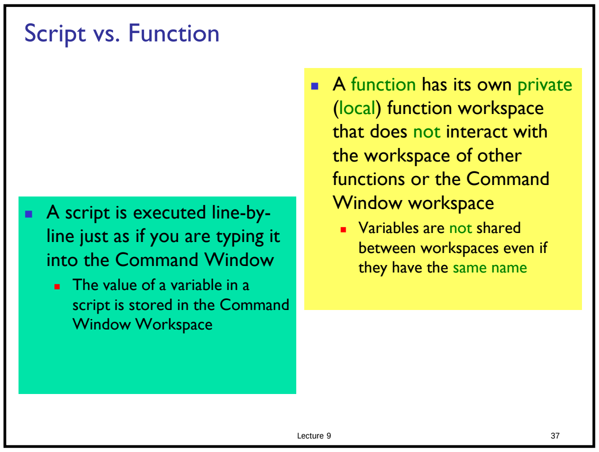
Next year:

I discover a more efficient way to approximate ellipse perimeters. I change the implementation of

```
EPerimeter(a,b)
```

You do not have to change your software at all.

Lecture 9 35

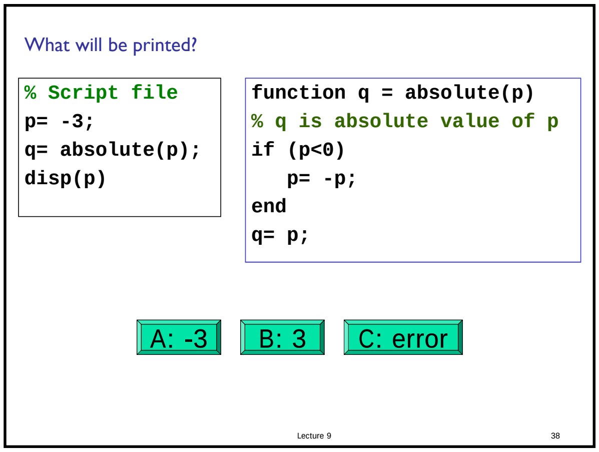


Script vs. Function

- A script is executed line-by-line just as if you are typing it into the Command Window
  - The value of a variable in a script is stored in the Command Window Workspace

- A function has its own private (local) function workspace that does not interact with the workspace of other functions or the Command Window workspace
  - Variables are not shared between workspaces even if they have the same name

Lecture 9 37



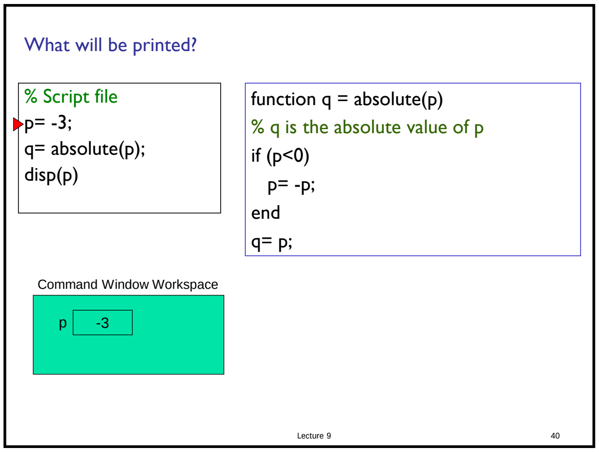
What will be printed?

```
% Script file
p= -3;
q= absolute(p);
disp(p)
```

```
function q = absolute(p)
% q is absolute value of p
if (p<0)
   p= -p;
end
q= p;
```

A: -3 B: 3 C: error

Lecture 9 38

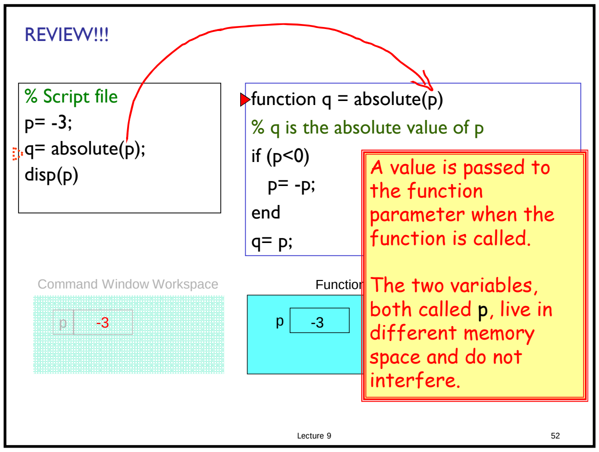


What will be printed?

```
% Script file
p= -3;
q= absolute(p);
disp(p)
```

```
function q = absolute(p)
% q is the absolute value of p
if (p<0)
   p= -p;
end
q= p;
```

Lecture 9 40



REVIEW!!!

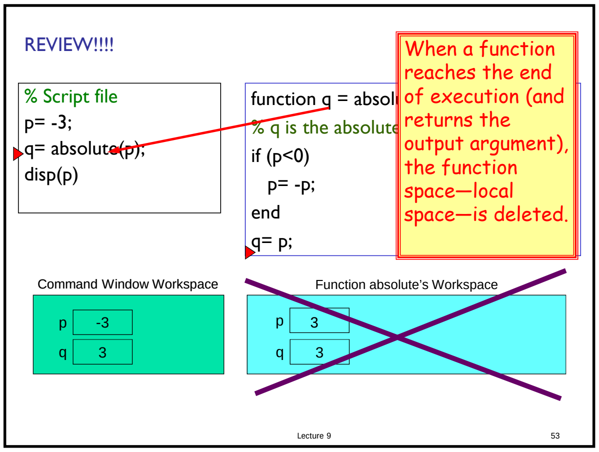
```
% Script file
p= -3;
q= absolute(p);
disp(p)
```

```
function q = absolute(p)
% q is the absolute value of p
if (p<0)
   p= -p;
end
q= p;
```

A value is passed to the function parameter when the function is called.

The two variables, both called p, live in different memory space and do not interfere.

Lecture 9 52



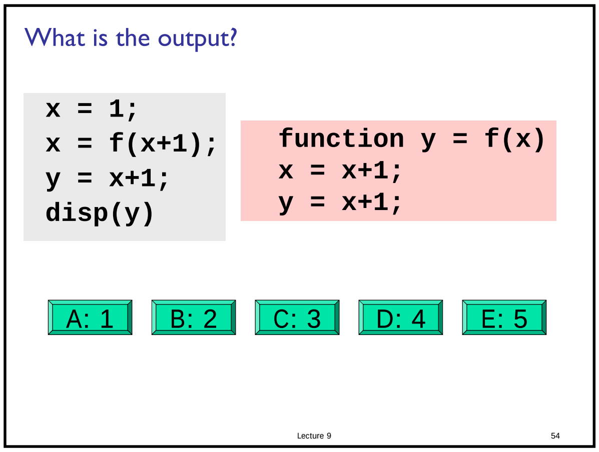
REVIEW!!!!

```
% Script file
p= -3;
q= absolute(p);
disp(p)
```

```
function q = absol
% q is the absolute
if (p<0)
   p= -p;
end
q= p;
```

When a function reaches the end of execution (and returns the output argument), the function space—local space—is deleted.

Lecture 9 53

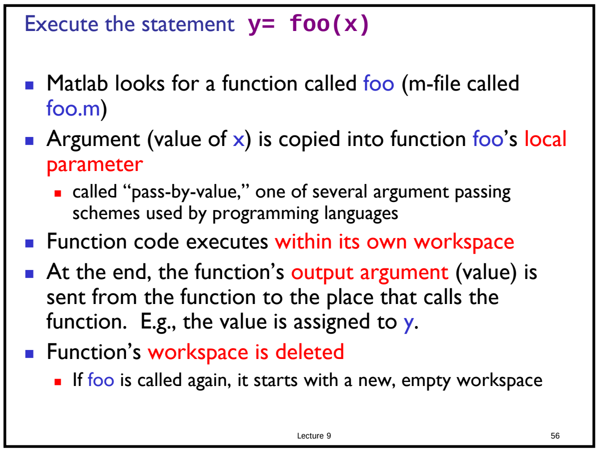


What is the output?

```
x = 1;
x = f(x+1);
y = x+1;
disp(y)
```

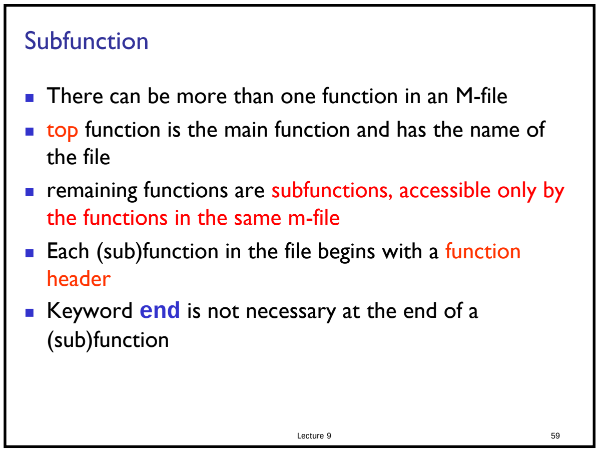
```
function y = f(x)
x = x+1;
y = x+1;
```

A: 1 B: 2 C: 3 D: 4 E: 5



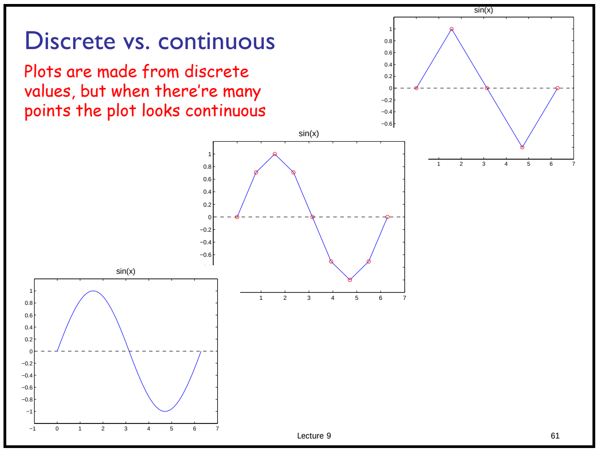
Execute the statement `y= foo(x)`

- Matlab looks for a function called foo (m-file called foo.m)
- Argument (value of x) is copied into function foo's local parameter
  - called "pass-by-value," one of several argument passing schemes used by programming languages
- Function code executes within its own workspace
- At the end, the function's output argument (value) is sent from the function to the place that calls the function. E.g., the value is assigned to y.
- Function's workspace is deleted
  - If foo is called again, it starts with a new, empty workspace



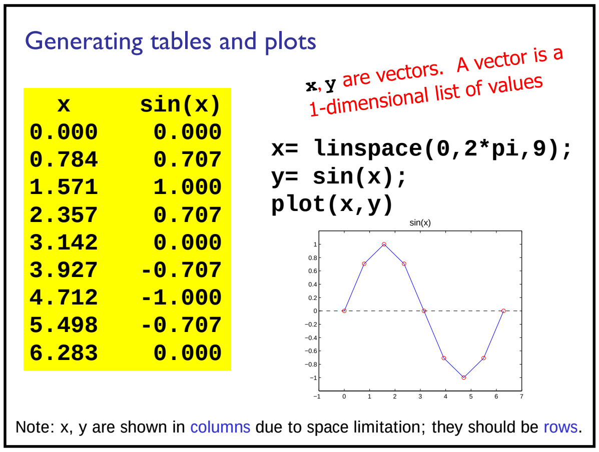
Subfunction

- There can be more than one function in an M-file
- top function is the main function and has the name of the file
- remaining functions are subfunctions, accessible only by the functions in the same m-file
- Each (sub)function in the file begins with a function header
- Keyword **end** is not necessary at the end of a (sub)function



Discrete vs. continuous

Plots are made from discrete values, but when there're many points the plot looks continuous



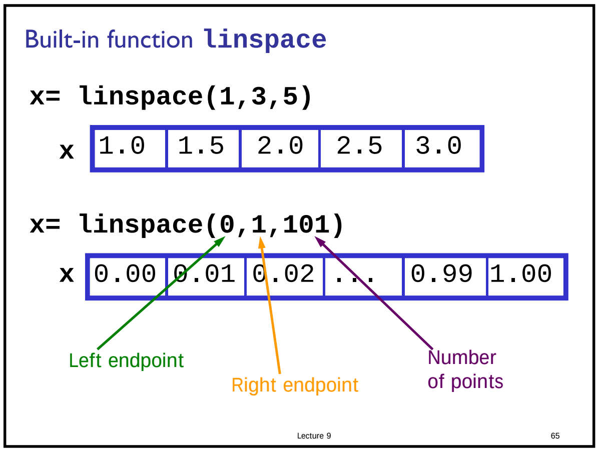
Generating tables and plots

x, **y** are vectors. A vector is a 1-dimensional list of values

x	sin(x)
0.000	0.000
0.784	0.707
1.571	1.000
2.357	0.707
3.142	0.000
3.927	-0.707
4.712	-1.000
5.498	-0.707
6.283	0.000

```
x= linspace(0,2*pi,9);
y= sin(x);
plot(x,y)
```

Note: x, y are shown in columns due to space limitation; they should be rows.



Built-in function `linspace`

```
x= linspace(1,3,5)
```

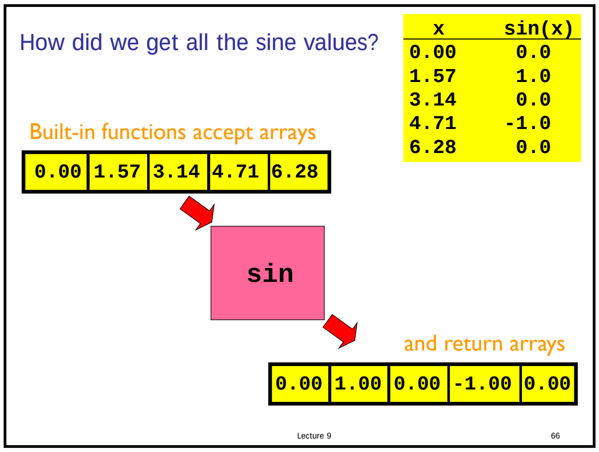
x

1.0	1.5	2.0	2.5	3.0

```
x= linspace(0,1,101)
```

x

0.00	0.01	0.02	. . .	0.99	1.00



How did we get all the sine values?

x	sin(x)
0.00	0.0
1.57	1.0
3.14	0.0
4.71	-1.0
6.28	0.0

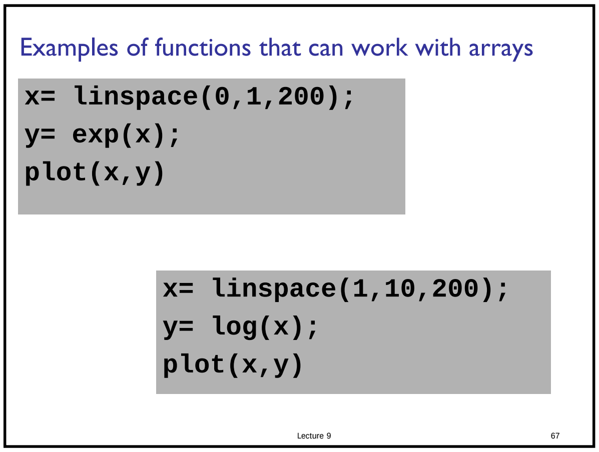
Built-in functions accept arrays

0.00	1.57	3.14	4.71	6.28

sin

and return arrays

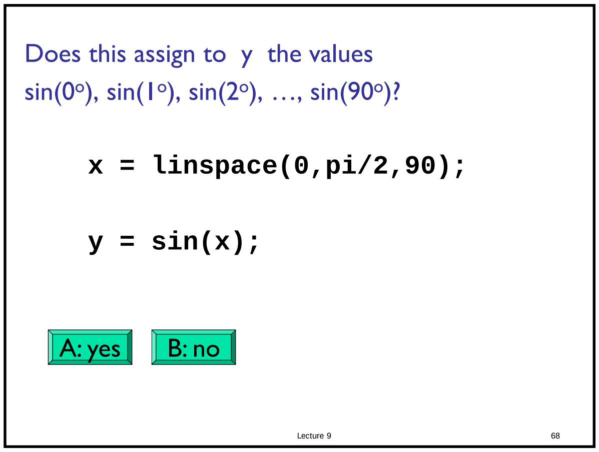
0.00	1.00	0.00	-1.00	0.00



Examples of functions that can work with arrays

```
x= linspace(0,1,200);
y= exp(x);
plot(x,y)
```

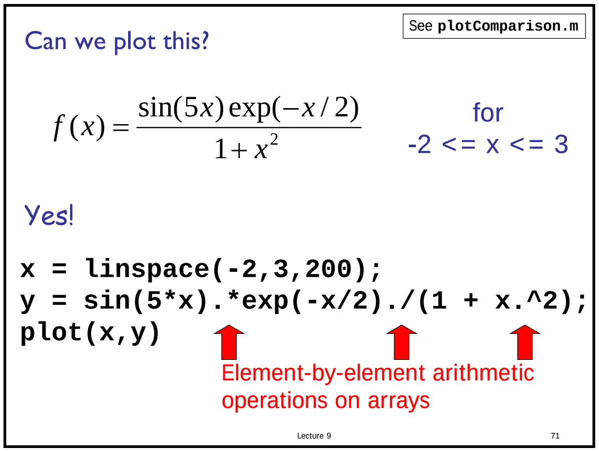
```
x= linspace(1,10,200);
y= log(x);
plot(x,y)
```



Does this assign to y the values sin(0°), sin(1°), sin(2°), …, sin(90°)?

```
x = linspace(0,pi/2,90);

y = sin(x);
```

A: yes B: no



See `plotComparison.m`

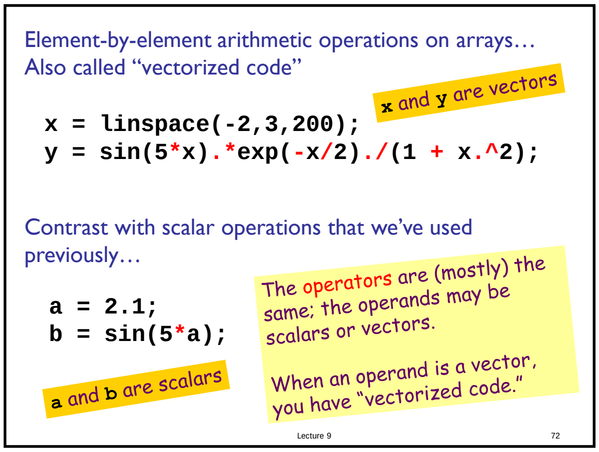
Can we plot this?

$$f(x) = \frac{\sin(5x)\exp(-x/2)}{1+x^2} \quad \text{for } -2 <= x <= 3$$

Yes!

```
x = linspace(-2,3,200);
y = sin(5*x).*exp(-x/2)./(1 + x.^2);
plot(x,y)
```

Element-by-element arithmetic operations on arrays



Element-by-element arithmetic operations on arrays…
Also called “vectorized code”

```
x = linspace(-2,3,200);
y = sin(5*x).*exp(-x/2)./(1 + x.^2);
```

x and y are vectors

Contrast with scalar operations that we’ve used previously…

```
a = 2.1;
b = sin(5*a);
```

a and b are scalars

The operators are (mostly) the same; the operands may be scalars or vectors.

When an operand is a vector, you have “vectorized code.”