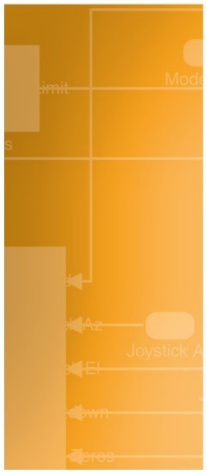
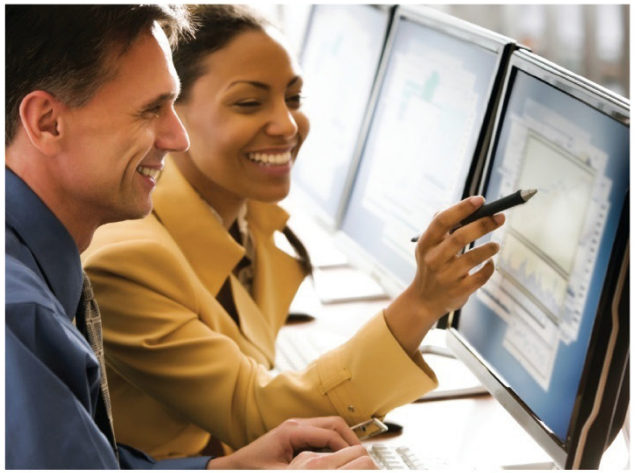
MathWorks | Training Services

Increasing Automation with Functions

MATLAB® Fundamentals for Aerospace Applications

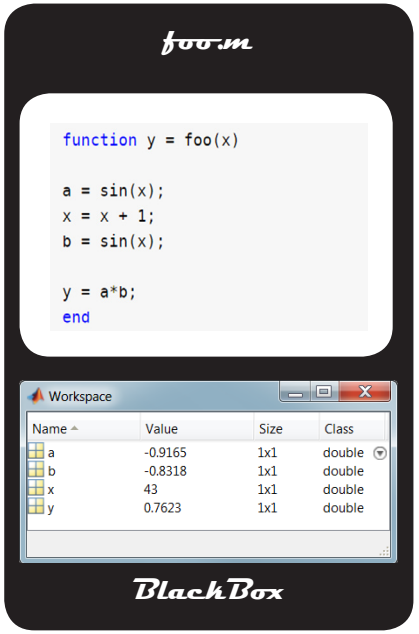


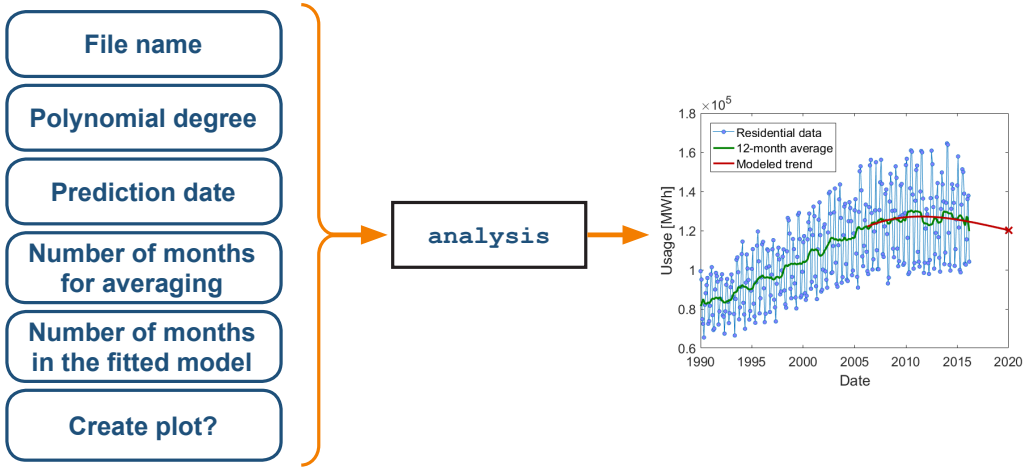


MathWorks | Training Services

Outline

- Creating and calling functions
- Workspaces
- Plain text code files
- Path and precedence
- Debugging
- Using structures





Chapter Learning Outcomes

The attendee will be able to:

- Create and call a function.
- Set the MATLAB® path to ensure a code file is visible.
- Determine which file or variable is being accessed when a MATLAB command is issued.
- Use diagnostic tools to find and correct problems in code files.
- Store and manipulate data in structures.

Course Example: Electricity Modeling

The script `analysis_script.mlx` implements a complete data analysis workflow on the electricity usage data:

- Import data from file.
- Preprocess the data (remove `NaN`s).
- Fit a polynomial to selected data.
- Use the fitted model to predict the usage at a future date.
- Make a plot of the data and the fitted model.

The first few lines of the script define some important variables:

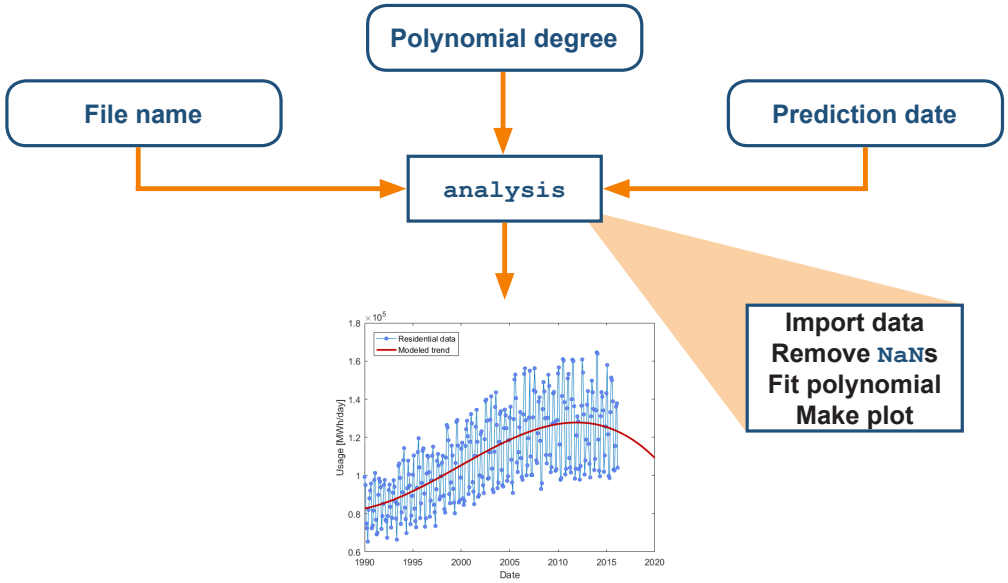
- The name of the data file
- The degree of the polynomial to fit to the data
- The future date at which to predict the usage

Try

Open and run a script that analyzes residential electricity usage data: `analysis_script.mlx`.

These are typically the kinds of parameters that you would change before re-running the script.

The goal of the example in this chapter is to convert this script into a function so that the analysis can be run with different parameters without changing the code.

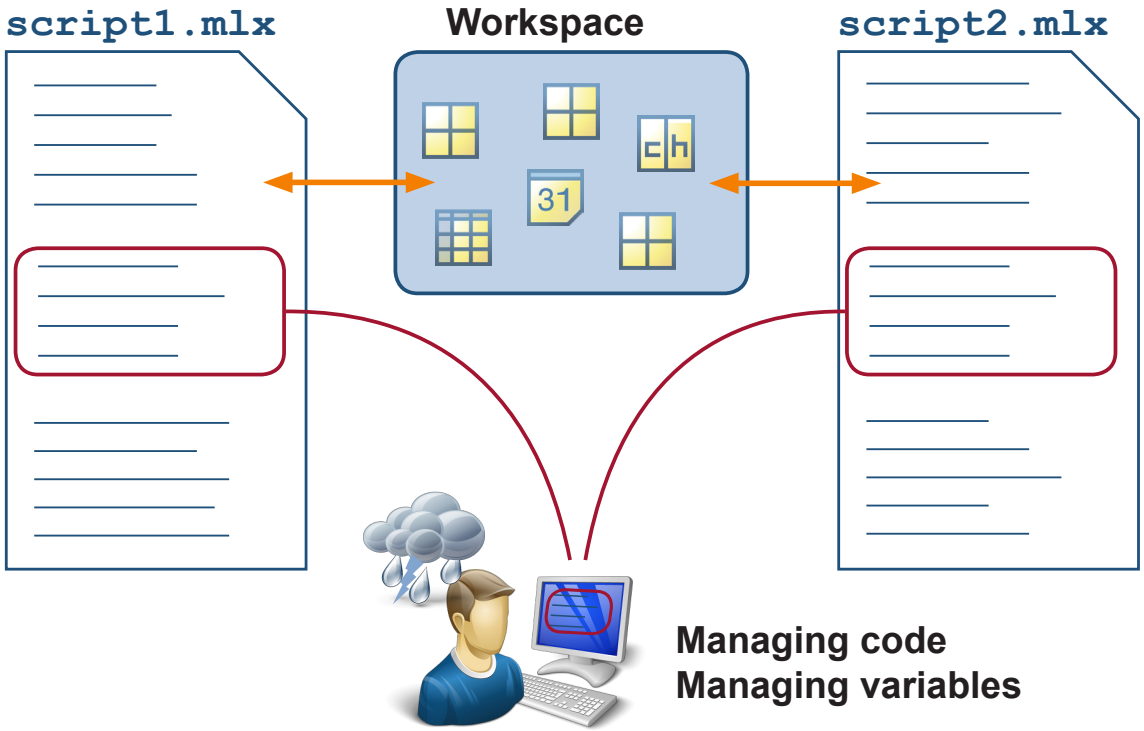


Why Use Functions?

As the complexity of your scripts increases, you will most likely find that certain pieces of code are repeated in several different places. In this case, it makes sense to package this segment of code as a separate, general "helper" code that you can call with a single command. In this way you reduce maintenance effort: if you change this block of code, you now need change it in one place only.

It is possible to call scripts from within other scripts. However, this is generally not recommended as all scripts share the MATLAB (base) workspace. Hence, any change made to a variable by one script affects all the other scripts that reference that variable. Furthermore, you need to keep a careful inventory of your workspace so that all scripts are using the same variable names. This undermines the purpose of keeping your code modular so as to be more maintainable.

A better solution is to create your own functions. A function can be seen as a "black box": code that requires certain types of inputs, performs some action, and returns some outputs. As it is running, you typically never see the actions "inside the box," just the output. The variables in a function occupy their own, separate workspace.



Creating a Function

Any script can contain code to define a function. Functions are created using a function declaration.

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

- The declaration starts with the `function` keyword.
- The function declaration syntax (after the `function` keyword) is identical to the syntax for calling functions in MATLAB.
- For functions with a single output, the square brackets are unnecessary (and slightly less efficient).
- Values are assigned to input variables when the function is called with specific inputs.
- Code following the function declaration describes how output variables are computed from input variables.
- Each declared output must be assigned a value somewhere in the code.
- The function ends with an `end` statement.

Functions defined within a script are called *local functions*. Local function definitions must be located after all other script code. In the Live Editor, a section break is automatically created before the first local function definition.

Try

In `analysis_script.mlx`, create a local function to predict future electricity usage. The function should accept the name of the data file (`char`), a polynomial degree (scalar value), and a prediction date (`datetime`) as inputs, and return the usage predicted by the model as an output. Name the function `polyPrediction`.

Solution: `analysis_locfunc.mlx`.

There are times when you may want to create a function that has no input or no output (or even both). To accommodate such situations, the following are all valid declaration syntaxes:

```
function [out1,out2,...] = function_name()
function [out1,out2,...] = function_name

function function_name(in1,in2,...)
function [] = function_name(in1,in2,...)

function function_name
function function_name()
function [] = function_name
function [] = function_name()
```

Calling a Function

User-defined functions are called just like any other MATLAB functions. The calling syntax is specified in the function declaration.

The values of the inputs given when the function is called are assigned, in order, to the variables in the function declaration line:

```
[x,y] = myfunction(42,pi,0);
```

```
function [frogs,bats] = myfunction(a,b,c)
frogs = a + b;
bats = b + c;
end
```

Inside `myfunction`, `a` will have the value 42, `b` will have the value `pi`, and `c` will have the value 0.

Similarly, the outputs are assigned in order. In the above example, the variable `frogs` inside the function will have the value 45.1416 and `bats` will have the value `pi`. Hence, in the base MATLAB workspace, `x` will be assigned the value 45.1416 and `y` will be assigned the value `pi`.

If a function is written to return multiple outputs, it is not necessary to return all of the outputs. For example, if only the output `x` is needed, you can call `myfunction` by typing

```
x = myfunction(42,pi,0);
```

Now `x` will be assigned the value 45.1416. The value assigned to `bats` inside the function (`pi`) will be ignored.

Try

Edit your script to call the `polyPrediction` function.

```
enddate = datetime(2020,1,1);
res2020 = polyPrediction(...
     'elec_res.xlsx',3,enddate)
```

Call it again with different parameters.

```
enddate = datetime(2018,9,1);
com2018 = polyPrediction(...
     'elec_com.xlsx',2,enddate)
```

If only `y` is needed, however, you can tell MATLAB to ignore `x` by using the tilde (~) as a placeholder:

```
[~,y] = myfunction(42,pi,0);
```

Now `y` will be assigned the value `pi` and the value assigned to `frogs` inside the function will be ignored

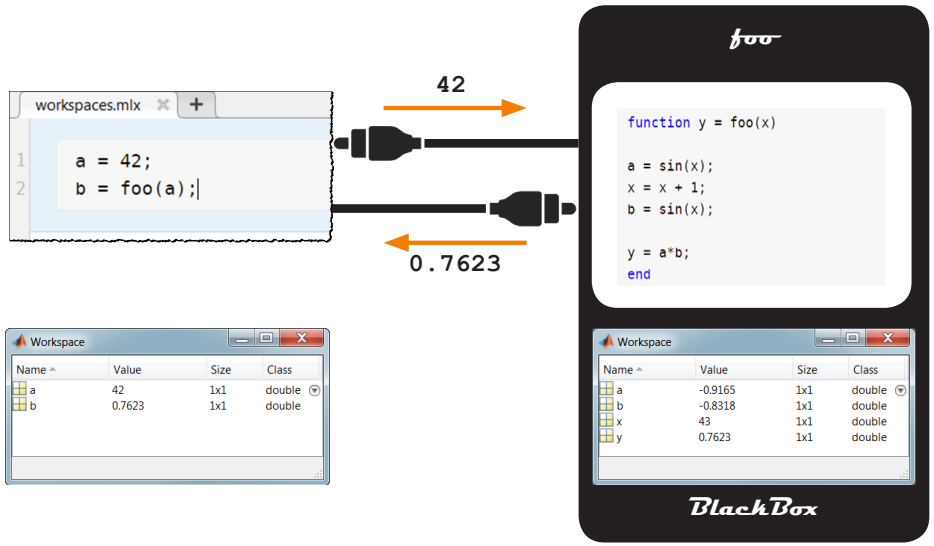
As always, if no return variable is specified when the function is called, the result is returned to the default variable `ans`. If the function returns more than one output, the first will be assigned to `ans` (and the others ignored).

Workspaces

Functions operate within their own *function workspace*, separate from the *base workspace* accessed at the prompt or from within scripts. If a function calls another function, each maintains its own separate workspace.

The separate workspaces of functions are convenient, but also a possible source of error.

The convenience comes with variable naming. You could, for example, have all of your functions accept input `x` and return output `y`. There would never be any confusion because the values of the variables are *local* to the individual function workspaces. Similarly, you may create variables within a function safe in the knowledge that these variables are unique to that one function. There is no need to worry about variable name conflicts among your functions.



Try

Open `workspaces.mlx`. Verify that function variables are local to the function workspace.

A common source of error in function programming is to refer to a variable in one function workspace (or the base workspace) from another function workspace. Because the workspaces are separated in memory, their variables are hidden from one another. MATLAB will tell you that the variable is not found.

Because workspaces are kept separate, MATLAB passes variables by value. That is, when you make a call such as

```
a = foo(b)
```

to a function with a declaration line of

```
function y = foo(x)
```

MATLAB passes the *value* of `b` into `foo`. The variable `x` in the `foo` workspace thus takes this value, but is a separate variable to `b` (which resides in the base workspace). Hence, changes to `x` do not affect `b`. Similarly, once `foo` is done running, it copies the *value* currently in its variable `y` back to the workspace from which it was called, in this case into the base workspace variable `a`. Once the function is finished, its workspace is destroyed.

The MATLAB® Editor

While the Live Editor is an environment designed for exploratory programming and creating interactive documents, the MATLAB Editor is an environment with support for more advanced programming activities, like functions, class definitions, or application development.

The Editor is used to write, edit, run, debug, and publish *MATLAB code files*, which are plain text files containing MATLAB commands. MATLAB code files always have a `.m` extension, so MATLAB knows they contain executable commands.

You can open the Editor and begin a new code file by clicking the **New Script** button in the **File** section of the **Home** tab on the toolstrip.

To create a plain code file from a live script, open the live script and select **Save → Save As...** from the **File** section of the **Live Editor** tab. In the dialog box, select "MATLAB Code files (*.m)" as the **Save as type**.

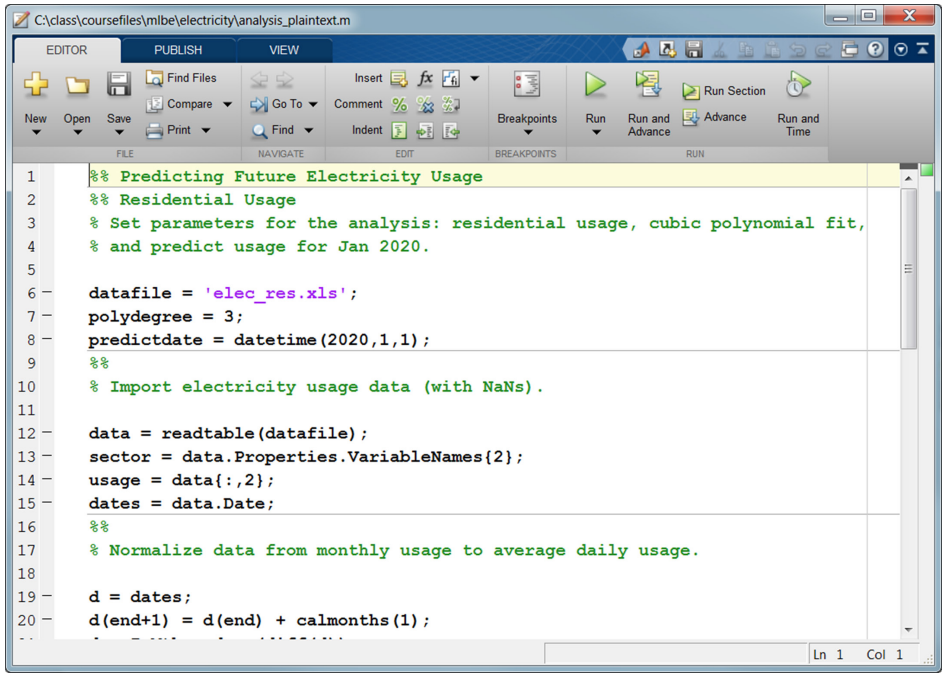
Try

Save the live script `analysis_script.mlx` as the MATLAB code file `analysis_plaintext.m`. Open the saved file in the MATLAB Editor.

By default, the Editor regularly creates *autosave* (`.asv`) backups of files open in the Editor. These are deleted when you close your file. To adjust file backup options, open the preferences and select the **Editor/Debugger → Backup Files** node of the Preferences dialog.

Plain Text Code File

(.m extension)



Creating a Function File

Function files are created in MATLAB code files with a `.m` extension. While local functions can only be accessed by the script in which they live, functions defined in function files can be called from the command line, live scripts, and other MATLAB code files.

Function files have one key syntactic element: function files always begin with the function declaration.

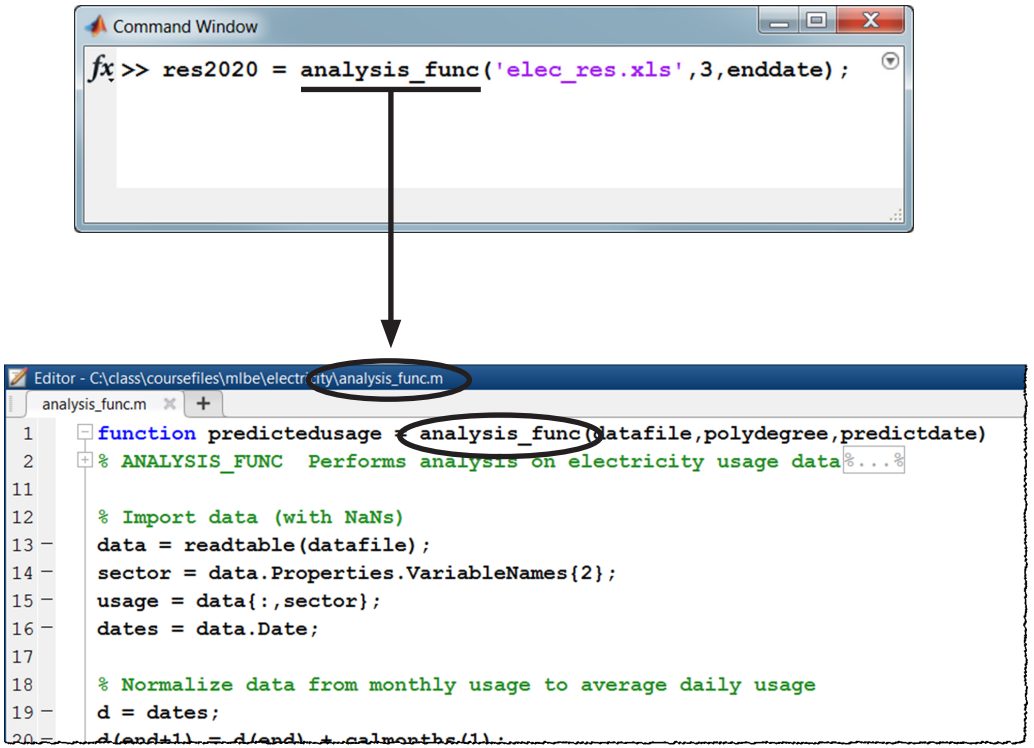
- The keyword `function` must be the first (noncomment) code in the file.
- To avoid confusion, the file should be named *`function_name`*`.m` (that is, the function name used in the declaration should match the file name).

Note MATLAB always uses the function definition that is saved on disk. Be sure to save any changes made to a function before calling it.

Try

Edit `analysis_plaintext.m` to turn it into a function that accepts a file name, a polynomial degree (scalar value), and a date as inputs, and returns the usage predicted by the model as an output.

Solution: `analysis_func.m`



Calling Precedence

Whenever MATLAB encounters the word `analysis` (at the command prompt or in an executing code file), it looks for the first instance of something called `analysis` from the following list:

Location		Reference
1	**Variable in the local workspace**	**Chapter 3**
2	Function from an imported package	`doc import`
3	Nested function within the current function	*MATLAB® Programming Techniques*
4	**Local function within the same file**	**Chapter 11**
5	Private function	*MATLAB® Programming Techniques*
6	Class method	*MATLAB® Programming Techniques*
7	Class constructor in an @ folder	*Object-Oriented Programming with MATLAB®*
8	**File in the current directory**	**Chapter 3**
9	**File on the MATLAB path**	**Next page**

(Items in bold are discussed in this course.)

You can use the `which` command to see how MATLAB is resolving a particular reference:

```
which analysis
which analysis -all
```

Try

Determine which files are being accessed.

```
>> which analysis_func
>> which analysis_func -all
```

Observe how local variables take precedence over functions.

```
>> which mean -all
>> mean = 2
>> which mean
>> which mean -all
>> mean(1)
>> mean([1 5 3])
>> clear mean
>> which mean
```

If there is more than one file named `analysis` in the directory found in step 8 or 9, MATLAB gives precedence according to extension:

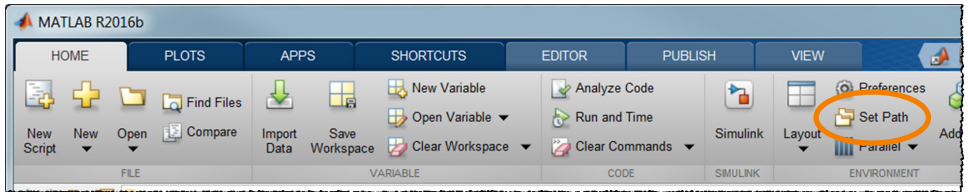
1. Built-in files (`.bi` extension)
2. MATLAB executable (MEX) files (`.mex*` extension)
3. Simulink® models (`.slx` or `.mdl` extension)
4. MATLAB live scripts (`.mlx` extension)
5. Obfuscated MATLAB code files (`.p` extension)
6. MATLAB code files (`.m` extension)

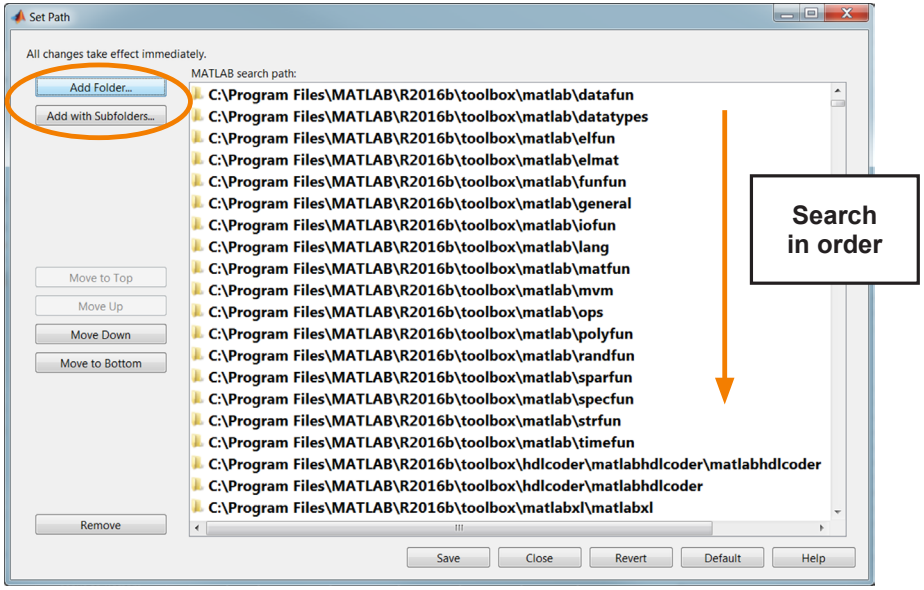
The MATLAB® Path

Rather than search everywhere on your computer (and possibly your network) to determine if `analysis` is a valid command or function call, MATLAB has its own *search path* of directories where it will look for code files.

During installation, all files supplied with MATLAB and any installed MathWorks® toolboxes are placed on the search path. If you create new MATLAB related files, you must ensure that the directories containing the files are placed on the search path. Subdirectories must be explicitly added, even if parent directories are on the path.

To set and edit the search path, open the **Set Path** dialog by clicking the **Set Path** button in the **Environment** section of the **Home** tab of the toolstrip:





The path can also be edited programmatically using functions such as `path`, `addpath`, `rmpath`, and `savepath`.

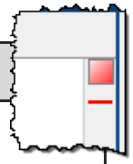
The order of the directories on the search path is important if you have more than one file with the same name (independent of the extension). When MATLAB looks for a file, only the one with highest precedence is found. Other files with the same name are said to be *shadowed*. Use the `which` function to determine which instance of a file is being accessed.

Debugging

Because function workspaces are local and temporary, finding problems with your code can be difficult.

The MATLAB Editor has an integrated code analysis tool that provides a check for syntax errors as the code is being written. For example: strings missing a delimiter (') are colored red, while closed strings turn purple; code inside loops is indented; closing parentheses are briefly highlighted with their opening counterpart.

A small, square box in the upper right corner of the Editor shows the current state of the code.



Color	Meaning
Green	There are no detectable syntax errors in the code.
Orange	There is a potential for unexpected results or poor code performance.
Red	There are syntax errors that will prevent code from running.

Additionally, specific problems within the code appear with an orange or red underline. A description is shown whenever the mouse is hovered over the highlighted code. Sometimes the message suggests a correction or improvement.

A green box indicates that the Code Analyzer could not detect any syntax errors or common problems. This does not mean, however, that the code is necessarily free of run-time errors.

Try

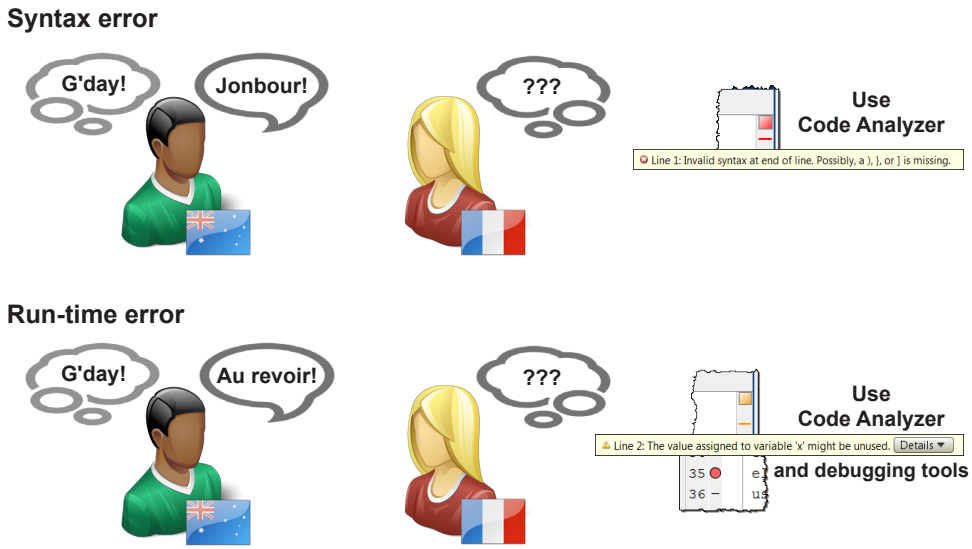
View a version of `analysis_func` that has a bug in it. Use the Code Analyzer messages to find potential problems.

Try to call the function. Did it work? If not, follow the error message link to find where the problem occurred.

```
res2020 = analysis_debug(...
      'elec_res.xlsx',3,enddate)
```

Scripts and functions can invoke other scripts and functions. Hence, when errors do occur, they may originate several layers deep into the progression of function calls. MATLAB shows the error message from each function. Careful review of this *stack trace* may reveal the source of the error.

Error messages in MATLAB give the line number where the error occurred. Clicking the error message will open the appropriate file in the Editor.



Using Breakpoints

If an error message is not enough to diagnose a problem, you can activate the MATLAB Debugger by setting a *breakpoint* in the Editor. Breakpoints are specific lines of code where you would like MATLAB to stop execution and "hold everything" – that is, keep all active function workspaces open and accessible.

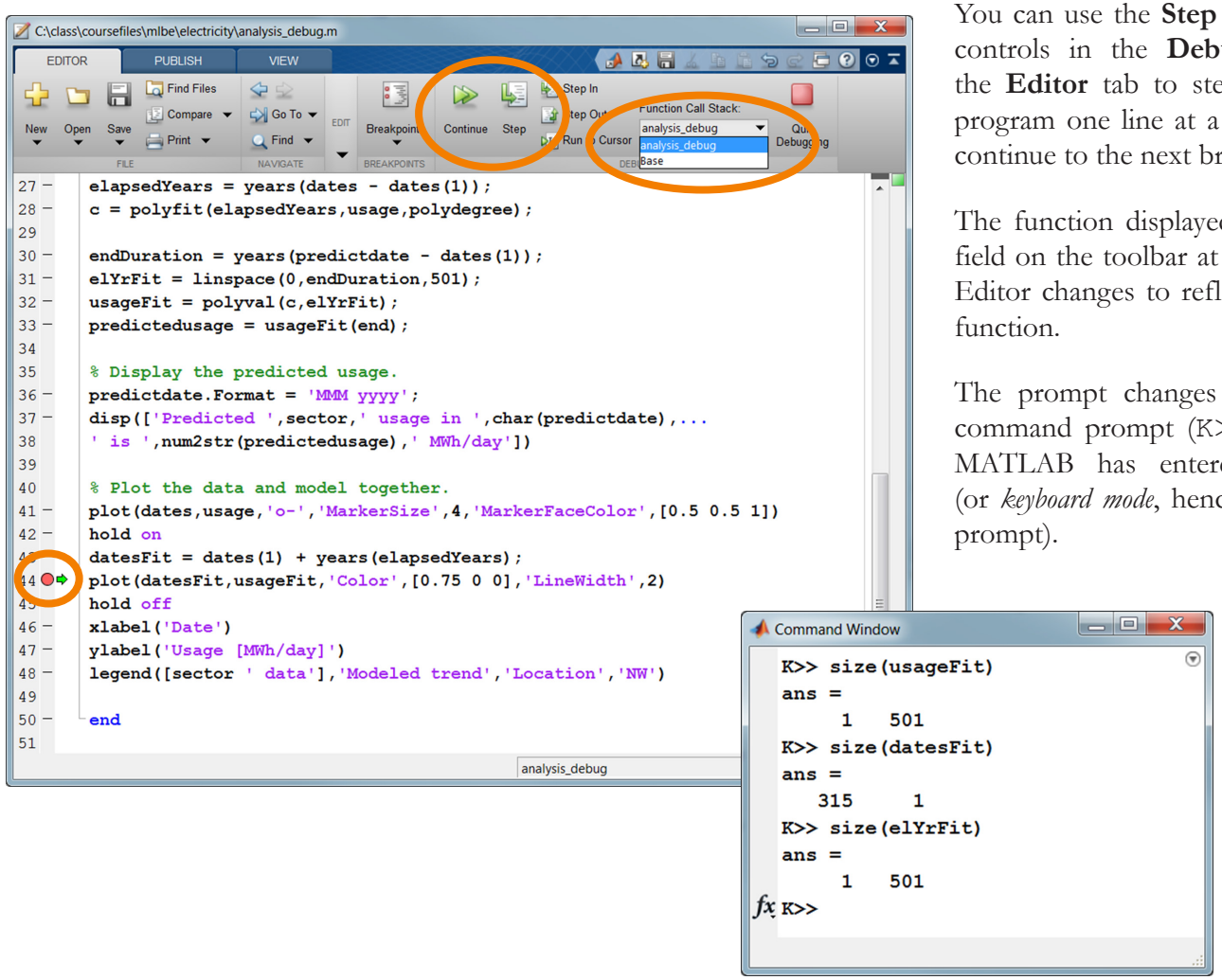
Try

Set a breakpoint in `analysis_debug.m` near a line relevant to the error message. Rerun the previous command to activate debug mode.

To set a breakpoint using the Editor, click a dash in the breakpoint column next to the line number of an executable line of code. The dash turns into a red dot. The breakpoint is disabled (gray) if there is a syntax error or if you have not saved to a directory on the path. To remove the breakpoint, click the dot.

After setting a breakpoint, run the file. The program pauses at the first breakpoint. A green arrow just to the right of the breakpoint column indicates where execution is paused. **Note** The line of code indicated by the arrow has not yet been executed.

A hollow arrow indicates that control is in another function call.



You can use the **Step** and **Continue** controls in the **Debug** section of the **Editor** tab to step through the program one line at a time (F10), or continue to the next breakpoint (F5).

The function displayed in the **Stack** field on the toolbar at the top of the Editor changes to reflect the current function.

The prompt changes to the debug command prompt (`K>>`) to indicate MATLAB has entered *debug mode* (or *keyboard mode*, hence the `K` in the prompt).

Examining Values

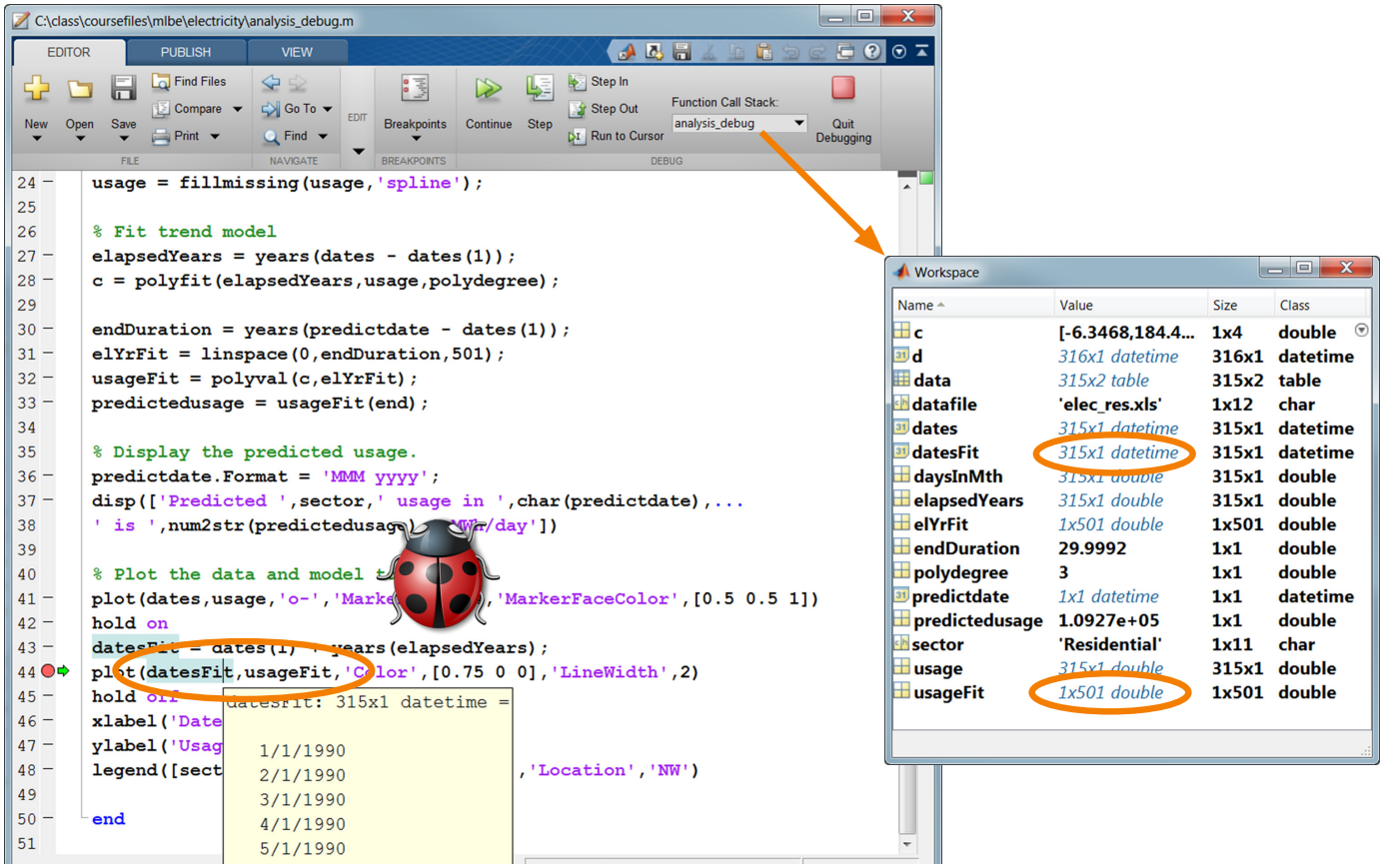
Examine values of variables when you want to see whether a line of code has produced the expected result or not. If the result is as expected, continue running or step to the next line. If not, then that line, or a previous line, contains an error.

In debug mode, you can enter commands in the Command Window to examine values. In the Editor, if you position your cursor near a variable, its current value pops up as a *data tip*. The data tip stays in view until you move the cursor.

Try

Examine the dimensions of the variables in `analysis_debug.m` while in debug mode. Locate and correct the run-time error.

In debug mode, you can choose any active workspace from the **Function Call Stack** in the **Debug** section of the **Editor** tab. The workspace variables then appear in the Workspace browser where they can be examined in the Variable Editor or in the Command Window.



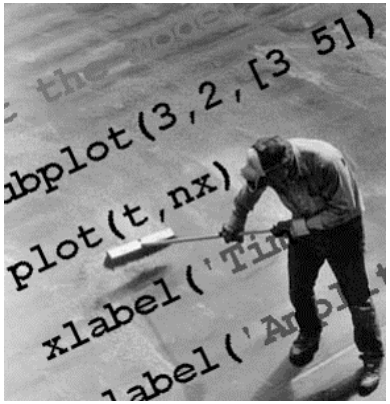
Ending Debugging

While debugging, you can change the value of a variable in the current workspace to see if a new value produces the expected results. While the program is paused, assign a new value to the variable in the Command Window, Workspace browser, or Array Editor, then continue running or stepping through the program. If the new value does not produce the expected results, the program has another problem.

After identifying a problem, end the debugging session. You must end a debugging session if you want to change and save a file to correct a problem, or if you want to run other functions in MATLAB. If you edit a file while in debug mode, you can get unexpected results when you run the file.

To end debugging, click the **Quit Debugging** button. After quitting debugging, the pause arrows in the Editor display no longer appear, and the normal prompt >> reappears in the Command Window. You can no longer access the call stack.

- Exit debug mode
- Correct errors
- Save changes
- Test and confirm
- Clear or disable breakpoints



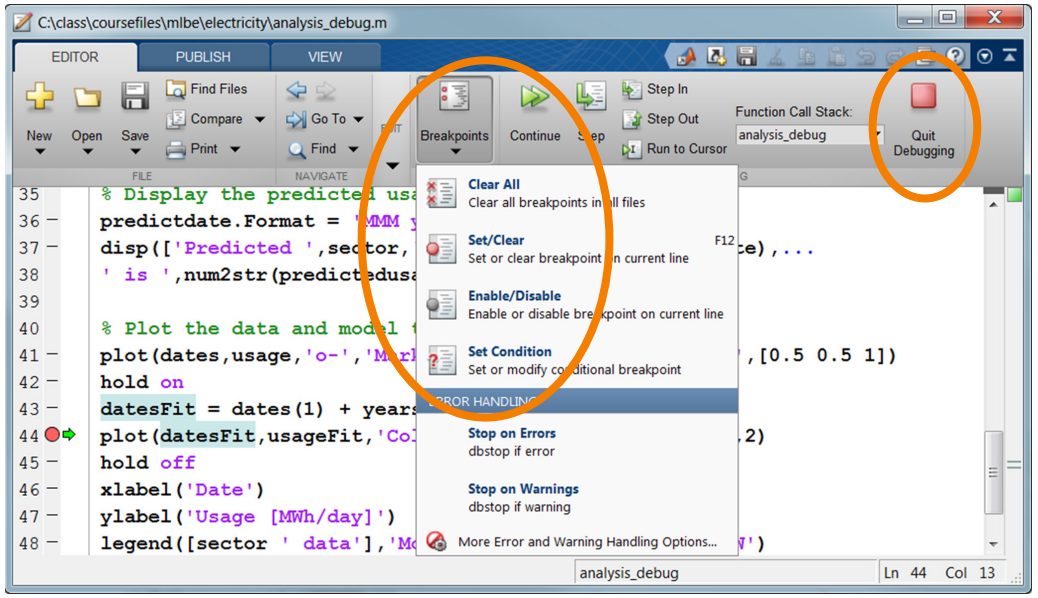
Try

Find and correct all bugs in `analysis_debug.m`. Exit debug mode, save the file, and clear all breakpoints.

Verify that the function is now working correctly.

```
res2020 = analysis_debug(...
      'elec_res.xlsx',3,enddate)
```

After you think you have identified and corrected a problem, you can disable breakpoints so that the program ignores them, or you can remove them entirely. Clicking a breakpoint dot clears it. Right-clicking a breakpoint dot shows a context menu that will allow you to disable the breakpoint (an X will appear through the dot) or clear it.



Course Example: Adding Model Parameters

The function `analysis_inputs.m` implements a more complicated analysis of the electricity usage data, including averaging consecutive blocks of months before fitting a polynomial to a subset of the averaged data.

The function returns the predicted usage, and also optionally creates a plot.

This more complicated algorithm requires more parameters, which means that this function takes six inputs:

- The name of the data file
- The degree of the polynomial to fit to the data
- The future date at which to predict the usage
- The number of months over which to average
- The number of months to use as the data subset to which the model is fitted
- A flag that specifies whether or not to make the plot

Try

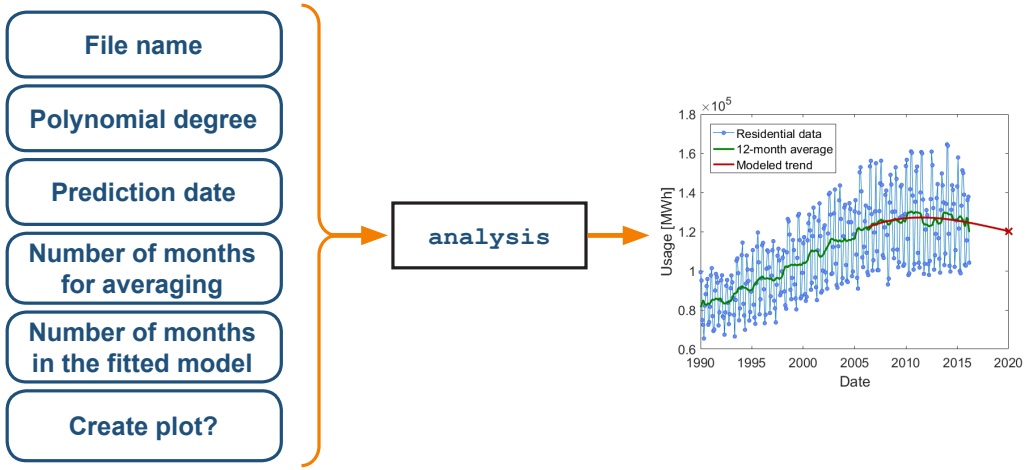
Call the analysis function with six inputs.

```
enddate = datetime(2020,1,1);
res2020 = analysis_inputs('elec_res.xlsx',...
   3,enddate,12,120,true)
```

A function call with six inputs is not easy to interpret. Furthermore, as part of a larger application, some or all of these inputs may be required as inputs to other functions.

In situations like this, it is helpful to have the function accept an input that is a collection of individual parameters. This is commonly achieved with structures, which allow you to combine data of diverse types and sizes. Furthermore, structures use named indexing, which makes interpretation of the data organization easier.

Several MATLAB functions that implement algorithms with many options take this approach. The options are specified in a structure, which is typically created by a separate function.



Combining Heterogeneous Data with Structures

You can use structure arrays to assemble data of dissimilar types and sizes into one variable. Structure arrays use named referencing like tables, but there is no requirement that the data conform to a tabular arrangement.

The command

```
S.apples = 42
```

creates a structure `S` with a field called `apples`, which takes the value of 42.

If `S` is already a structure, the field `apples` will be added to it (if it doesn't exist already). If `S` already exists, but is not a structure, an error will result.

Each field of a structure can contain data of different sizes and types:

```
S.bananas = 'green'
```

Try

Open `analysis_struct.m`. Examine the input parameters. Create a single structure containing necessary model information.

```
mymodel = struct(...
    'datafile','elec_res.xlsx',...
    'polydegree',3,'predictdate',enddate,...
    'monthavg',12,'nummonthmodel',120,...
    'makeplot',true)
```

Pass a single structure argument to a function that analyzes electricity usage data.

```
res2020 = analysis_struct(mymodel)
```

As usual, referencing a nonexistent fieldname will result in an error, but assigning to a nonexistent fieldname will cause that field to be created:

```
x = S.oranges          % error!
S.oranges = [1,2,3]    % OK
```

Summary

- Creating and calling functions
- Workspaces
- Plain text code files
- Path and precedence
- Debugging
- Using structures

Function/Keyword	Use
`function`	Define a function
`which`	Determine the interpretation of a function or variable name

`path`
`addpath`
`rmpath`
`savepath`
	Programmatically query and modify the search path

`struct`	Create a structure out of workspace variables

Test Your Knowledge

Name:_____

1. Suppose the workspace contains a vector `x` and the function `foo` creates a scalar variable also called `x`. What will happen to the vector `x` if you issue the command `z = foo(7)`?

   A. It will change to the scalar value defined in `foo`.

   B. Nothing, because the two variables are of different dimensions (i.e., a vector cannot be changed to a scalar).

   C. Nothing, because the vector `x` is not passed as an argument to `foo`.

   D. Nothing, because `foo` maintains its own workspace, so there is no name conflict.

2. Which of the following is a valid function declaration?

   A. `function [x,y] = foo(a,b)`

   B. `function foo(a,b) = [x,y]`

   C. `[x,y] = function foo(a,b)`

   D. `[x,y] = foo(a,b)`

3. (Select all that apply) Which of the following are valid calls to a function with the declaration line `function [x,y] = foo(a,b)`?

   A. `[x,y] = foo(a)`

   B. `x = foo(a,b)`

   C. `[x,y] = foo[a,b]`

   D. `(x,y) = foo(a)`

   E. `[x,y] = foo(a,b)`