MathWorks®

Technical Articles and Newsletters

Tips and Tricks: Writing MATLAB Functions with Flexible Calling Syntax

MATLAB® functions often have flexible calling syntax with required inputs, optional inputs, and name-value pairs. While this flexibility is convenient for the end user, it can mean a lot of work for the programmer who must implement the input handling. You can greatly reduce the amount of code needed to handle input arguments by using the `inputParser` object.

Our goal is to define a function with the following calling syntax:

```
function a = findArea(width,varargin)
% findArea(width)
% findArea(width,height)
% findArea(... 'shape',shape)
```

With `inputParser` you can specify which input arguments are required (`width`), which are optional (`height`), and which are optional name-value pairs (`'shape'`). `inputParser` also lets you confirm that each input is valid—for instance, that it is the right size, shape, or data type. Finally, `inputParser` lets you specify default values for any optional inputs.

`inputParser` is a MATLAB object. To use it, you first create an object and then call functions to add the various input arguments. Let's start by adding the required input argument, `width`:

```
p = inputParser;
addRequired(p,'width');
```

Our input parser ensures that the user has specified at least one input argument. We want to go one step further—to make sure that the user entered a numeric value. We include a validation function, which is a handle to the built-in function `isnumeric`:

```
p = inputParser;
addRequired(p,'width',@isnumeric);
```

The input parser will now generate an error if given a value for width that is not numeric.

When we add the optional height input, we also include a default value:

```
defaultHeight = 1;
addOptional(p,'height',defaultHeight,@isnumeric);
```

Adding support for the `'shape'` name-value pair is trickier, since we must make sure that the user entered either `'square'` or `'rectangle'`. We create a custom anonymous function that returns true if the input string matches, and false if it does not:

```
defaultShape = 'rectangle';
checkString = @(s) any(strcmp(s,{'square','rectangle'}));
addParamValue(p,'shape',defaultShape,checkString);
```

Now that we have defined our input parsing rules, we are ready to parse the inputs. We pass the function inputs to the `parse` function of the `inputParser`, using `{:}` to convert the input cell array `varargin` to a comma-separated list of input arguments. We get the validated user input values from the `Results` structure in our `inputParser` to use in the rest of the function:

```
parse(p,width,varargin{:});
width  = p.Results.width;
height = p.Results.height;
shape  = p.Results.shape;
```

To see how the code snippets in this article fit together, read the input validation example in the `inputParser` documentation. You'll then be ready to use `inputParser` to offer flexible calling syntax in your own MATLAB functions.

Published 2012 - 92053v00

Products Used

- MATLAB

Learn More

- inputParser Documentation

Technical Articles and Newsletters

View Articles for Related Capabilities

- Algorithm Development

Join the conversation