Introduction

> Style: 1b. the shadow-producing pin of a sundial.
> 2c. the custom or plan followed in spelling, capitalization, punctuation, and typographic arrangement and display.
>
> – *Webster's New Collegiate Dictionary*

The syntax of a programming language tells you what code it is possible to write – what machines will understand. Style tells you what you ought to write – what humans reading the code will understand. Code written with a consistent, readable style is robust, maintainable, and contains fewer bugs. Code written with no regard to style contains more bugs, and it may simply be thrown away and replaced rather than supported and extended.

Attending to style is particularly important when developing software as a team. Consistent style facilitates communication because it enables team members to read and understand each other's work more easily. The value of consistent programming style increases dramatically with the number of people working with the code and with the amount of code in the project.

Two style guides are classics: Strunk and White's *The Elements of Style* and Kernighan and Plauger's *The Elements of Programming Style*. These small books work because they are simple: a list of rules, each containing an explanation and examples of correct, and sometimes incorrect, use. This book follows the same pattern.

Some of the advice in this book may seem obvious to you, particularly if you have been writing and reading code for a long time. You may disagree with some of the specific suggestions about formatting, naming, or usage. Some guidelines will require trade-offs, for example, making identifiers meaningful, easy to read, and of modest length. The most important message is that you practice consistency. Departures should be a conscious choice.

What I tried to do here is distill decades of development experience into an easily accessible set of heuristics that encourage consistent coding practice (and hopefully help you avoid some coding traps along the way). The idea is to provide a clear standard to follow so you can spend your time solving the customer problems instead of worrying, or even arguing, about things such as naming conventions and formatting.

Issues of style are becoming increasingly important as the MATLAB language changes and its use becomes more widespread. Usage has grown from small-scale demonstration or prototype code to large and complex production code developed by teams. In the early versions, all variables were double-precision matrices; now many data types are available and useful. Integration with Java is standard, and Java classes can appear in MATLAB code. The MATLAB language now has its own object-oriented features. These changes have made clear code writing more important and more challenging.

The goal of these guidelines is to help you produce code that is more likely to be correct, understandable, sharable, maintainable, and extendable. Here is the test: when people look at your code, will they see what you are doing? The spirit of this book can be pithily expressed as "Avoid write-only code."

Several practices can help you become more productive with these guidelines. Understand the motivation. There will be

times when you do not want to follow a guideline. Understanding the motivation will help you decide what to do. Choose the guidelines that work for you. Be consistent. Using some set of guidelines consistently is better than inconsistent practice.

Follow the guidelines *while* writing code. Do not expect to go back and apply them later. They are not just window dressing. They really do help. Make the guidelines part of the code inspection process. Teach them to new programmers. Adopt them incrementally if that works better in your situation.

Disclaimer

I dramatically simplified the code samples used in this book to highlight the concepts related to a particular guideline and to fit the relatively narrow text width of the book. In many cases, these code fragments do not conform to the conventions described elsewhere in this book. Do not treat these fragments as definitive examples of real code!

Some elements of the MATLAB product change from release to release. The guidelines in this book apply to recent releases, but some of the details may apply only to the most recent.

1.

General Principles

Good software gets the immediate job done. But great software, written with a consistent and readable style, is predictable, robust, maintainable, supportable, and extensible. A few general principles provide the foundation for great software.

1. Avoid Causing Confusion

Avoid doing things that would be an unpleasant surprise to other software developers. The interfaces and the behavior exhibited by your software must be predictable and consistent. If they are not, then the documentation must clearly identify and justify any unusual instances of use or behavior.

To minimize the chances that anyone would encounter something surprising in your software, you should emphasize the following characteristics in its design, implementation, packaging, and documentation:

Simplicity	Meet the expectations of your users with simple functions, classes, and methods.
Clarity	Ensure that each variable, function, class, and method has a clear purpose.
Completeness	Provide at least the minimum functionality that any reasonable user would expect to find and use.
Consistency	Design similar entities with a similar look and behavior. Create and apply consistent standards whenever possible.

Robustness Provide predictable, documented behavior in response to errors and exceptions. Do not hide errors, and do not force users to detect errors.

2. Avoid Throw-Away Code

Apply these rules to any code you write, not just code destined for production. All too often, some piece of prototype or experimental code will make its way into a finished product. Even if your code never makes it into production, someone else may still have to read it.

Anyone who must look at your code will appreciate your professionalism and foresight at having applied consistent rules from the start. That person may well be you, looking at the code, trying to figure it out long after you wrote it.

3. Help the Reader

People who read your code are not passive. They actively try to interpret and organize it in their brains. You can help the reader by making your naming consistent with meaning, by using layout consistent with logical organization, and by supplying additional relevant information. You may well be that reader, even while writing the code.

4. Maintain the Style of the Original

When modifying existing software that works, your changes should usually follow the style of the original code. Do not attempt to rewrite the old software just to make it match a new style. Rewriting old code simply to change its style may result in the introduction of costly, yet avoidable defects. The use of different styles within a single source file produces code that is more difficult to read and comprehend.

If the existing style is problematic, then consider changing the style of the entire file to make it consistent. Use of Smart

Indent to clarify structure through layout is relatively safe. Carefully changing the comments for compatibility with the Help browser is also worth considering. If you make changes, then be sure to run regression tests and any existing software that interacts with the module.

5. Document Style Deviations

No standard is perfect, and no standard is universally applicable. Sometimes you will find yourself in a situation where you need to deviate from an established standard. If so, then strive for clarity and consistency.

Before you decide to ignore a rule, you should first make sure you understand why the rule exists and what the consequences are if the rule is not applied. If you decide you must violate a rule, then document why you have done so. Some organizations will have reasons to deviate from some of these guidelines, but most organizations will benefit from adopting written style guidelines.

2.

Formatting

Graphic designers have long known that the appropriate use of space around and within text can enhance the reading experience. Use layout, white space, and visual organization to clarify relationships and avoid straining the reader's short-term memory. When you modify your code, preserve layout and spacing to make sure that its format is correct. Code that is easy to scan and read is more likely to be correct.

Layout

The purpose of layout is to help the reader understand the code. It should accurately and consistently represent the logical structure of the code. Indentation is particularly helpful for revealing code structure and patterns to provide context for individual statements.

6. Keep Content Within the First Eighty Columns

Avoid writing code in long lines. Short lines are easier to read than long ones. In general, the readability of text decreases as column width increases. The recommended eighty-column width is a common dimension for editors, terminal emulators, printers, and debuggers.

Readability improves if unintentional line breaks and horizontal scrolling are avoided when passing a file between programmers. Limiting lines to eighty columns also makes side-by-side viewing in two windows easier.

7. Split Long Code Lines at Graceful Points

A long line is one that exceeds the suggested eighty-column limit. If you have two or more statements on one line, then write each on a line of its own. If a line is too long because it contains a complicated expression, then rewrite the code to make one or more subexpressions on separate lines. Give the subexpressions meaningful names. Keep parenthetical elements together, if possible. The statement

```
if isnan(thisValue)&(thisValue>=initial
Threshold) &~ismember(value,valueArray)
  :
end
```

should be replaced by something like

```
valueIsPresent = ~isnan(thisValue);
valueIsValid = thisValue >= initialThreshold;
valueIsNew = ~ismember(thisValue, valueArray);
if (valueIsPresent && valueIsValid && valueIsNew)
  :
end
```

If the line is still too long, then wrap it at a point that is most easily readable. In general, break after a comma, right parenthesis, string terminator, or space. For example,

```
thisComment = ['All values above ' ...
    int2str(threshold) ' are discarded.'];
```

If the line is still too long, then wrap after an operator:

```
currentThreshold = baseValue+thisOffset- ...
    thisFudgeFactor;
```

8. Indent Statement Groups Three or Four Spaces

Good indentation is probably the single best way to reveal program structure. It can set off groups of statements from

other code. It can also clarify the beginning and end of loops or conditional statements.

Indentation of one space is too small to be clearly visible, so indentation of two spaces is sometimes suggested to reduce the number of line breaks required to stay within eighty columns for nested statements. However, two-character indentation is marginally visible, and in any case, MATLAB code is usually not deeply nested.

Indentation larger than four spaces can make nested code difficult to read because it increases the chance that the lines must be split. Indentation of four spaces is the current default in the MATLAB Editor; three was the default in some previous versions.

Do not rely on counting spaces for indentation. Establish your standard in the Editor Preferences menu. Match both the tab size and indent size settings to achieve consistent display.

9. Indent Consistently with the MATLAB Editor

The MATLAB Editor automatically provides indentation that clarifies code structure and is consistent with most recommended practices for C++ and Java. If you use a different editor, then set it and your MATLAB Editor to produce consistent indenting so that you can easily transfer code.

Indentation problems when using the Editor usually occur when the code is not written in sequence, for example, when code is inserted with copy and paste or another case is added after a selection construct is initially written. Use the Smart Indent feature in the Text menu to provide uniform indentation at the right places.

10. Do Not Use Hard Tabs

If you use the MATLAB Editor, then select the option "Tab key inserts spaces" in the Preferences menu. If you use a different editor, then do not rely on hard tabs for indentation.

Hard tabs can easily produce different amounts of indentation when displayed in different editors.

11. Put Only One Executable Statement in a Line of Code

This practice improves readability and allows the JIT/Accelerator to improve performance. Putting multiple statements on one line for conditional constructs or loops lacks an indentation cue for the reader. It may also produce lines that are too long. One line conditional constructs also tend to leave out the often important `else` code.

Replace the conditional

```
if beta >= delta, alpha = beta; end
```

with

```
if beta >= delta
    alpha = beta;
end
```

Replace the one line loop

```
for k = 1:10, y = k.^2; end;
```

with an indented loop.

```
for k = 1:nValues
    y = k.^2;
end
```

12. Define Each Variable on a Separate Line

Using one line for multiple definitions tends to produce overcrowded expressions, variable names that are too short, and unnecessary literal numbers.

Replace

```
rows=10; cols=5; array=zeros(10,5);
```