WIKIPEDIA
The Free Encyclopedia

Programming style

Programming style, also known as **code style**, is a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers read and understand source code conforming to the style, and help to avoid introducing errors.

A classic work on the subject was *The Elements of Programming Style*, written in the 1970s, and illustrated with examples from the Fortran and PL/I languages prevalent at the time.

The programming style used in a particular program may be derived from the coding conventions of a company or other computing organization, as well as the preferences of the author of the code. Programming styles are often designed for a specific programming language (or language family): style considered good in C source code may not be appropriate for BASIC source code, etc. However, some rules are commonly applied to many languages.

Elements of good style

Good style is a subjective matter, and is difficult to define. However, there are several elements common to a large number of programming styles. The issues usually considered as part of programming style include the layout of the source code, including indentation; the use of white space around operators and keywords; the capitalization or otherwise of keywords and variable names; the style and spelling of user-defined identifiers, such as function, procedure and variable names; and the use and style of comments.

Code appearance

Programming styles commonly deal with the visual appearance of source code, with the goal of readability. Software has long been available that formats source code automatically, leaving coders to concentrate on naming, logic, and higher techniques. As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without debates.

Indentation

Indentation styles assist in identifying control flow and blocks of code. In some programming languages, indentation is used to delimit logical blocks of code; correct indentation in these cases is more than a matter of style. In other languages, indentation and white space do not affect function, although logical and consistent indentation makes code more readable. Compare:

```
if (hours < 24 && minutes < 60 && seconds < 60) {
    return true;
} else {
    return false;
}
```

or

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
```

```
    return true;
}
else
{
    return false;
}
```

with something like

```
if  ( hours   < 24
   && minutes < 60
   && seconds < 60
)
{return    true
;}         else
{return   false
;}
```

The first two examples are probably much easier to read because they are indented in an established way (a "hanging paragraph" style). This indentation style is especially useful when dealing with multiple nested constructs.

ModuLiq

The ModuLiq Zero Indentation Style groups with carriage returns rather than indentations. Compare all of the above to:

```
if (hours < 24 && minutes < 60 && seconds < 60)
return true;

else
return false;
```

Lua

Lua does not use the traditional curly braces or parentheses; rather, the expression in a conditional statement must be followed by `then`, and the block must be closed with `end`.

```
if hours < 24 and minutes < 60 and seconds < 60 then
  return true
else
  return false
end
```

Indentation is optional in Lua. `and`, `or`, and `not` function as logical operators in Lua.

They are true/false statements, as

```
print(not true)
```

would mean false.

Python

Python uses indentation to indicate control structures, so *correct indentation* is required. By doing this, the need for bracketing with curly braces (i.e. { and }) is eliminated. On the other hand, copying and pasting Python code can lead to problems, because the indentation level of the pasted code may not be the same as the indentation level of the current line. Such reformatting can be tedious to do by hand, but some text editors and IDEs have features to do it automatically. There are also problems when Python code being rendered unusable when posted on a forum or web page that removes white space, though this problem can be avoided where it is possible to enclose code in white space-preserving tags such as "<pre> ... </pre>" (for HTML), "[code]" ... "[/code]" (for bbcode), etc.

```
if hours < 24 and minutes < 60 and seconds < 60:
    return True
else:
    return False
```

Notice that Python does not use curly braces, but a regular colon (e.g. `else:`).

Many Python programmers tend to follow a commonly agreed style guide known as PEP8.[1] There are tools designed to automate PEP8 compliance.

Haskell

Haskell similarly has the off-side rule, i.e. it has a two-dimension syntax where indentation is meaningful to define blocks (although, an alternate syntax uses curly braces and semicolons). Haskell is a declarative language, there are statements, but declarations within a Haskell script. Example:

```
let c_1 = 1
    c_2 = 2
in
    f x y = c_1 * x + c_2 * y
```

may be written in one line as:

```
let {c_1=1;c_2=2} in f x y = c_1 * x + c_2 * y
```

Haskell encourages the use of literate programming, where extended text explains the genesis of the code. In literate Haskell scripts (named with the `lhs` extension), everything is a comment except blocks marked as code. The program can be written in LaTeX, in such case the `code` environment marks what is code. Also, each active code paragraph can be marked by preceding and ending it with an empty line, and starting each line of code with a greater than sign and a space. Here an example using LaTeX markup:

```
The function \verb+isValidDate+ test if date is valid
\begin{code}
isValidDate :: Date -> Bool
isValidDate date = hh>=0  && mm>=0 && ss>=0
                 && hh<24 && mm<60 && ss<60
 where (hh,mm,ss) = fromDate date
\end{code}
observe that in this case the overloaded function is \verb+fromDate :: Date -> (Int,Int,Int)+.
```

And an example using plain text:

```
The function isValidDate test if date is valid
```

```
> isValidDate :: Date -> Bool
> isValidDate date = hh>=0  && mm>=0 && ss>=0
>                  && hh<24 && mm<60 && ss<60
>  where (hh,mm,ss) = fromDate date

observe that in this case the overloaded function is fromDate :: Date -> (Int,Int,Int).
```

Vertical alignment

It is often helpful to align similar elements vertically, to make typo-generated bugs more obvious. Compare:

```
$search = array('a', 'b', 'c', 'd', 'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

$value = 0;
$anothervalue = 1;
$yetanothervalue = 2;
```

with:

```
$search      = array('a',   'b',   'c',   'd',   'e');
$replacement = array('foo', 'bar', 'baz', 'quux');

// Another example:

$value           = 0;
$anothervalue    = 1;
$yetanothervalue = 2;
```

The latter example makes two things intuitively clear that were not clear in the former:

- the search and replace terms are related and match up: they are not discrete variables;
- there is one more search term than there are replacement terms. If this is a bug, it is now more likely to be spotted.

However, note that there are arguments *against* vertical alignment:

- **Inter-line false dependencies**; tabular formatting creates dependencies across lines. For example, if an identifier with a long name is added to a tabular layout, the column width may have to be increased to accommodate it. This forces a bigger change to the source code than necessary, and the essential change may be lost in the noise. This is detrimental to Revision control where inspecting differences between versions is essential.
- **Brittleness**; if a programmer does not neatly format the table when making a change, maybe legitimately with the previous point in mind, the result becomes a mess that deteriorates with further such changes. Simple refactoring operations, such as search-and-replace, may also break the formatting.
- **Resistance to modification**; tabular formatting requires more effort to maintain. This may put off a programmer from making a beneficial change, such as adding, correcting or improving the name of an identifier, because it will mess up the formatting.
- **Reliance on mono-spaced font**; tabular formatting assumes that the editor uses a fixed-width font. Many modern code editors support proportional fonts, and the programmer may prefer to use a proportional font for readability.
- **Tool dependence**; some of the effort of maintaining alignment can be alleviated by tools (e.g. a source code editor that supports elastic tabstops), although that creates a reliance on such tools.

For example, if a simple refactoring operation is performed on the code above, renaming variables "$replacement" to "$r" and "$anothervalue" to "$a", the resulting code will look like this:

```
$search      = array('a',   'b',   'c',   'd',   'e');
$r = array('foo', 'bar', 'baz', 'quux');

// Another example:

$value           = 0;
$a    = 1;
$yetanothervalue = 2;
```

The original sequential formatting will still look fine after such change:

```
$search = array('a', 'b', 'c', 'd', 'e');
$r = array('foo', 'bar', 'baz', 'quux');

// Another example:

$value = 0;
$a = 1;
$yetanothervalue = 2;
```

Spaces

In those situations where some white space is required, the grammars of most free-format languages are unconcerned with the amount that appears. Style related to white space is commonly used to enhance readability. There are currently no known hard facts (conclusions from studies) about which of the whitespace styles have the best readability.

For instance, compare the following syntactically equivalent examples of C code:

```
int i;
for(i=0;i<10;++i){
    printf("%d",i*i+i);
}
```

versus

```
int i;
for (i = 0; i < 10; ++i) {
    printf("%d", i * i + i);
}
```

Tabs

The use of tabs to create white space presents particular issues when not enough care is taken because the location of the tabulation point can be different depending on the tools being used and even the preferences of the user.

As an example, one programmer prefers tab stops of four and has their toolset configured this way, and uses these to format their code.

```
int     ix;     // Index to scan array
long    sum;    // Accumulator for sum
```

Another programmer prefers tab stops of eight, and their toolset is configured this way. When someone else examines the original person's code, they may well find it difficult to read.

```
int             ix;             // Index to scan array
long    sum;    // Accumulator for sum
```

One widely used solution to this issue may involve forbidding the use of tabs for alignment or rules on how tab stops must be set. Note that tabs work fine provided they are used consistently, restricted to logical indentation, and not used for alignment:

```
class MyClass {
    int foobar(
        int qux, // first parameter
        int quux); // second parameter
    int foobar2(
        int qux, // first parameter
        int quux, // second parameter
        int quuux); // third parameter
};
```

See also

- Coding conventions
- MISRA C
- Naming convention (programming)

References

1. "PEP 0008 -- Style Guide for Python Code" (https://www.python.org/dev/peps/pep-0008/). python.org.

External links

- Source Code Formatters (https://curlie.org//Computers/Programming/Development_Tools/Source_Code_Formatters/) at Curlie