Use underscores to separate words. It is generally easier to read $var_names_like_this than $VarNamesLikeThis, especially for non-native speakers of English. It's also a simple rule that works consistently with VAR_NAMES_LIKE_THIS.

Package/Module names are an exception to this rule. Perl informally reserves lowercase module names for 'pragma' modules like integer and strict. Other modules normally begin with a capital letter and use mixed case with no underscores (need to be short and portable).

You may find it helpful to use letter case to indicate the scope or nature of a variable. For example:

```
$ALL_CAPS_HERE   constants only (beware clashes with Perl vars)
$Some_Caps_Here  package-wide global/static
$no_caps_here    function scope my() or local() variables
```

Function and method names seem to work best as all lowercase. e.g., `$obj->as_string()`.

You can use a leading underscore to indicate that a variable or function should not be used outside the package that defined it.

- Select what to export.

  Do NOT export method names!

  Do NOT export anything else by default without a good reason!

  Exports pollute the namespace of the module user. If you must export try to use @EXPORT_OK in preference to @EXPORT and avoid short or common names to reduce the risk of name clashes.

  Generally anything not exported is still accessible from outside the module using the ModuleName::item_name (or `$blessed_ref->method`) syntax. By convention you can use a leading underscore on names to indicate informally that they are 'internal' and not for public use.

  (It is actually possible to get private functions by saying: `my $subref = sub { ... }; &$subref;`. But there's no way to call that directly as a method, because a method must have a name in the symbol table.)

  As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then @EXPORT_OK anything but use @EXPORT with caution.

- Select a name for the module.

  This name should be as descriptive, accurate, and complete as possible. Avoid any risk of ambiguity. Always try to use two or more whole words. Generally the name should reflect what is special about what the module does rather than how it does it. Please use nested module names to group informally or categorize a module. There should be a very good reason for a module not to have a nested name. Module names should begin with a capital letter.

  Having 57 modules all called Sort will not make life easy for anyone (though having 23 called Sort::Quick is only marginally better :-). Imagine someone trying to install your module alongside many others. If in any doubt ask for suggestions in comp.lang.perl.misc.

  If you are developing a suite of related modules/classes it's good practice to use nested classes with a common prefix as this will avoid namespace clashes. For example: Xyz::Control, Xyz::View, Xyz::Model etc. Use the modules in this list as a naming guide.

  If adding a new module to a set, follow the original author's standards for naming modules and the interface to methods in those modules.

  If developing modules for private internal or project specific use, that will never be released to the public, then you should ensure that their names will not clash with any future public module. You can do this either by using the reserved Local::* category or by using a category name that includes an underscore like Foo_Corp::*.

  To be portable each component of a module name should be limited to 11 characters. If it might be used on MS-DOS then try to ensure each is unique in the first 8 characters. Nested modules make this easier.

- Have you got it right?

  How do you know that you've made the right decisions? Have you picked an interface design that will cause problems later? Have you picked the most appropriate name? Do you have any questions?