# THE SKRIPT STYLE GUIDE

unofficial!

## I. Introduction

This document represents a proposed set of coding standards for the "Skript"[1] language, designed to improve code's overall consistency and readability, as well as its overall efficiency and reliability.

Note: This guide reflects *recommended* coding conventions, which may be adapted to one's preferences.

**This guide is <u>not approved or promoted</u> by SkriptLang, nor any other implementers of "Skript".**

### A. The necessity of coding conventions

Having a consistent set of coding standards is important in any programming language, as conventions offer many benefits, among which:

- Conventions keep code consistent.
- Conventions make code easier to read and understand, resulting in simpler maintenance.
- Conventions make development more efficient and help prevent potential coding flaws.

### B. Terminology notes

- The notion of "Skript" refers interchangeably to the language and the implementation as a Bukkit plugin.
- The notion of "script" refers to any file respecting Skript's file format and containing valid Skript code. Generally, scripts use the special ".sk" extension.

These two notions should not be confused; otherwise, various problems can arise when using elements sensitive to this distinction (for example, the loading/unloading events).

[1] This guide is specifically tailored for SkriptLang's implementation of the "Skript" language but may apply to other implementations as well.

## II. The script's macrostructure

*Structures* are the top-level (root-level) elements of any script. They should be placed under any beginning comments the file has, in the following order, separated by the previous structure using a single empty line:

- options structures
- variables structures
- aliases structures
- other metadata structures[2]
- function declarations
- other callable structures[3]
- event listeners (in ascending order of priority)
- custom command declarations

Multiple structures that fall in the same one of the above categories should be placed in an *explainable logical* order to each other.

---

[2] Metadata structures are used to provide information for other elements (for example, skript-reflect's **import** section).

[3] Callable structures, as their name suggests, can be dynamically run from the script they are present in or from other scripts (for example, custom syntax structures from skript-reflect).

## III. Formatting

Formatting is an essential aesthetic factor in programming. Proper formatting keeps code easy to read, understand, and modify. By following a clear structure, avoiding errors and finding problems becomes much easier.

### A. Indentation

Indenting can be done using either tabs or spaces, although consistency is mandated throughout structures and recommended throughout all scripts. Indentation should consistently use a width of four.

Each time a new block of code begins, started by a new section, the indent increases. When the block ends, the indent returns to the previous indent level. The indent level applies both to the code and comments throughout the block.

### B. Column limit

The length of every line of code is important. Generally, the longer a line is, the longer it will take to be parsed. Long lines are also harder to read, increasing confusion.

A good practice is keeping every line reasonably short; the key is finding the balance between clarity and length. Due to Skript's syntax being quite verbose, the suggested limit is 120 characters.

Shortening lines ought not be achieved by removing clarifying constructs. Instead, long, hard-to-comprehend lines should be broken down into multiple statements while maintaining clarity.

For example, the code:

```
set {_point} to vector (sin({_pitch}) * cos({_yaw})), (cos({_pitch})),
(sin({_pitch}) * sin({_yaw}))
```

can become:

```
set {_x} to sin({_pitch}) * cos({_yaw})
set {_y} to cos({_pitch})
set {_z} to sin({_pitch}) * sin({_yaw})
set {_point} to vector {_x}, {_y}, {_z}
```

### C. Empty lines

Empty lines can make code easier to read and understand by separating blocks of code by their purpose.

Empty lines should contain no characters (except when used in comments). Structures should *always* be separated from one another using a single empty line to enhance clarity. Empty lines throughout contiguous comment blocks should keep the comment

symbol. A script should never end in an empty line.

Adding empty lines throughout inner blocks is up to the programmer's judgement.

### D. Grouping parentheses

Parentheses can prove helpful in various cases, as they can lower the chance of code misinterpretation both by the user and by the parser. If there is no reasonable chance for misinterpretation, they may be omitted. However, one must not assume that everyone reading the code knows the order in which expressions are evaluated.

Parentheses also help with parsing as they can clearly delimit expressions, making the parser's work easier. On the other hand, overusing them can lead to harder-to-read code.

For example, parentheses are commonly used in arithmetic expressions:

```
set {_multiplier} to (5% * (1 + {_level})) + ({_level} / 2)
```

Conversely, it is **not desirable** to overuse parentheses, especially in evident scenarios:

```
if (({_player}) is (first element of ({thing::*}))): # Why?!?
```

### E. Comments

Comments are human-readable explanations throughout the code detailing what it does. Proper use of comments can make maintenance easier. Commenting is also very important when writing code that other people will use. In Skript, comments are started with a single "#" symbol and continue until the end of the line.

Any script should *always* start with header comments explaining what the code does and what dependencies it might require. Header comments should be separated from the rest of the code by two or three empty lines. (The amount should be kept consistent throughout all files.)

Furthermore, most structures should be introduced using comments detailing what they do. Such comments shouldn't be separated from the structure. Placing comments throughout structures is recommended if it can help explain the role of ambiguous lines.

The comment symbol should *always* be separated by any text using *at least* one space on **both** sides.

For example, the following are recommended:

```
# This is a nice comment :D
set {_phi} to (1 + sqrt(5)) / 2 # This is the golden ratio

set {_list::*} to request_objects()   # Forward the given objects to
propagate({_list::*}, true)           # all active handlers
```

While the following are **discouraged**:

```
#This is a hideous comment!
kill all players#Unacceptable.
```

### F. Custom command declarations

When creating a custom command, entries should be ordered as follows:

- aliases
- executable by
- usage
- description
- permission
- permission message
- cooldown
- cooldown message
- cooldown bypass
- cooldown storage
- trigger

### G. Inline conditions

Inline conditions are conditions that do not need to be indented (not to be confused with the "do if" effect). While in some cases they make code more compact, repeated use of them can diminish readability because, due to their nature, they look more like assertions than conditions. They should be used sparingly.

#### a. Guard clauses

Inline conditions may be used as guard clauses, but a standard **if** block or the "do if" effect is preferred for more clarity.

## IV. Naming

This section will discuss some general guidelines for naming. It's very important that names reflect purpose. To improve readability, the name of any element should refer to what it does. All names should provide just enough information to properly describe what the named objects serve for; nothing less, nothing more. The recommended length limit for an element's name is 20 characters.

Naming should be done using alphanumeric characters, underscores, and hyphens only. (See each object below for more information.) In addition, identifiers for elements of the same type should use a consistent casing style.

Note on acronyms and abbreviations: Acronyms and abbreviations are to be avoided unless the shortened form is much more widely used than the long one (examples: HTML, URL, YAML, etc.). Furthermore, they should follow the casing rules of each element.

### A. Files

The name of a script file should describe the script's purpose clearly and succinctly. The recommendation is to use lowercase alphabetic characters and dashes. Examples: `thing.sk`, `animal-quests.sk`, `mob-farm.sk`.

### B. Variables

Variable names should be kept short and describe either the purpose of the stored object or its type. Variables should be named in *lower-kebab-case*, using alphanumeric characters and hyphens only (and sometimes dots[4]). This naming style applies to function parameters and named command arguments as well.

### C. Options

Options are, in essence, fragments of code that get replaced at parse time. They should follow the same naming style as variables.

### D. Functions

Functions should be named in *lower_snake_case*. Their names should be short and clearly describe their purpose. Function names typically start with a verb or verb phrase and should contain alphanumeric characters and underscores only.

Because all functions share the same namespace, there exists the chance of name conflicts. If a function cannot be simply

[4] Despite popular belief, there is nothing inherently wrong with using "." in variable names.

renamed to avoid such conflicts, one can use specific prefixes or suffixes (for example, the script's name).

Moreover, if a function exists for internal use only, its name can also use a distinguishing prefix or a suffix (for example, **internal_write_to_file, do_thing_unsafe**). This tells other users to be extra careful when using the function or not to use it at all.

## V. Coding practices

This section covers a few helpful practices that can enhance the code's readability, reliability, and overall efficiency.

### A. Conditionals

Always use **if-else if-else** blocks where possible if it makes the code clearer.

Avoid using the "do if" effect repeatedly if it can be replaced with **if-else if-else** blocks because, most of the time, it can be detrimental to performance. For example, the following is **not recommended**:

```
set {_x} to 1 if the entity is a zombie
set {_x} to 2 if the entity is a pig
set {_x} to 3 if the entity is a horse
set {_x} to 4 if the entity is a player
```

The above code is inefficient because every time it's run, it checks *all four* conditions. Using **if-else if** blocks here would limit the number of checks, thus improving performance. The same applies to the "ternary" expression as well.

### B. String comparisons for non-string objects

The use of strings to compare non-text objects is not recommended. This is prone to problems. For example, the following should **not** be done:

```
if "%{_value}%" is "<none>":
```

This is rarely a good idea. The default none text can be changed, breaking scripts that do this.

```
if "%block%" is "oak wood stairs":
```

This is flawed. There is no point in converting these to texts. The comparison is also faulty because it does not account for different block states.

This is mostly encountered in old scripts, but this practice no longer works because the string format of blocks was changed.

There might be cases when string comparison may be needed, most likely due to a problem[5] in Skript itself. In these situations, comparing objects as texts is acceptable but should be changed immediately after the problem has been resolved.

---

[5] If such a problem is encountered, it is always helpful to ensure the maintainers know about it. One should report the issue if it hasn't been signalled already. This speed up the fixing process.

### C. "*Clean code reads like well-written prose*"[6]

Code should promote the "English style" Skript had always aimed to achieve. It should prefer the more verbose, "Englishy" syntaxes over programming jargon or "symbolic nonsense".

#### a. "is" for equality

A notable example is using the "=" symbol for object equality. Not only does this symbol lack the "English style" of Skript, but it can also be confusing when it comes to inline conditions:

```
command /foo <player>:
    trigger:
        {_thing} = 3 # It sets it to 3... or does it?
        add {_thing} to the balance of the argument
```

### D. A none value

Skript does not come packed with any expression to explicitly indicate a none/null value. However, unset variables have been intentionally modified to act like such a value.

When there is a need to reference none (rarely the case), it is recommended to use a special local variable with a meaningful name (for example, `{_none}` or `{_null}`). It is discouraged to use the empty name variable `{_}` because of its inexpressive name.

Evidently, care must be taken to assure that the chosen none variable is never assigned an actual value.

[6] Quote by Grady Booch extracted from "*Clean Code*" by Robert C. Martin

## VI. Closing notes

This was a short presentation of a recommended set of coding standards to be respected while using Skript.

Guide written by Mr. Darth; last updated on 31st August 2022. This guide was published as part of the "Skriptness"[7] collection.

**Special thanks** to Pesekjak and UnderscoreTud for the helpful reviews and suggestions.

---

*"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*

*Martin Fowler*

---

[7] https://github.com/Mr-Darth/Skriptness