THE SKRIPT STYLE GUIDE

unofficial!

I. Introduction

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

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, understand, and maintain.
- Conventions make development more efficient and help prevent potential coding flaws.

B. About this style guide and any others

The practices presented here are **not** in any way obligatory when using Skript. It is up to the programmer to adopt conventions that they find practical. This guide does not advocate for the 'religious' following of these conventions or any other set thereof. Instead, it promotes consistency in general; one does not have to follow these specific standards, but rather adopt a consistent style throughout their projects that helps themselves and others who may read the code. Regardless of the chosen style, it is important to understand that deviations are allowed and, at times, even necessary; however, whenever it's possible, consistency should be protected.

This guide may sometimes not offer a conclusive standard, or it may provide multiple options to pick from (ideally, only one). Such practices are marked with the superscript symbol '⚖' to reflect that the guide refrains from endorsing one way in particular.

C. Terminology notes

- The notion of 'Skript' refers interchangeably to the language and SkriptLang's 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.

---

[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 elements of any script. They are to be placed under any beginning comments the file has, in the following order:

- feature opt-in structures
- other *simple* structures[2]
- options structures
- variables structures
- aliases structures
- other metadata structures[3]
- function declarations
- other executable structures[4]
- event listeners
- 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.

It is preferred that different structures with the same functionality are merged into one. Thus, a script should not have *more than one* 'options' structure, 'variables' structure, 'aliases' structure, etc., unless there are good reasons for a split.

---

[2] Simple structures are structures that do not have a body (i.e., they do not start a section).

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

[4] Executable structures may be dynamically run (called) 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. Following a clear structure makes avoiding errors and finding problems considerably easier.

A. Indentation

Indenting can be done using either tabs or spaces⚖. Although consistency is mandated only throughout structures, it is highly recommended throughout all scripts. Space indentation should consistently use a width of *four*.

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 more difficult for users to read and understand.

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 for even simple things, setting a fixed limit is not a trivial task; however, lines should be limited to *around* 120 characters.

Shortening lines ought not to 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}))
```

May 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 purpose. Empty lines must not contain any characters, except when used in comments.

Non-simple structures must be separated from other structures using a single empty line to enhance clarity.

Empty lines throughout contiguous comment blocks should keep the comment symbol.

A script should not end in an empty line.

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

D. 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 others will read or use. In Skript, line comments are started with a single '#' symbol and continue until the end of the line. Block comments start on a line containing only three consecutive comment marks and whitespace and end on another line with only three consecutive comment marks and whitespace.

Any script should always start with header comments explaining what the code does and what dependencies it might require; additionally, these may include authors, date of last revision, etc. Header comments should be separated from the rest of the code by three empty lines.

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

The comment symbol must be separated by other text using *at least* one space on the left side and *exactly* one space on the right.

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 nasty comment!
kill all players#Unacceptable.
```

E. Grouping parentheses

Parentheses are helpful in various cases, as they can lower the chance of code misinterpretation by both the user and the parser. If there is no reasonable chance for misinterpretation, they may be omitted. Parentheses also help with parsing as they

can explicitly delimit expressions, making the parser's work easier.

For example, parentheses are commonly used to isolate arguments in function calls:

```
location((x of {_v}), (y of {_v}), (z of {_v}))
```

Conversely, it is **not** desirable to overuse parentheses, especially in evident scenarios, since this can lead to harder-to-read code, at (usually) no benefit:

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

F. Operators and other delimiters

Binary operators[5] must be separated from their operands with *exactly one* space on each side. This includes arithmetic expressions, the 'default value' expression, vector offsetting, etc. Binary relations (<,<=, >=, >) must follow the same rules.

Commas in explicit lists must not be preceded by space but be followed by *exactly one* space. The last delimiter in an explicit list must be a conjunction ('and'/'or'/'nor') unless the list represents arguments for a function.

For example, the following are recommended:

```
{_a} + {_b}
get_value() ? {@default}
1, 2, 3 and 4
```

The following are **discouraged**:

```
{multiplier}*{_value}+500
{_origin}~{_displacement}
0,2,4,6,8,10
```

Colon (':') entry separators must follow the same spacing rules as commas, while equals-sign ('=') entry separators must follow the same spacing rules as binary operators.

The following are recommended:

```
prefix: "hello"
{variable} = 3
thingy = minecraft:dirt # The colon is not an entry separator here :)
```

---

[5] Binary operators have an arity of two (i.e., they accept only two operands). This guide considers only binary operators that are written in infix notation.

While the following are **not recommended:**

```
name:banana
{value} =3
```

In function declarations, the parameter list should follow the same rules. The '::' return type delimiter should be separated by *exactly one* space on each side.[6]

```
function send(message: string, recipient: command sender = console) ::
boolean:
```

G. Custom command declarations

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

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

H. 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, their repeated use can sometimes diminish clarity and is thus discouraged (see, for example, V.C.a. 'is' for equality).⚖ Inline conditions may be used as guard clauses, but a standard **if** block or the 'do if' effect is generally preferred.

---

[6]On older versions, this was mandatory. Not adding spaces around '::' would make the function declaration invalid.

IV. Naming

This section will discuss some general guidelines for naming. The general rule is that names must reflect purpose. To improve readability, the name of every element should refer to what it does. All names should provide just enough information to unambiguously describe the named objects' purpose.

Naming is done using alphanumeric characters, underscores, and hyphens only, with a few exceptions depending on what the named object is. (See each object below for more information.)

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

A. Scripts

The name of a script file should describe the script's purpose clearly and succinctly. The recommended casing style is *lower-kebab-case*. Examples: `thing.sk`, `animal-quests.sk`, `mob-farm.sk`. The same applies to folders.

B. Variables

Variable names should be kept short and describe either the purpose of the stored object or its type. Variables are to be named in *lower-kebab-case*, using alphanumeric characters and hyphens only (and sometimes dots[7]). Variable names should always start with a letter.[8] This naming style applies to function parameters and named command arguments as well.

Specific signalling prefixes should be chosen carefully. For example, '-' is popularly chosen for variables not guaranteed to persist across restarts.

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 must be named in *lower_snake_case*. Their names should be short and clearly describe their purpose. Function names

---

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

[8] Specific prefixes ('_', '-', '$', etc.) are not counted as part of the name by this guide. Internally, the only prefix that is treated separately is the (literal) underscore token for local variables. Other prefixes are technically part of the name. This subtle technicality is, however, not relevant for the matter at hand and is ignored here.

typically start with a verb or verb phrase and contain alphanumeric characters and underscores only.

Since all global functions are stored in the same place, name conflicts may appear. If a global function can not be simply renamed to avoid such conflicts, it is recommended to use specific prefixes or suffixes (for example, the script's name). If a function exists for local or internal use only, it should be marked as **local** instead.

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 it checks *all four* conditions every time it's run. Using **if-else if** blocks here would limit the number of checks and thus improve 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 inconsistent and 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 by anyone, easily 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.[9]

Checking by use of the **contains** condition is to be avoided for the same reasons.

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

---

[9] This doesn't actually work in recent Skript versions, because the string format of blocks was changed. It is generally encountered in old scripts.

C. *'Clean code reads like well-written prose'*[10]

Code should promote the 'English style' Skript had always aimed to achieve. It should prefer more verbose, 'Englishy' syntax 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
```

b. Function return types

Function return types may be declared using either the double colon token ('::') or the `returns` keyword.⚖ For historical reasons, this guide favours the former.

c. Being over the top

Writing code that is too wordy isn't recommended either, as mentioned in III.b. Column limit. For instance, the following is **not recommended**, since it clutters the code and makes it harder to follow.

```
on join:
    set the variable {_name} to the name of the player
    if the variable {_name} is "bob":
        set the variable {_location} to location of the player
        set the variable {_destination} to the variable {_location}
offset by vector 2, 2, 2
        teleport the player to the variable {_destination}
        send message "teleport :D" to the player
```

D. A none value

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

When there is a need to reference none (albeit rarely the case), it is recommended to use a special local variable with a meaningful name (for example, `{_none}` or `{_null}`)⚖. For parity with Skript's typical '<none>' text, this guide tends towards `{_none}`.

It is discouraged to use the empty name variable `{_}` because of its inexpressive name.

Evidently, care must be taken to ensure that the chosen variable is never assigned an actual value.

---

[10] Quote by Grady Booch extracted from '*Clean Code*' by Robert C. Martin

VI. Closing notes

This was a presentation of recommended coding standards to be respected while using Skript. To reiterate what was mentioned in the introduction: these standards are **not official** and **not a must**, but are, hopefully, an aid in understanding the importance of adopting a consistent and easy-to-work-with style.

Guide **written** by Mr. Darth; **last updated** on 21st August 2024. This guide was published as part of the 'Skriptness' collection.

Special thanks to Pesekjak, UnderscoreTub, and everyone else 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