

[Overview](#) [Intro and Tutorial](#)[Organization](#) [Expansion](#) [Development](#) [Menu](#)[FAQ](#) [Reference](#)

# Writing snippets

## Table of Contents

- [Snippet development](#)
  - [Quickly finding snippets](#)
  - [Using the snippet-mode major mode](#)
- [File content](#)
  - [# key: snippet abbrev](#)
  - [# name: snippet name](#)
  - [# condition: snippet condition](#)
  - [# group: snippet menu grouping](#)
  - [# expand-env: expand environment](#)
  - [# binding: direct keybinding](#)
  - [# type: snippet or command](#)
  - [# uuid: unique identifier](#)
  - [# contributor: snippet author](#)
- [Template Syntax](#)
  - [Plain Text](#)
  - [Embedded Emacs-lisp code](#)
  - [Tab stop fields](#)
  - [Placeholder fields](#)
  - [Mirrors](#)
  - [Mirrors with transformations](#)
  - [Fields with transformations](#)
  - [Choosing fields value from a list and other tricks](#)
  - [Nested placeholder fields](#)

## Snippet development

### Quickly finding snippets

There are some ways you can quickly find a snippet file or create a new one:

- `M-x yas-new-snippet`

Creates a new buffer with a template for making a new snippet. The buffer is in `snippet-mode` (see below). When you are done editing the new snippet, use `C-c C-c` to save it. This will prompt for a directory two steps: first, the snippet table (with a default based on the major mode you started in), and then the snippet collection directory (defaults to the first directory in `yas-snippet-dirs`. (See [Organizing Snippets](#) for more detail on how snippets are organized.)

- `M-x yas-find-snippets`

Lets you find the snippet file in the directory the snippet was loaded from (if it exists) like `find-file-other-window`. The directory searching logic is similar to `M-x yas-new-snippet`.

- `M-x yas-visit-snippet-file`

Prompts you for possible snippet expansions like [yas-insert-snippet](#), but instead of expanding it, takes you directly to the snippet definition's file, if it exists.

Once you find this file it will be set to `snippet-mode` (see ahead) and you can start editing your snippet.

## Using the `snippet-mode` major mode

There is a major mode `snippet-mode` to edit snippets. You can set the buffer to this mode with `M-x snippet-mode`. It provides reasonably useful syntax highlighting.

Two commands are defined in this mode:

- `M-x yas-load-snippet-buffer`

When editing a snippet, this loads the snippet into the correct mode and menu. Bound to `C-c C-c` by default while in `snippet-mode`.

- `M-x yas-tryout-snippet`

When editing a snippet, this opens a new empty buffer, sets it to the appropriate major mode and inserts the snippet there, so you can see what it looks like. This is bound to `C-c C-t` while in `snippet-mode`.

There are also *snippets for writing snippets*: `vars`, `$f` and `$m` :-).

# File content

A file defining a snippet generally contains the template to be expanded.

Optionally, if the file contains a line of `# --`, the lines above it count as comments, some of which can be *directives* (or meta data). Snippet directives look like `# property: value` and tweak certain snippets properties described below. If no `# --` is found, the whole file is considered the snippet template.

Here's a typical example:

```
# contributor: pluskid <pluskid@gmail.com>
# name: __...__
# --
__${init}__
```

Here's a list of currently supported directives:

## # key: snippet abbrev

This is the probably the most important directive, it's the abbreviation you type to expand a snippet just before hitting the key that runs [yas-expand](#). If you don't specify this the snippet will not be expandable through the trigger mechanism.

## # name: snippet name

This is a one-line description of the snippet. It will be displayed in the menu. It's a good idea to select a descriptive name for a snippet – especially distinguishable among similar snippets.

If you omit this name it will default to the file name the snippet was loaded from.

## # condition: snippet condition

This is a piece of Emacs-lisp code. If a snippet has a condition, then it will only be expanded when the condition code evaluate to some non-nil value.

See also [yas-buffer-local-condition](#) in [Expanding snippets](#)

## # group: snippet menu grouping

When expanding/visiting snippets from the menu-bar menu, snippets for a given mode can be grouped into sub-menus . This is useful if one has too many snippets for a mode which will make the menu too long.

The `# group:` property only affect menu construction (See [the YASnippet menu](#)) and the same effect can be achieved by grouping snippets into sub-directories and using the `.yas-make-groups` special file (for this see [Organizing Snippets](#))

Refer to the bundled snippets for ruby-mode for examples on the `# group: directive`. Group can also be nested, e.g. `control structure.loops` tells that the snippet is under the `loops` group which is under the `control structure` group.

## # expand-env: expand environment

This is another piece of Emacs-lisp code in the form of a `let varlist form`, i.e. a list of lists assigning values to variables. It can be used to override variable values while the snippet is being expanded.

Interesting variables to override are `vas-wrap-around-region` and `vas-indent-line` (see [Expanding Snippets](#)).

As an example, you might normally have `yas-indent-line` set to 'auto and `yas-wrap-around-region` set to t, but for this particularly brilliant piece of ASCII art these values would mess up your hard work. You can then use:

```
# name: ASCII home
# expand-env: ((yas-indent-line 'fixed) (yas-wrap-around-region 'nil))
# --
```

## # binding: direct keybinding

You can use this directive to expand a snippet directly from a normal Emacs keybinding. The keybinding will be registered in the Emacs keymap named after the major mode the snippet is active for.

Additionally a variable `yas-prefix` is set to to the prefix argument you normally use for a command. This allows for small variations on the same snippet, for example in this "html-mode" snippet.

```
# name: <p>...</p>
# binding: C-c C-c C-m
# --
<p>`(when yas-prefix "\n")`$0`(when yas-prefix "\n")`</p>
```

This binding will be recorded in the keymap `html-mode-map`. To expand a paragraph tag newlines, just press `C-u C-c C-c C-m`. Omitting the `C-u` will expand the paragraph tag without newlines.

## # type: snippet or command

If the `type` directive is set to `command`, the body of the snippet is interpreted as lisp code to be evaluated when the snippet is triggered.

If it's `snippet` (the default when there is no `type` directive), the snippet body will be parsed according to the [Template Syntax](#), described below.

## # uuid: unique identifier

This provides to a way to identify a snippet, independent of its name. Loading a second snippet file with the same `uuid` would replace the previous snippet.

## # contributor: snippet author

This is optional and has no effect whatsoever on snippet functionality, but it looks nice.

# Template Syntax

The syntax of the snippet template is simple but powerful, very similar to TextMate's.

## Plain Text

Arbitrary text can be included as the content of a template. They are usually interpreted as plain text, except `$` and ```. You need to use `\` to escape them: `\$` and `\``. The `\` itself may also needed to be escaped as `\\` sometimes.

## Embedded Emacs-lisp code

Emacs-Lisp code can be embedded inside the template, written inside back-quotes (`). The lisp forms are evaluated when the snippet is being expanded. The evaluation is done in the same buffer as the snippet being expanded.

Here's an example for c-mode to calculate the header file guard dynamically:

```
#ifndef ${1:_(upcase (file-name-nondirectory (file-name-sans-extension (buffer-file-name))))}_H_}
#define $1

$0

#endif /* $1 */
```

From version 0.6, snippets expansions are run with some special Emacs-lisp variables bound. One of this is [yas-selected-text](#). You can therefore define a snippet like:

```
for ($1;$2;$3) {
  `yas-selected-text`$0
}
```

to "wrap" the selected region inside your recently inserted snippet. Alternatively, you can also customize the variable [yas-wrap-around-region](#) to t which will do this automatically.

## Tab stop fields

Tab stops are fields that you can navigate back and forth by TAB and S-TAB. They are written by \$ followed with a number. \$0 has the special meaning of the *exit point* of a snippet. That is the last place to go when you've traveled all the fields. Here's a typical example:

```
<div$1>
  $0
</div>
```

## Placeholder fields

Tab stops can have default values – a.k.a placeholders. The syntax is like this:

```
${N:default value}
```

They acts as the default value for a tab stop. But when you firstly type at a tab stop, the default value will be replaced by your typing. The number can be omitted if you don't want to create [mirrors](#) or [transformations](#) for this field.

## Mirrors

We refer the tab stops with placeholders as a *field*. A field can have mirrors. Its mirrors will get updated when you change the text of a field. Here's an example:

```
\begin{${1:enumerate}}
  $0
\end{${1}}
```

When you type "document" at `${1:enumerate}`, the word "document" will also be inserted at `\end{${1}}`. The best explanation is to see the screencast([YouTube](#) or [avi video](#)).

The tab stops with the same number to the field act as its mirrors. If none of the tab stops has an initial value, the first one is selected as the field and others mirrors.

## Mirrors with transformations

If the value of an `${n:-construct}` starts with and contains `$ (`, then it is interpreted as a mirror for field `n` with a transformation. The mirror's text content is calculated according to this transformation, which is Emacs-lisp code that gets evaluated in an environment where the variable [yas-text](#) is bound to the text content (string) contained in the field `n`. Here's an example for Objective-C:

```
- (${1:id})${2:foo}
{
    return $2;
}

- (void)set${2:$(capitalize yas-text)}:($1)aValue
{
    [$2 autorelease];
    $2 = [aValue retain];
}
$0
```

Look at `${2:$(capitalize yas-text)}`, it is a mirror with transformation instead of a field. The actual field is at the first line: `${2:foo}`. When you type text in `${2:foo}`, the transformation will be evaluated and the result will be placed there as the transformed text. So in this example, if you type "baz" in the field, the transformed text will be "Baz". This example is also available in the screencast.

Another example is for `rst-mode`. In `reStructuredText`, the document title can be some text surrounded by "=" below and above. The "=" should be at least as long as the text. So

```
====
Title
====
```

is a valid title but

```
===
Title
===
```

is not. Here's an snippet for `rst` title:

```

${1:$(make-string (string-width yas-text) ?\=)}
${1:Title}
${1:$(make-string (string-width yas-text) ?\=)}

$0

```

## Fields with transformations

From version 0.6 on, you can also have lisp transformation inside fields. These work mostly mirror transformations but are evaluated when you first enter the field, after each change you make to the field and also just before you exit the field.

The syntax is also a tiny bit different, so that the parser can distinguish between fields and mirrors. In the following example

```
#define "${1:mydefine$(upcase yas-text)}"
```

`mydefine` gets automatically upcased to `MYDEFINE` once you enter the field. As you type text, it gets filtered through the transformation every time.



Note that to tell this kind of expression from a mirror with a transformation, YASnippet needs extra text between the `:` and the transformation's `$`. If you don't want this extra-text, you can use two `$`'s instead.

```
#define "${1:$(uppercase yas-text)}"
```

Please note that as soon as a transformation takes place, it changes the value of the field and sets its internal modification state to `true`. As a consequence, the auto-deletion behaviour of normal fields does not take place. This is by design.

## Choosing fields value from a list and other tricks

As mentioned, the field transformation is invoked just after you enter the field, and with some useful variables bound, notably [yas-modified-p](#) and [yas-moving-away-p](#). Because of this feature you can place a transformation in the primary field that lets you select default values for it.

The [yas-choose-value](#) does this work for you. For example:

```
<div align="${2:$(yas-choose-value '("right" "center" "left"))}">
  $0
</div>
```

See the definition of [yas-choose-value](#) to see how it was written using the two variables.

Here's another use, for LaTeX-mode, which calls `reflex-label` just as you enter snippet field 2. This one makes use of [yas-modified-p](#) directly.

```
\section{${1:"Titel der Tour"}}%
\index{${1}}%
\label{{2:"waiting for reflex-label call..."$(unless yas-modified-p (reflex-label nil 'dont-
insert))}}%
```

The function [yas-verify-value](#) has another neat trick, and makes use of [yas-moving-away-p](#). Try it and see! Also, check out this [thread](#)

## Nested placeholder fields

From version 0.6 on, you can also have nested placeholders of the type:

```
<div${1: id="${2:some_id}"}>$0</div>
```

This allows you to choose if you want to give this `div` an `id` attribute. If you tab forward after expanding it will let you change `"some\_id"` to whatever you like. Alternatively, you can just press `C-d` (which executes [yas-skip-and-clear-or-delete-char](#)) and go straight to the exit marker.

By the way, `C-d` will only clear the field if you cursor is at the beginning of the field *and* it hasn't been changed yet. Otherwise, it performs the normal Emacs `delete-char` command.

---

Generated by [Emacs](#) 24.5.1 ([Org](#) mode 8.2.10) on from df229b9ab8db87fe5a1133365fdc299a65f9be86

[Validate](#)