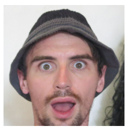
SEPTEMBER 24, 2020 / #VSCODE

Visual Studio Code Snippets – the Definitive VS Code Snippet Guide for Beginners



Rob O'Leary

Snippets can add a touch of magic to your editor. It's like an incantation. Utter a short phrase (type a prefix), wave your wand (press `Enter` or `Tab`), and presto! A wonderful event unfolds in front of you. ✨

Most code editors support snippets out of the box. The code editor I will use to showcase snippets is Visual Studio Code (VS Code), the most popular editor of the day.

Also, there are some "text expander" apps that allow you to use snippets globally (across all apps). I will briefly cover how you can utilise these apps to get even more out of snippets.

Let's dive into all things snippets. ?

Definition

> *A snippet is a template that can be inserted into a document. It is*

commonly used blocks of text. The general idea is to save you typing out the same things completely again and again, and again. ?

Why should you use Snippets?

I'm not going to shock you with this statement: the internet is home to a lot conflicting opinions! Snippets do not escape this ignominy, but I don't think it is a topic that make people's blood pressure soar!

For the sake of balance, I will present a cross-section of those opinions here.

You don't have to pick a camp and be all-for or all-against snippets. I suggest you adopt them to a degree that serves you best.

Yay Camp ?

- Snippets can boost your productivity, saving you keystrokes and reducing input errors.
- Snippets leaves me with more mental CPU and enjoyment to spend on writing the code that I care about and want to focus on.
- Snippets can help you to remember to include something important!
- Integrating snippets into your workflow implicitly encourages you to use the mouse less often. Well-written snippets offer a logical path which you can tab through, stopping to edit along the way to complete the template exactly the way you want, and when you are done, you arrive on the other side ready to write your next line

reliant on any given tool.

- I never use snippets. I prefer to invest time in avoiding repetition rather than making it easier.
- I found that at some point I forgot how to write the code without using the snippet. For trivial stuff that I understand, it is OK, but I don't want to forget some other stuff!
- Most, if not all, snippets I've seen online, for code I'm looking for, have mistakes in them. I've not once been able to find a numerical algorithm that didn't have floating point errors in it. I can't imagine there being any resource of perfectly clean code snippets.

When should you use snippets?

Donald Knuth, one of the grand-wizards of computer science, said "premature optimization [of code] is the root of all evil".

I think this is relevant to snippets also. Snippets are an optimization of code production. If you don't know a language or framework very well, implementing a slew of snippets for that language or framework is likely to be a premature move.

If you are comfortable, then try some out!

What I use snippets for

Personally, I use snippets often but judiciously. I use a set of snippets for Markdown and most of the languages I work with.

I haven't used snippets much for frameworks. I started using some

I haven't used snippets for algorithms.

Types of Snippets

Snippets can be classified according to the scope of interactivity between the snippet and the editor.

Static Snippets

You can think of it as a copy-and-paste of some source text as a single command.

Dynamic Snippets

A dynamic snippet can be customised to provide a wizard-like experience for completion of a snippet.

It can include:

- *Tab Stops*: You can number stops that can be tabbed through in order,
- *Mirrored Tab Stops*: There are times when you need to provide the same value in several places in the inserted text. You can mirror a tab stop to achieve this, and any edit will be reflected in the related tab stops instantly.
- *Placeholders* : It is a tab stop with a default value which can be overwritten on focus.
- *Choices* : At a tab stop you are presented with a dropdown list of values to choose from.
- *Variables*: Input values from the environment such as: the selected text in the editor, system dates, or content from the

with 2 tasks. It uses *tab stops*, *placeholders*, and *choices* for checking a task.

Macro Snippets

The top level of sorcery is to have the capability to transform input. Transformations allow you to alter the value of a variable before they are inserted, or alter a placeholder after you have made an edit.

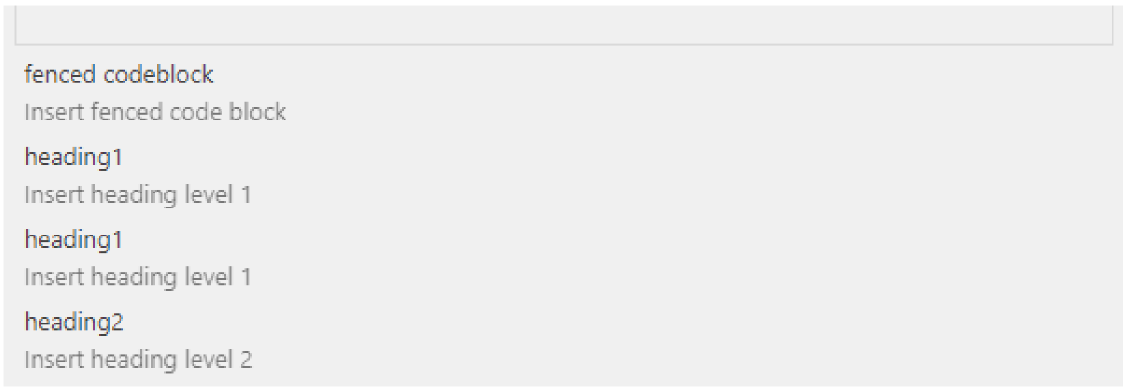
For example, you may want to capitalise a class name once it is entered.

Anything that you can think of doing with a regex is typically possible. Some editors offer more advanced scripting possibilities.

Snippets in Visual Studio Code

In VS Code, snippets appear in **IntelliSense** (`Ctrl+Space` gives you a suggestion list), they are mixed in with other suggestions.

You can also access them in a dedicated snippet picker by using the **'Insert Snippet' command**. This combines all user, extension, and



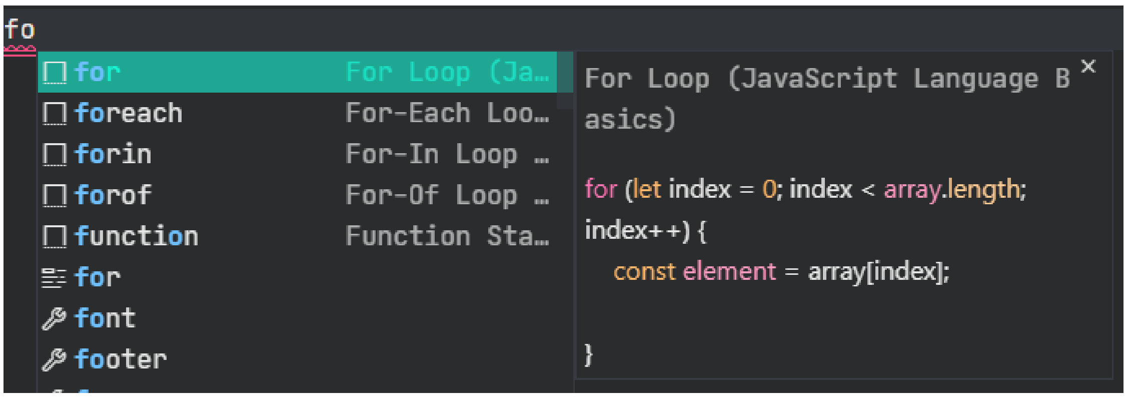
```
fenced codeblock
Insert fenced code block
heading1
Insert heading level 1
heading1
Insert heading level 1
heading2
Insert heading level 2
```

[Emmet](https://emmet.io/) is integrated into VS Code and has it's own CSS-selector inspired syntax for inserting HTML and CSS snippets.

Emmet is it's own thing really, but the mechanics are the same. You can learn about Emmet with the Emmet in Visual Studio Code guide.

Related User Settings

Snippets will appear as **quick suggestions** if the setting `editor.quickSuggestions` is set to true for the language you are working in. Quick suggestions are enabled by default for most languages except markdown.



```
fo
for         For Loop (Ja…     For Loop (JavaScript Language B
foreach     For-Each Loo…     asics)
forin       For-In Loop …
forof       For-Of Loop …     for (let index = 0; index < array.length;
function    Function Sta…     index++) {
for                              const element = array[index];
font
footer                        }
```

Snippets support **tab-completion**. You can type a snippet prefix (the

- `on` : Tab completion is enabled for all sources.
- `off` : Disable tab completions. This is the *default value*.
- `onlySnippets` : Tab completion only for snippets.

```
"editor.tabCompletion": "onlySnippets",
```

If you would like to control how **snippets suggestions** are shown, you can edit the setting `editor.snippetSuggestions` .

The values are:

- `top` : Show snippet suggestions on top of other suggestions. I use this value.
- `bottom` : Show snippet suggestions below other suggestions.
- `inline` : Show snippets suggestions with other suggestions. This is the *default value*.
- `none` : Do not show snippet suggestions.

```
"editor.snippetSuggestions": "top",
```

These are the most important settings for snippets, but there are a few more. You can check out this list of the default settings to explore more, or do a search in the Settings UI.

They aren't documented in the VS Code docs, though. And inside VS Code, there is no central point to browse them. So, you may not know what they are.

So, how can you find out what languages have built-in snippets?

Long story short, I was frustrated by this scenario, so I wrote an extension called **Snippets Ranger** to give a nice UI to explore snippets easily. Think of it as a *Marauder's Map* for snippets!

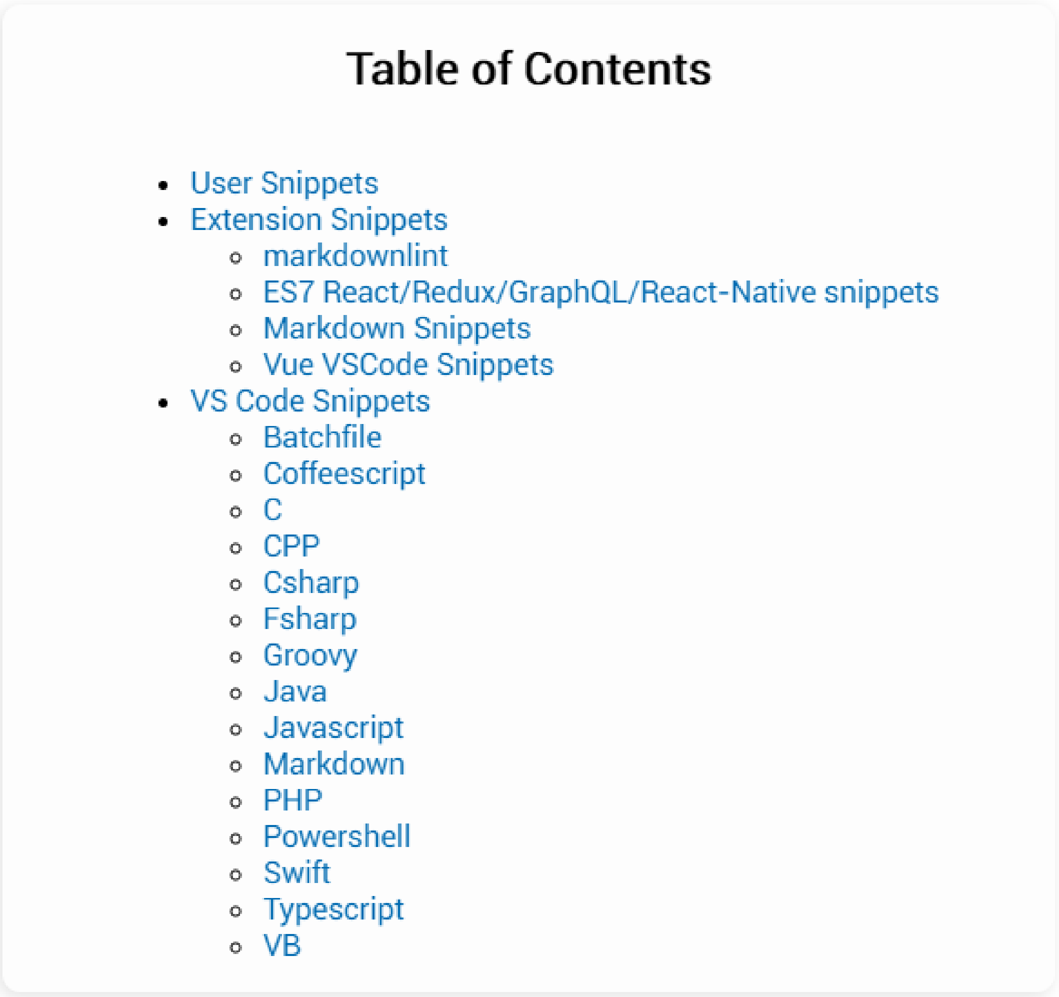


Table of Contents

- User Snippets
- Extension Snippets
  - markdownlint
  - ES7 React/Redux/GraphQL/React-Native snippets
  - Markdown Snippets
  - Vue VSCode Snippets
- VS Code Snippets
  - Batchfile
  - Coffeescript
  - C
  - CPP
  - Csharp
  - Fsharp
  - Groovy
  - Java
  - Javascript
  - Markdown
  - PHP
  - Powershell
  - Swift
  - Typescript
  - VB

As I mentioned earlier, the **Insert Snippet** command will show you all snippets for the language of the active document.

Remember though, this is an *aggregate* of all of the user, extension, and built-in snippets. So, if you want to find out if a particular language has built-in snippets, you need to open a file for that language, and run the command to see that list.

If you have an snippets extension installed for that language that makes it too hard to identify which is which, you could disable it to ensure that only the built-in snippets are showing. ?

If you want to track down the source file yourself, the built-in snippets live inside each individual language extension directory. The file is located at `«app root»\resources\app\extensions\«language»\snippets\«language».code-snippets` on Windows. The location is similar for Mac and Linux.

Snippets Extensions

The Visual Studio Marketplace has a snippets category where you can find snippets for almost anything.

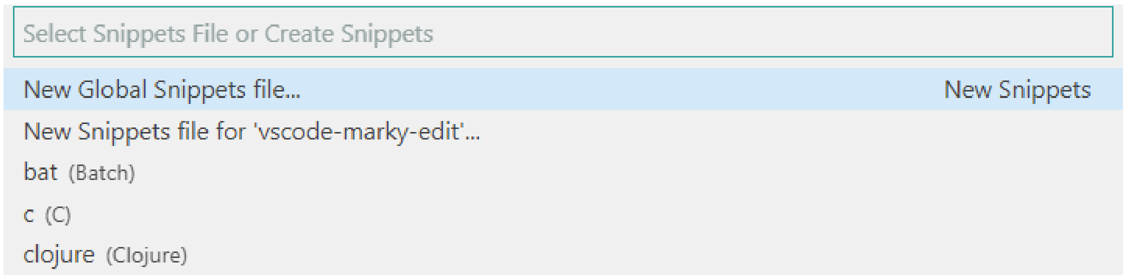
A lot of Programming Language Pack extensions include snippets also (Python, C#, Go, Java, and C/C++ amongst others).

How do I write my own?

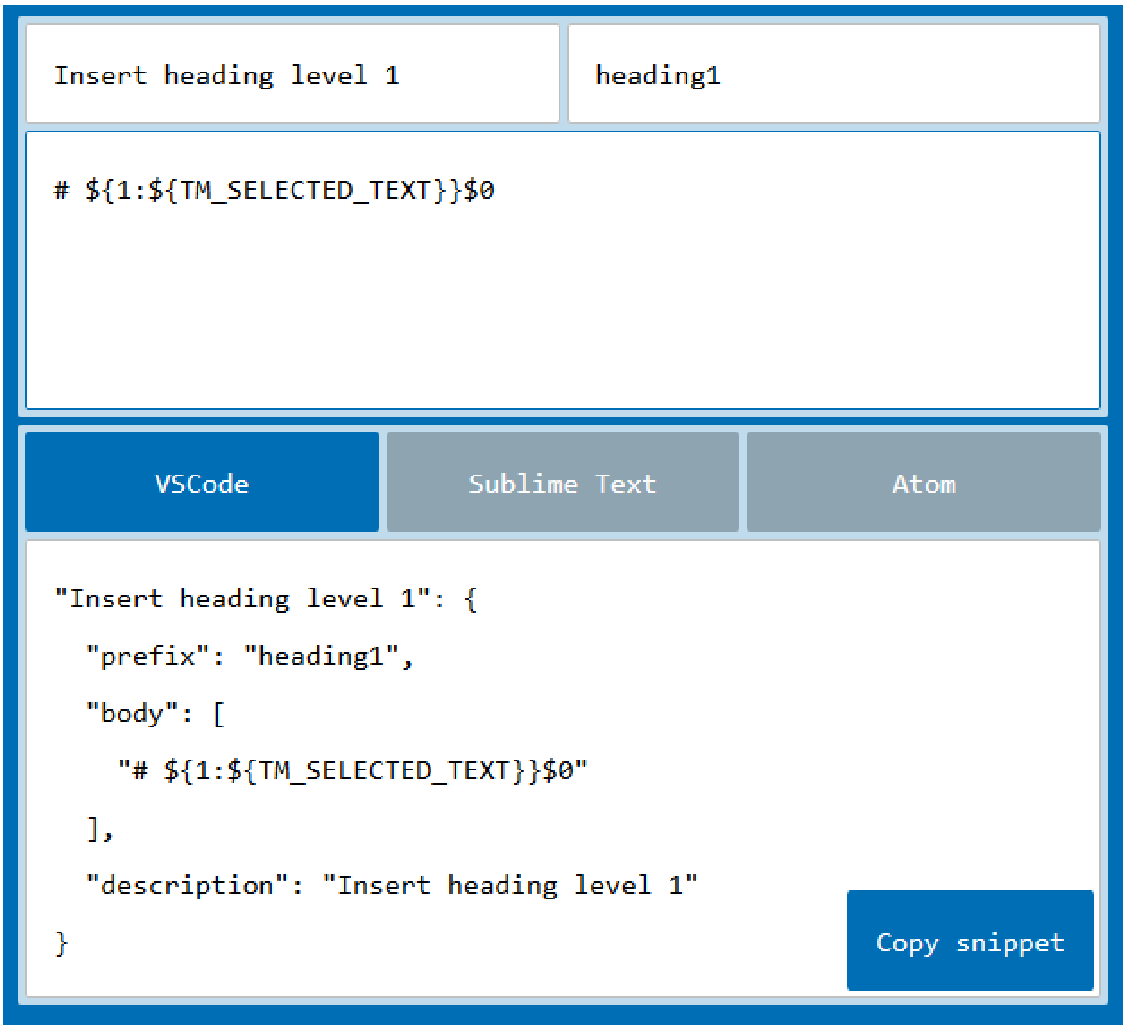
Snippets files are written in JSON. You can also add C-style comments if you wish (technically it is Microsoft's "JSONC" format).

You can create snippets for different scopes: global, workspace, and for a particular language.

selection will open a file for editing.


Select Snippets File or Create Snippets
New Global Snippets file... New Snippets
New Snippets file for 'vscode-marky-edit'...
bat (Batch)
c (C)
clojure (Clojure)

If you would prefer to write a snippet in a GUI, you can use the snippet generator web app.



```
# ${1:${TM_SELECTED_TEXT}}$0
```

```
"Insert heading level 1": {
  "prefix": "heading1",
  "body": [
    "# ${1:${TM_SELECTED_TEXT}}$0"
  ],
  "description": "Insert heading level 1"
}
```

Example

Here is a markdown snippet that comes with VS Code.

```
{
    "Insert heading level 1": {
       "prefix": "heading1",
       "body": ["# ${1:${TM_SELECTED_TEXT}}$0"],
       "description" : "Insert heading level 1"
    }
}
```

This snippet inserts a level 1 heading which wraps the markdown around the current selection (if there is one).

A snippet has the following properties:

1. "Insert heading level 1" is the snippet name. This is the value that is displayed in the IntelliSense suggestion list if no `description` is provided.
2. The `prefix` property defines the trigger phrase for the snippet. It can be a string or an array of strings (if you want multiple trigger phrases). Substring matching is performed on prefixes, so in this case, typing "h1" would match our example snippet.
3. The `body` property is the content that is inserted into the editor. It is an array of strings, which is one or more lines of content. The content is joined together before insertion.
4. The `description` property can provide more information about the snippet. It is optional.

snippet file.

The body of this snippet has 2 tab stops and uses the variable `${TM_SELECTED_TEXT}`.

Let's get into the syntax to understand this fully.

Snippet syntax

VS Code's snippet syntax is the same as the TextMate snippet syntax. However, it does not support 'interpolated shell code' and the use of the `\u` transformation.

The `body` of a snippet supports the following features:

1. Tab Stops

Tab stops are specified by a dollar sign and an ordinal number e.g. `$1` . `$1` will be the first location, `$2` will the second location, and so on. `$0` is the final cursor position, which exits the snippet mode.

For example, let's say we want to make an HTML *div* snippet and we want the first tab stop to be between the opening and closing tags. We also want to allow the user to tab outside of the tags to finish the snippet.

Then we could make a snippet like this:

```
{
    "Insert div": {
        prefix: "div",
        body: ["<div>","$1","</div>", "$0"]
    }
}
```

2. Mirrored Tab Stops

There are times when you need to provide the same value in several places in the inserted text. In these situations you can re-use the same ordinal number for tab stops to signal that you want them mirrored. Then your edits are synced.

A typical example is a *for* loop which uses an *index* variable multiple times. Below is a JavaScript example of a *for* loop.

```
{
    "For Loop": {
        "prefix": "for",
        "body": [
            "for (let ${1:index} = 0; ${1:index} < ${2:array}.le
            "\tconst ${3:element} = ${2:array}[${1:index}];",
            "\t$0",
            "}"
        ]
    }
}
```

3. Placeholders

Placeholders are tab stops with default values. They are wrapped in curly braces, for example `${1:default}` . The placeholder text is selected on focus such that it can be easily edited. Placeholders can be nested, like this: `${1:first ${2:second}}` .

4. Choices

Choices present the user with a list of values at a tab stop. They are written as a comma-separated list of values enclosed in pipe-characters e.g. `${1|yes,no|}` .

```
{
  "Insert task list": {
    "prefix": "task",
    "body": ["- [${1| ,x|}] ${2:text}", "${0}"]
}
```

5. Variables

There is a good selection of variables you can use. You simply prefix the name with a dollar sign to use them, for example `$TM_SELECTED_TEXT`.

For example, this snippet will create a block comment for any language with today's date:

```
{
    "Insert block comment with date": {
        prefix: "date comment",
        body: ["${BLOCK_COMMENT_START}",
               "${CURRENT_YEAR}/${CURRENT_MONTH}/${CURRENT_DATE}
               "${BLOCK_COMMENT_END}"]
    }
}
```

You can specify a default for a variable if you wish, like `${TM_SELECTED_TEXT:default}`. If a variable does not have a value assigned, the default or an empty string is inserted.

If you make a mistake and include a variable name that is not

- `TM_SELECTED_TEXT` : The currently selected text or the empty string,
- `TM_CURRENT_LINE` : The contents of the current line,
- `TM_CURRENT_WORD` : The contents of the word under cursor or the empty string,
- `TM_LINE_INDEX` : The zero-index based line number,
- `TM_LINE_NUMBER` : The one-index based line number,
- `TM_FILENAME` : The filename of the current document,
- `TM_FILENAME_BASE` : The filename of the current document without its extensions,
- `TM_DIRECTORY` : The directory of the current document,
- `TM_FILEPATH` : The full file path of the current document,
- `CLIPBOARD` : The contents of your clipboard,
- `WORKSPACE_NAME` : The name of the opened workspace or folder.

The following time-related variables can be used:

- `CURRENT_YEAR` : The current year,
- `CURRENT_YEAR_SHORT` : The current year's last two digits,
- `CURRENT_MONTH` : The month as two digits (example '07'),
- `CURRENT_MONTH_NAME` : The full name of the month (example 'July'),
- `CURRENT_MONTH_NAME_SHORT` : The short name of the month (example 'Jul'),
- `CURRENT_DATE` : The day of the month,

(example 'Mon'),

- `CURRENT_HOUR` : The current hour in 24-hour clock format,
- `CURRENT_MINUTE` : The current minute,
- `CURRENT_SECOND` : The current second,
- `CURRENT_SECONDS_UNIX` : The number of seconds since the Unix epoch.

The following comment variables can be used. They honour the syntax of the document's language:

- `BLOCK_COMMENT_START` : For example, `<!--` in HTML,
- `BLOCK_COMMENT_END` : For example , `-->` in HTML,
- `LINE_COMMENT` : For example, `//` in JavaScript.

6. Transformations

Transformations can be applied to a variable or a placeholder. If you are familiar with regular expressions (regex), most of this should be familiar.

The format of a transformation is: `${«variable or placeholder»/«regex»/«replacement string»/«flags»}` . It is similar to String.protoype.replace() in JavaScript. The "parameters" do the following:

- `«regex»` : This is a regular expression that is matched against the value of the variable or placeholder. The JavaScript regex syntax is supported.
- `«replacement string»` : This is the string you want to replace the original text with. It can reference capture

/capitalize), and perform conditional insertions. See TextMate Replacement String Syntax for more in-depth information.

- «flags» : Flags that are passed to the regular expression. The JavaScript regex flags can be used:
  - g : Global search,
  - i : Case-insensitive search,
  - m : Multi-line search,
  - s : Allows . to match newline characters,
  - u : Unicode. Treat the pattern as a sequence of Unicode code points,
  - y : Perform a "sticky" search that matches starting at the current position in the target string.

To reference a capture group, use `$n` where `n` is the capture group number. Using `$0` means the entire match.

This can be a bit confusing since tab stops have the same syntax. Just remember that if it is contained within forward slashes, then it is referencing a capture group.

The easiest way to understand the syntax fully is to check out a few examples.

SNIPPET *BODY*	INPUT
`["${TM_SELECTED_TEXT/^.+$/• $0/gm}"]`	line1 line2
`["${TM_SELECTED_TEXT/^(\\w+)/${1:/capitalize}/}"]`	the cat is on the ma

`[`
`"[",`
`"${CLIPBOARD/^(.+)$/'$1',/gm}",`
`"]"`
`]`
	line1
	line2

As you can see from the second example above, metacharacter sequences must be escaped, for example insert `\\w` for a word character.

Placeholder Transformations

Placeholder transforms do <u>not</u> allow a default value or choices! Maybe it is more suitable to call them tab stop transformations.

The example below will uppercase the text of the first tab stop.

```
{
  "Uppercase first tab stop": {
```

You can have a placeholder and perform a transformation on a mirrored instance. The transformation will not be performed on the initial placeholder. ?

Would you use this behaviour somewhere? I find it confusing initially, so it may have the same affect on others.

```
{
  "Uppercase second tab stop instance only": {
    "prefix": "up",
    "body": ["${1:title}", "${1/(.*)/${1:/upcase}/}", "$0"]
  }
}
```

How do I assign Keyboard Shortcuts for snippets?

By adding your shortcuts to `keybindings.json` . You can open the file by running the **'Preferences: Open Keyboard Shortcuts File (JSON)'** command.

For example, to add a shortcut for the built-in markdown snippet "Insert heading level 1":

```
{
    "key": "ctrl+m ctrl+1",
    "command": "editor.action.insertSnippet",
    "when": "editorTextFocus && editorLangId == markdown",
    "args": {
        "langId": "markdown",
```

You define a shortcut by specifying the key combination you want to use, the command ID, and an optional when clause context for the context when the keyboard shortcut is enabled.

Through the `args` object, you can target an existing snippet by using the `langId` and `name` properties. The `langId` argument is the language ID of the language that the snippets were written for. The `name` is the snippet's name as it is defined in the snippet file.

You can define an inline snippet if you wish using the `snippet` property.

```
[
  {
    "key": "ctrl+k 1",
    "command": "editor.action.insertSnippet",
    "when": "editorTextFocus",
    "args": {
      "snippet": "${BLOCK_COMMENT_START}${CURRENT_YEAR}/${CURREN
    }
  }
]
```

You can use the *Keyboard Shortcuts UI* also, but it does not have the ability to add a new shortcut.

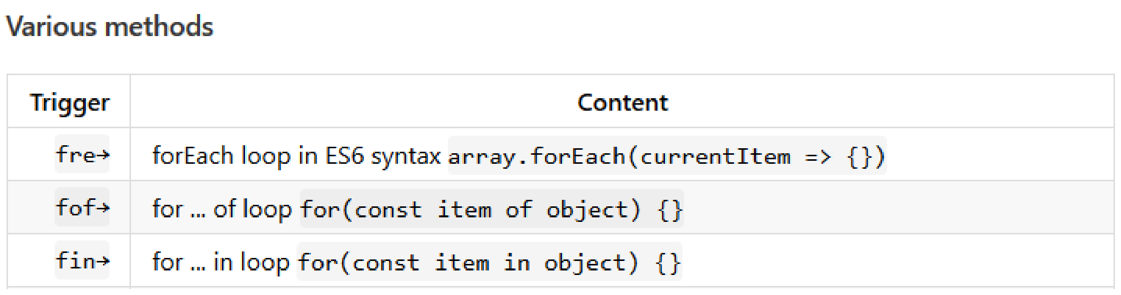
Another downside of the UI is that it does not show the `args` object, which makes it more difficult to find and edit your custom shortcuts. ?

editor.action.insertSnippet
Insert Snippet editor.action.insertSnippet Ctrl + K 1 editorTextFocus && editorLangId == 'markdown' User

A question of style

Something that I found offputting initially with snippets was the propensity for people to create snippets with abbreviated prefixes. Do I have to learn a big list of gibberish acronyms to use someone else's snippets?

What do I mean by abbreviated prefixes? The table below list a few of the snippets from the JavaScript (ES6) code snippets VS Code extension. You can see in the *Trigger* column, the prefixes listed are abbreviations, for example *fre* to represent a "for each" loop.



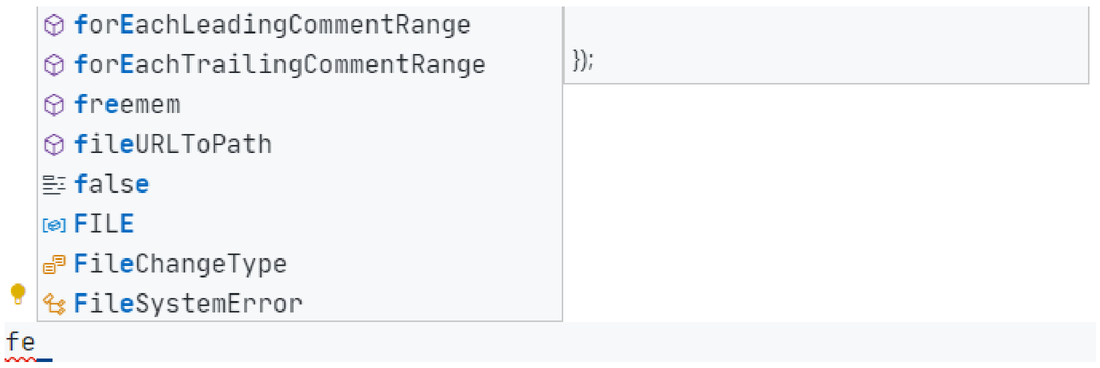
Various methods

Trigger	Content
`fre→`	forEach loop in ES6 syntax `array.forEach(currentItem => {})`
`fof→`	for ... of loop `for(const item of object) {}`
`fin→`	for ... in loop `for(const item in object) {}`

This is unnecessary in two ways.

Firstly, the quick suggestions offered by VS Code are produced from a **fuzzy substring search**. If I type "fe" and the prefix of a snippet is "foreach", this will match and be offered as a quick suggestion.

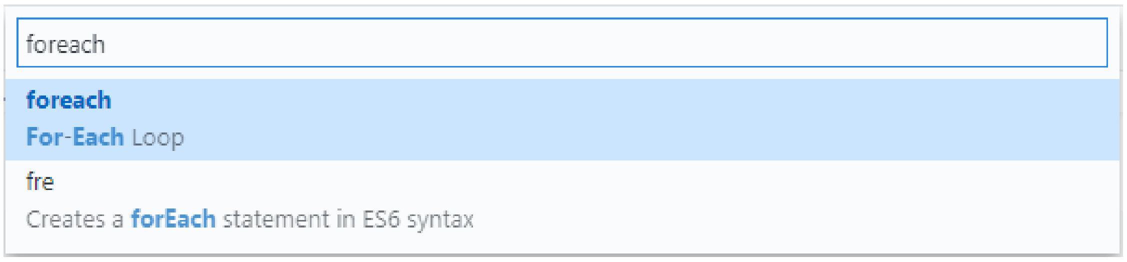
As you can see below, this is the second match.



```
forEachLeadingCommentRange
forEachTrailingCommentRange
freemem
fileURLToPath
false
FILE
FileChangeType
FileSystemError
fe
```

The first match is *fre*, which is a snippet from the extension. Which suggestion is more descriptive? ?

If you use the "Insert Snippet" command for snippets, it does not make much of a difference. The description field makes amends for this shortcoming. I don't use snippets in this way, so I would prefer a more descriptive prefix.



```
foreach
foreach
For-Each Loop
fre
Creates a forEach statement in ES6 syntax
```

Secondly, *fre* is a **duplicate** of the built-in snippet *foreach*.

Some people turn off quick suggestions for snippets and use tab completion only. In this case, you need to type a prefix out without getting visual feedback. Some people may prefer to use an abbreviated prefix to save keystrokes here.

The same fuzzy substring search is being performed in the background, so the first snippet match is inserted when you hit tab.

Looking at the example above, you can see that typing "fr" and hitting *tab* inserts the *fre* snippet. Typing "fore" and hitting tab inserts the *foreach* snippet.

So, you do not need to type out the entire prefix, if you really don't want to! ? If you have many similarly named snippet prefixes for a language, it would be impractical I imagine.

It is more practical to learn the prefixes properly, and type them out entirely before hitting tab.

There are some trade-offs depending on your preferences for using snippets.

Personally, I like to use quick suggestions as I like the visual feedback. I have snippets set to be the top suggestions, that way I can type abbreviated versions of the prefixes without needing to memorise them.

Some snippet authors have rigid patterns to overcome this, but that's just something I can't get into easily.

If you use snippets for different frameworks and libraries in a language, they can add up and overlap. I haven't needed to do this, but you may need to do it eventually.

Global Snippets

Outside of your code editor, you can benefit from snippets also. Having snippets available in every app offers more possibilities.

Common use cases are:

- canned responses for messages,
- autocorrecting common typos,
- adding contact information or signatures to documents,
- inserting dates,
- formatting of selected text and pasted text,
- inserting search phrases for your search engine or app,
- HTML snippets available inside your email client,
- adding different templates to documents.

Most of the apps for snippets are touted as "text expanders", but quite a few task and productivity apps also include snippet-esque features.

Global snippets are **bit more limited that code editor snippets**, as you cannot use tab stops and placeholders. In most apps you can use some dynamic variables such as dates.

[AutoHotkey](#) is a **free, open-source scripting language for Windows** to do all kinds of tasks.

It has it's own unique syntax. You can install the AutoHotKey extension to add language support to VS Code for a better editing experience.

For defining prefixes to trigger a snippet insertion, you use the following format: `::<<prefix>>::<<text to insert>>`. The following script will insert Rob's email address when you type "robmail" and hit *space* or *tab* or *enter*.

```
::robmail::rob@someservername.com
```

The following script will insert the text "This is the snippet text" when you press `Ctrl+D`.

```
^d::  Send This is the snippet text
```

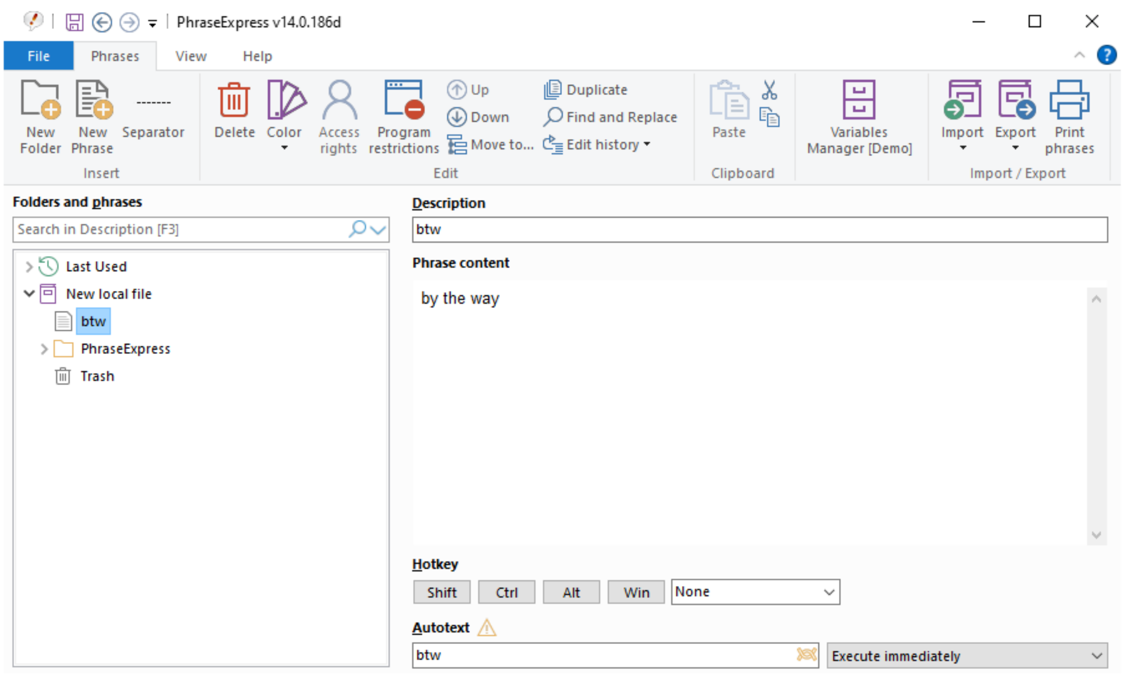
You can read the docs to learn more.

PhraseExpress (Windows, Mac, iOS)

PhraseExpress is "a text expander software, that manages frequently used text templates for insertion into any program".

It is a **freemium, GUI-based app**. It is aimed at a wider audience it will be active in the background.

Your snippets can be organized into custom folders and synced using cloud services.



Description: btw

Phrase content: by the way

Hotkey: None

Autotext: btw — Execute immediately

Espanso (Windows, Mac, Linux)

This is a **open-source, cross-platform text expander written in Rust.**

It uses a **file-based configuration approach**. The config files are written in YAML.

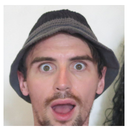
The `default.yml` file contains the main configuration. The config below will insert Rob's email address when you type "robmail" .

```
matches:
  - trigger: ":robmail"
```

You can **specify the initial cursor position**, however you cannot define tab stops.

You can **add extensions** to increase the capability of Espanso. There are extensions for running external scripts, including dates, generating random text, and including clipboard data.

And that's about it! I hope you learned something about snippets today, and you can use them to make yourself more productive.



Rob O'Leary

Read more posts.

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

Forum Donate

Learn to code — free 3,000-hour curriculum

Trending Guides

Python Parse JSON	HTML Video
HTML Cheat Sheet	CSS Overflow
ROW_NUMBER in SQL	What is REST?
HTML Center Image	VGA No Signal
JavaScript Replace	CSS Text Align
HTML HR Tag Example	HTML Label Tag
JavaScript UpperCase	CSS Button Style
Ordered List in HTML	For Loop in Java
Screenshot on Windows	Linux cp Command
Sort an Array in Java	Link CSS to HTML
Cast a Function in SQL	Declare an Array in Java
Java Logical Operators	Get Current Time in Python
Chmod Command in Linux	Change Div Background Color
Python Get Last Element	Get Variable Type in Python
Python Add to Dictionary	Logical Operators in Python

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty

Code of Conduct Privacy Policy Terms of Service Copyright Policy