Search (S)

Overview
Tutorial
Reference
LANGUAGE
Syntax
Styling
Scripting
Context
LIBRARY
Foundations
Model
Text
Math
Symbols
Layout
Visualize
Introspection
Data Loading
Guides
Changelog
Roadmap
Community

Reference > Syntax

Syntax

Typst is a markup language. This means that you can use simple syntax to accomplish common layout tasks. The lightweight markup syntax is complemented by set and show rules, which let you style your document easily and automatically. All this is backed by a tightly integrated scripting language with built-in and user-defined functions.

Modes

Typst has three syntactical modes: Markup, math, and code. Markup mode is the default in a Typst document, math mode lets you write mathematical formulas, and code mode lets you use Typst's scripting features.

You can switch to a specific mode at any point by referring to the following table:

New mode	Syntax	Example
Code	Prefix the code with #	`Number: #(1 + 2)`
Math	Surround equation with `$..$`	`$-x$ is the opposite of $x$`
Markup	Surround markup with `[..]`	`let name = [*Typst!*]`

Once you have entered code mode with #, you don't need to use further hashes unless you switched back

to markup or math mode in between.

Markup

Typst provides built-in markup for the most common document elements. Most of the syntax elements are just shortcuts for a corresponding function. The table below lists all markup that is available and links to the best place to learn more about their syntax and usage.

Name	Example	See
Paragraph break	Blank line	parbreak
Strong emphasis	`*strong*`	strong
Emphasis	`_emphasis_`	emph
Raw text	`` `print(1)` ``	raw
Link	`https://typst.app/`	link
Label	`<intro>`	label
Reference	`@intro`	ref
Heading	`= Heading`	heading
Bullet list	`- item`	list
Numbered list	`+ item`	enum
Term list	`/ Term: description`	terms
Math	`$x^2$`	Math
Line break	`\`	linebreak
Smart quote	`'single' or "double"`	smartquote
Symbol shorthand	`~`, `---`	Symbols
Code expression	`#rect(width: 1cm)`	Scripting
Character escape	`Tweet at us \#ad`	Below
Comment	`/* block */`, `// line`	Below

Math mode

Math mode is a special markup mode that is used to typeset mathematical formulas. It is entered by wrapping an equation in $ characters. This works both in markup and code. The equation will be typeset into its own block if it starts and ends with at least one space (e.g. `$ x^2 $`). Inline math can be produced by omitting the whitespace (e.g. `$x^2$`). An overview over the syntax specific to math mode follows:

Name	Example	See
Inline math	`$x^2$`	Math
Block-level math	`$ x^2 $`	Math
Bottom attachment	`$x_1$`	attach
Top attachment	`$x^2$`	attach
Fraction	`$1 + (a+b)/5$`	frac
Line break	`$x \ y$`	linebreak
Alignment point	`$x &= 2 \ &= 3$`	Math
Variable access	`$#x$, $pi$`	Math
Field access	`$arrow.r.long$`	Scripting
Implied multiplication	`$x y$`	Math
Symbol shorthand	`$->$, $!=$`	Symbols
Text/string in math	`$a "is natural"$`	Math
Math function call	`$floor(x)$`	Math
Code expression	`$#rect(width: 1cm)$`	Scripting
Character escape	`$x\^2$`	Below
Comment	`$/* comment */$`	Below

Code mode

Within code blocks and expressions, new expressions

can start without a leading # character. Many syntactic elements are specific to expressions. Below is a table listing all syntax that is available in code mode:

Name	Example	See
None	`none`	none
Auto	`auto`	auto
Boolean	`false`, `true`	bool
Integer	`10`, `0xff`	int
Floating-point number	`3.14`, `1e5`	float
Length	`2pt`, `3mm`, `1em`, ..	length
Angle	`90deg`, `1rad`	angle
Fraction	`2fr`	fraction
Ratio	`50%`	ratio
String	`"hello"`	str
Label	`<intro>`	label
Math	`$x^2$`	Math
Raw text	`` `print(1)` ``	raw
Variable access	`x`	Scripting
Code block	`{ let x = 1; x + 2 }`	Scripting
Content block	`[*Hello*]`	Scripting
Parenthesized expression	`(1 + 2)`	Scripting
Array	`(1, 2, 3)`	Array
Dictionary	`(a: "hi", b: 2)`	Dictionary
Unary operator	`-x`	Scripting
Binary operator	`x + y`	Scripting
Assignment	`x = 1`	Scripting
Field access	`x.y`	Scripting
Method call	`x.flatten()`	Scripting
Function call	`min(x, y)`	Function
Argument spreading	`min(..nums)`	Arguments

Unnamed function	`(x, y) => x + y`	Function
Let binding	`let x = 1`	Scripting
Named function	`let f(x) = 2 * x`	Function
Set rule	`set text(14pt)`	Styling
Set-if rule	`set text(..) if ..`	Styling
Show-set rule	`show heading: set block(..)`	Styling
Show rule with function	`show raw: it => {..}`	Styling
Show-everything rule	`show: template`	Styling
Context expression	`context text.lang`	Context
Conditional	`if x == 1 {..} else {..}`	Scripting
For loop	`for x in (1, 2, 3) {..}`	Scripting
While loop	`while x < 10 {..}`	Scripting
Loop control flow	`break, continue`	Scripting
Return from function	`return x`	Function
Include module	`include "bar.typ"`	Scripting
Import module	`import "bar.typ"`	Scripting
Import items from module	`import "bar.typ": a, b, c`	Scripting
Comment	`/* block */, // line`	Below

Comments

Comments are ignored by Typst and will not be included in the output. This is useful to exclude old versions or to add annotations. To comment out a

single line, start it with `//`:

```
// our data barely supports
// this claim

We show with $p < 0.05$
that the difference is
significant.
```

We show with $p < 0.05$ that the difference is significant.

Comments can also be wrapped between `/*` and `*/`. In this case, the comment can span over multiple lines:

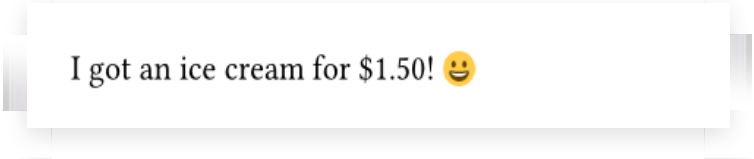
```
Our study design is as follows:
/* Somebody write this up:
   - 1000 participants.
   - 2x2 data design. */
```

Our study design is as follows:

Escape sequences

Escape sequences are used to insert special characters that are hard to type or otherwise have special meaning in Typst. To escape a character, precede it with a backslash. To insert any Unicode codepoint, you can write a hexadecimal escape sequence: `\u{1f600}`. The same kind of escape sequences also work in strings.

```
I got an ice cream for
\$1.50! \u{1f600}
```



Paths

Typst has various features that require a file path to reference external resources such as images, Typst files, or data files. Paths are represented as strings. There are two kinds of paths: Relative and absolute.

- A **relative path** searches from the location of the Typst file where the feature is invoked. It is the default:

  ```
  #image("images/logo.png")
  ```

- An **absolute path** searches from the *root* of the project. It starts with a leading `/`:

  ```
  #image("/assets/logo.png")
  ```

Project root

By default, the project root is the parent directory of the main Typst file. For security reasons, you cannot read any files outside of the root directory.

If you want to set a specific folder as the root of your project, you can use the CLI's `--root` flag. Make sure that the main file is contained in the folder's subtree!

```
typst compile --root .. file.typ
```

In the web app, the project itself is the root directory. You can always read all files within it, no matter which one is previewed (via the eye toggle next to each Typst file in the file panel).

Paths and packages

A package can only load files from its own directory. Within it, absolute paths point to the package root, rather than the project root. For this reason, it cannot directly load files from the project directory. If a package needs resources from the project (such as a logo image), you must pass the already loaded image, e.g. as a named parameter `logo: image("mylogo.svg")`. Note that you can then still customize the image's appearance with a set rule within the package.

In the future, paths might become a distinct type from strings, so that they can retain knowledge of where they were constructed. This way, resources could be loaded from a different root.

Reference
Previous page

Styling
Next page