Typst Examples Book

This book provides an extended *tutorial* and lots of snippets that can help you to write better Typst code.

Typst Basics

This is a chapter that consistently introduces you to the most things you need to know when writing with Typst.

It show much more things than official tutorial, so maybe it will be interesting to read for some of the experienced users too.

Some examples are taken from and . Most are created and edited specially for this book.

> *Important:* in most cases there will be used “clipped” examples of your > rendered documents (no margins, smaller width and so on). > > To set up the spacing as you want, see .

Tutorial by Examples

The first section of Typst Basics is very similar to , with more specialized examples and less words. It is *highly recommended to read the official tutorial anyway* .

Of course, you can use `set` rule with all built-in functions and all their named arguments to make some argument “default”.

For example, let’s make all quotes in this snippet authored by the book:

Opinionated defaults

That allows you to set Typst default styling as you want it: Don’t complain about bad defaults! `Set` your own.

Numbering

Of course, there are lots of other cool properties that can be *set* , so feel free to dive into and explore them!

And now we are moving into something much more interesting…

Advanced styling

== About syntax

Blocks

One of the most important usages is that you can set up all spacing using blocks. Like every element with text contains text that can be set up, every *block element* contains blocks:== Setting spacing

TODO: explain block spacing for common elements

Formatting to get an “article look”

Templates

Templates

If you want to reuse styling in other files, you can use the *template* idiom. Because `set` and `show` rules are only active in their current scope, they will not affect content in a file you imported your file into. But functions can circumvent this in a predictable way: This is equivalent to: Then in your main file: *This will apply a “template” function to the rest of your document!*

Passing arguments
Then, in template file: Writing templates is fairly easy if you understand .

See more information about writing templates in .

There is no official repository for templates yet, but there are a plenty community ones in .

Must-know
This section contains things, that are not general enough to be part of “tutorial”, but still are very important to know for proper typesetting.

Feel free to skip through things you are sure you will not use.

Boxing & Blocking
Both have similar useful properties:

`rect`
There is also `rect` that works like `block`, but has useful default inset and stroke:

Figures
For the purposes of adding a *figure* to your document, use `figure` function. Don’t try to use boxes or blocks there.

Figures are that things like centered images (probably with captions), tables, even code. In fact, you can put there anything you want:

Using spacing
Most time you will pass spacing into functions. There are special function fields that take only *size*. They are usually called like `width, length, in(out)set, spacing` and so on.

Like in CSS, one of the ways to set up spacing in Typst is setting margins and padding of elements. However, you can also insert spacing directly using functions `h` (horizontal spacing) and `v` (vertical spacing).

> Links to reference: , .

Absolute length units
Absolute length (aka just “length”) units are not affected by outer content and size of parent.

Relative to current font size
`1em = 1 current font size` : It is a very convenient unit, so it is used a lot in Typst.

Combined

Ratio length
`1% = 1% from parent size in that dimension`

Relative length
You can *combine* absolute and ratio lengths into *relative length*:

Fractional length

Single fraction length just takes *maximum size possible* to fill the parent: There are not many places you can use fractions, mainly those are `h` and `v`.

Several fractions

If you use several fractions inside one parent, they will take all remaining space *proportional to their number*:

Nested layout

Remember that fractions work in parent only, don't *rely on them in nested layout*:

Placing, Moving, Scale & Hide

This is **a very important section** if you want to do arbitrary things with layout, create custom elements and hacking a way around current Typst limitations.

TODO: WIP, add text and better examples

Place

Ignore layout, just put some object somehow relative to parent and current position. The placed object *will not* affect layouting Manually change position by `(dx, dy)` relative to intended.

Move

Scale

Scale content *without affecting the layout*.

Hide

Don't show content, but leave empty space there.

Tables and grids

While tables are not that necessary to know if you don't plan to use them in your documents, grids may be very useful for *document layout*. We will use both of them them in the book later.

Let's not bother with copying examples from official documentation. Just make sure to skim through it, okay?

Basic snippets

Spreading

Spreading operators (see) may be especially useful for the tables:

Highlighting table row

For individual cells, use

Splitting tables

Tables are split between pages automatically. However, if you want to make it breakable inside other element, you'll have to make that element breakable too:

Project structure

Large document

Once the document becomes large enough, it becomes harder to navigate it. If you haven't reached that size yet, you can ignore that section.

For managing that I would recommend splitting your document into *chapters* . It is just a way to work with this, but once you understand how it works, you can do anything you want.

Let's say you have two chapters, then the recommended structure will look like this: The exact file names are up to you.

Let's see what to put in each of these files.

Template

In the "template" file goes *all useful functions and variables* you will use across the chapters. If you have your own template or want to write one, you can write it there.

Main

This file should be compiled to get the whole compiled document.

Chapter

Notes

Note that modules in Typst can see only what they created themselves or imported. Anything else is invisible for them. That's why you need
`template.typ` file to define all functions within.

That means chapters *don't see each other either* , only what is in the template.

Cyclic imports

Important: Typst *forbids* cyclic imports. That means you can't import
`chapter_1` from `chapter_2` and `chapter_2` from `chapter_1` at the same time!

But the good news is that you can always create some other file to import variable from.

Scripting

Typst has a complete interpreted language inside. One of key aspects of working with your document in a nicer way

Basics

Variables I

Let's start with *variables* .

The concept is very simple, just some value you can reuse:

Variables II

You can store *any* Typst value in variable:

Functions

We have already seen some "custom" functions in chapter.

Functions are values that take some values and output some values:

Alternative syntax
You can write the same shorter:

Braces, brackets and default

Square brackets

You may remember that square brackets convert everything inside to *content* . We may use same for functions bodies: **Important:** It is very hard to convert *content* to *plain text* , as *content* may contain *anything* ! Sp be careful when passing and storing content in variables.

Braces

However, we often want to use code inside functions. That's when we use `{}`
:

Scopes

This is a very important thing to remember .

You can't use variables outside of scopes they are defined (unless it is file root, then you can import them) . Set and show rules affect things in their scope only.

Return

Important : by default braces return anything that "returns" into them. For example, To avoid returning everything, return only what you want explicitly, otherwise everything will be joined:

Default values

What we made just now was inventing "default values".

They are very common in styling, so there is a special syntax for them: You may have noticed that the argument became *named* now. In Typst, named argument is an argument *that has default value* .

Types, part I

Each value in Typst has a type. You don't have to specify it, but it is important.

Content (`content`)

>

We have already seen it. A type that represents what is displayed in document. **Important:** It is very hard to convert *content* to *plain text* , as *content* may contain *anything* ! So be careful when passing and storing content in variables.

None (`none`)

Nothing. Also known as `null` in other languages. It isn't displayed, converts to empty content.

String (`str`)

String contains only plain text and no formatting. Just some chars. That allows us to work with chars: You can convert other types to their string representation using this type's constructor (e.g. convert number to string):

Boolean (`bool`)

true/false. Used in `if` and many others

Integer (`int`)
A whole number.

The number can also be specified as hexadecimal, octal, or binary by starting it with a zero followed by either x, o, or b. You can convert a value to an integer with this type's constructor (e.g. convert string to int).

Float (`float`)
Works the same way as integer, but can store floating point numbers. However, precision may be lost. You can convert a value to a float with this type's constructor (e.g. convert string to float).

Types, part II
In Typst, most of things are **immutable** . You can't change content, you can just create new using this one (for example, using addition).

Immutability is very important for Typst since it tries to be *as pure language as possible* . Functions do nothing outside of returning some value.

However, purity is partly "broken" by these types. They are *super-useful* and not adding them would make Typst much pain.

However, using them adds complexity.

Arrays (`array`)
Mutable object that stores data with their indices. === List operations

Empty list

Dictionaries (`dict`)
Dictionaries are objects that store a string "key" and a value, associated with that key.

Empty dictionary

Conditions & loops

Conditions
> See .

In Typst, you can use `if-else` statements. This is especially useful inside function bodies to vary behavior depending on arguments types or many other things. Of course, `else` is unnecessary: You can also use `else if` statement (known as `elif` in Python):

Booleans
`if, else if, else` accept *only boolean* values as a switch. You can combine booleans as described in :

Loops
> See .

There are two kinds of loops: `while` and `for` . While repeats body while the condition is met: `for` iterates over all elements of sequence. The sequence may be an `array` , `string` or `dictionary` (`for` iterates over its *key- value pairs*). To iterate to all numbers from `a` to `b` , use `range(a, b+1)` : Because range is end-exclusive this is equal to

Break and continue

Inside loops can be used `break` and `continue` commands. `break` breaks loop, jumping outside. `continue` jumps to next loop iteration.

See the difference on these examples:

Advanced arguments

Spreading arguments from list

Spreading operator allows you to "unpack" the list of values into arguments of function: This may be super useful in tables:

Key arguments

The same idea works with key arguments:

Managing arbitrary arguments

Typst allows taking as many arbitrary positional and key arguments as you want.

In that case function is given special `arguments` object that stores in it positional and named arguments. You can combine them with other arguments. Spreading operator will "eat" all remaining arguments:

Optional argument

Currently the only way in Typst to create optional positional arguments is using `arguments` object:

TODO

Tips

There are lots of elements in Typst scripting that are not obvious, but important. All the book is designated to show them, but some of them

Equality

Equality doesn't mean objects are really the same, like in many other objects: That may be less obvious for dictionaries. In dictionaries **the order may matter** , so equality doesn't mean they behave exactly the same way:

Check key is in dictionary

Use the keyword `in` , like in `Python` : Note it works for lists too:

States & Query

This section is outdated. It may be still useful, but it is strongly recommended to study new context system (using the reference).

Typst tries to be a *pure language* as much as possible.

That means, a function can't change anything outside of it. That also means, if you call function, the result should be always the same.

Unfortunately, our world (and therefore our documents) isn't pure. If you create a heading №2, you want the next number to be three.

That section will guide you to using impure Typst. Don't overuse it, as this knowledge comes close to the Dark Arts of Typst!

States

This section is outdated. It may be still useful, but it is strongly recommended to study new context system (using the reference).

Before we start something practical, it is important to understand states in general.

Here is a good explanation of why do we *need* them: . It is highly recommended to read it first.

So instead of **THIS DOES NOT COMPILE:** Variables from outside the function are read-only and cannot be modified

Instead, you should write

Context magic

So what does this magic `context s.get()` mean?

>

In short, it specifies what part of your code (or markup) can *depend on states outside* . This context-expression is packed then as one object, and it is evaluated on layout stage.

That means it is impossible to look from "normal" code at whatever is inside the `context` . This is a black box that would be known *only after putting it into the document* .

We will discuss `context` features later.

Operations with states

Updating is *a content* that is an instruction. That instruction tells compiler that in this place of document the state *should be updated* . Here we can see one of *important* `context` *traits* : it "sees" states from outside, but can't see how they change inside it:

ID collision

*TLDR;****Never allow colliding states.***

States are described by their id-s, if they are the same, the code will break.

So, if you write functions or loops that are used several times, *be careful* ! However, this *may seem* okay: But in fact, it *isn't* :

Counters

This section is outdated. It may be still useful, but it is strongly recommended to study new context system (using the reference).

Counters are special states that *count* elements of some type. As with states, you can create your own with identifier strings.

Important: to initiate counters of elements, you need to *set numbering for them* .

States methods

Counters are states, so they can do all things states can do.

Displaying counters

Counters also support displaying *both current and final values* out-of-box:

Step

That's quite easy, for counters you can increment value using `step` . It works the same way as `update` .

You can use counters in your functions:

Measure, Layout

This section is outdated. It may be still useful, but it is strongly recommended to study new context system (using the reference).

Style & Measure

> Style

> Measure

`measure` returns *the element size*. This command is extremely helpful when doing custom layout with `place`.

However, there is a catch. Element size depends on styles, applied to this element. So if we will set the big text size for some part of our text, to measure the element's size, we have to know *where the element is located*. Without knowing it, we can't tell what styles should be applied.

So we need a scheme similar to `locate`.

This is what `styles` function is used for. It is *a content*, which, when located in document, calls a function inside on *current styles*.

Now, when we got fixed `styles`, we can get the element's size using `measure`:

Layout

Layout is similar to `measure`, but it returns current scope **parent size**.

If you are putting elements in block, that will be block's size. If you are just putting right on the page, that will be page's size.

As parent's size depends on it's place in document, it uses the similar scheme to `locate` and `style`: It may be extremely useful to combine `layout` with `measure`, to get width of things that depend on parent's size:

Query

This section is outdated. It may be still useful, but it is strongly recommended to study new context system (using the reference). Query is a thing that allows you getting location by *selector* (this is the same thing we used in show rules).

That enables "time travel", getting information about document from its parts and so on. *That is a way to violate Typst's purity.*

It is currently one of the *the darkest magics currently existing in Typst*. It gives you great powers, but with great power comes great responsibility. = Metadata

Metadata is invisible content that can be extracted using query or other content. This may be very useful with `typst query` to pass values to external tools.

Math

Math is a special environment that has special features related to… math.

Syntax

To start math environment, `$` . The spacing around `$` will make it either *inline* math (smaller, used in text) or *display* math (used on math equations on their own).

Math.equation

The element that math is displayed in is called `math.equation` . You can use it for set/show rules: Any symbol/command that is available in math, *is also available* in code mode using `math.command` :

Letters and commands

Typst aims to have as simple and effective syntax for math as possible. That means no special symbols, just using commands.

To make it short, Typst uses several simple rules: If you use kebab-case or snake_case for variables you want to use in math, you will have to refer to them as #snake-case-variable. Spacing matters there! Commands see (go to the links to see the commands).

All symbols see .

Multiline equations

To create multiline *display equation* , use the same symbol as in markup mode: `\` :

Escaping

Any symbol that is used may be escaped with `\` , like in markup mode. For example, you can disable fraction: The same way it works with any other syntax.

Wrapping inline math

Sometimes, when you write large math, it may be too close to text (especially for some long letter tails). You may easily increase the distance it by wrapping into box:

Symbols

Multiletter words in math refer either to local variables, functions, text operators, spacing or *special symbols* . The latter are very important for advanced math. You can write the same with unicode:

Symbols naming

> See all available symbols list .

General idea

Typst wants to define some “basic” symbols with small easy-to-remember words, and build complex ones using combinations. For example, I highly recommend using WebApp/Typst LSP when writing math with lots of complex symbols. That helps you to quickly choose the right symbol within all combinations.

Sometimes the names are not obvious, for example, sometimes it is used prefix `n-` instead of `not` :

Common modifiers

Packages

Once the was launched, this chapter has become almost redundant. The Universe is actually a very cool place to look for packages.

However, there are still some cool examples of interesting package usage.

General

Typst has packages, but, unlike LaTeX, you need to remember:

To use mighty package, just write, like this:

Contributing

If you are author of a package or just want to make a fair overview, feel free to make issues/PR-s!

Pretty things

Set bar to the text's left

(also known as quote formatting)= **book/snippets/numbering.md**

Numbering

== Math numbering

See .

Numbering each paragraph

By the 0.12 version of Typst, this should be replaced with good native solution.

Logos & Figures

Using SVG-s images is totally fine. Totally. But if you are lazy and don't want to search for images, here are some logos you can just copy-paste in your document.

Important : *Typst in text doesn't need a special writing (unlike LaTeX)* . Just write "Typst", maybe " **Typst** ", and it is okay. = **book/snippets/external.md**

Use with external tools

Currently the best ways to communicate is using

In some time there will be examples of successful usage of first two methods. For the third one, see .

Fractional grids

For tables with lines of changing length, you can try using *grids in grids* .

Don't use this where will do.

Automerge adjacent cells with same values

This example works for adjacent cells horizontally, but it's not hard to upgrade it to columns too.

Slanted column headers with slanted borders

Demos

= **book/snippets/index.md**

Typst Snippets

Useful snippets for common (and not) tasks.

Labels

= **book/snippets/code.md**

Code formatting

== Theme

See See section.

Color & Gradients

Gradients

Gradients may be very cool for presentations or just a pretty look.

Special symbols

> *Important:* I'm not great with special symbols, so I would additionally > appreciate additions and corrections.

Typst has a great support of *unicode* . That also means it supports *special symbols* . They may be very useful for typesetting.

In most cases, you shouldn't use these symbols directly often. If possible, use them with show rules (for example, replace all `-th` with `\u{2011}th` , a non-breaking hyphen).

Non-breaking symbols

Non-breaking symbols can make sure the word/phrase will not be separated. Typst will try to put them as a whole.

Non-breaking space

> *Important:* As it is spacing symbols, copy-pasting it will not help. Typst > will see it as just a usual spacing symbol you used for your source code to > look nicer in your editor. Again, it will interpret it *as a basic space* .

This is a symbol you should't use often (use Typst boxes instead), but it is a good demonstration of how non-breaking symbol work:

Non-breaking hyphen

Connectors and separators

World joiner

Initially, world joiner indicates that no line break should occur at this position. It is also a zero-width symbol (invisible), so it can be used as a space removing thing:

Zero width space

Similar to word-joiner, but this is a *space* . It doesn't prevent word break. On the contrary, it breaks it without any hyphen at all!

Typst Basics

This is a chapter that consistently introduces you to the most things you need to know when writing with Typst.

It show much more things than official tutorial, so maybe it will be interesting to read for some of the experienced users too.

Some examples are taken from and . Most are created and edited specially for this book.

> *Important:* in most cases there will be used “clipped” examples of your > rendered documents (no margins, smaller width and so on). > > To set up the spacing as you want, see .

Extra

Bibliography

Typst supports bibliography using BibLaTex `.bib` file or its own Hayagriva `.yml` format.

BibLaTex is wider supported, but Hayagriva is easier to work with.

> Link to Hayagriva and some

Citation Style

The style can be customized via CSL, citation style language, with more than 10 000 styles available online. See .

Image with original size

This function renders image with the size it “naturally” has.

Note: starting from v0.11 , Typst tries using default image size when width and height are `auto` . It only uses container’s size if the image doesn’t fit. So this code is more like a legacy, but still may be useful.

This works because measure conceptually places the image onto a page with infinite size and then the image defaults to 1pt per pixel instead of becoming infinitely larger itself.

Empty pages without numbering

Empty pages before chapters starting at odd pages

This snippet has been broken on 0.12.0. If someone will help fixing it, this would be cool.

Extracting plain text

Make all math display math

May slightly interfere with math blocks.

Horizontally align something with something

Remove indent from nested lists

Word count

This chapter is deprecated now. It will be removed soon.

Recommended solution

Use `wordometr` := **book/typstonomicon/index.md**

Typstonomicon, or The Code You Should Not Write

Totally cursed examples with lots of quires, measure and other things to hack around current Typst limitations. Generally you should use this code only if you really need it.

Code in this chapter may break in lots of circumstances and debugging it will be very painful. You are warned.

I think that this chapter will slowly die as Typst matures.

Try & Catch

Breakpoints on broken blocks

Implementation with table headers & footers
See a demo project (more comments, I stripped some of them) .

Implementation via headers, footers and stated
Limitations: **works only with one-column layout and one break** .

Create zero-level chapters

Multiple show rules

Sometimes there is a need to apply several rules that look very similar. Or generate them from code. One of the ways to deal with this, the most cursed one, is this: The recursion problem may be avoided with the power of `fold` , with basically the same idea: Note that just in case of symbols (if you don't need element functions), one can use regular expressions. That is a more robust way:

Tables

== Tablem: markdown tables

> See documentation

Render markdown tables in Typst.

Custom render

Presentations

Polylux

> See

Slydst

> See the documentation .

Much more simpler and less powerful than polulyx:

Physics

`physica`

> Physica (Latin for *natural sciences*) provides utilities that simplify > otherwise complex and repetitive mathematical expressions in natural > sciences.

> Its provides a full set of > demonstrations of how the package could be helpful.

Mathematical physics
The page has more examples on its math capabilities. Below is a preview that may be of particular interest in the domain of physics:

A partial glimpse: In the default font, the Typst built-in symbol `planck.reduce` looks a bit off: on letter “h” there is a slash instead of a horizontal bar, contrary to the symbol’s colloquial name “h-bar”. This package offers `hbar` to render the symbol in the familiar form. Contrast:

`quill` : quantum diagrams

> See .

Glossary

glossarium

>

Package to manage glossary and abbreviations.

One of the very first cool packages of Typst, made specially for (probably) the first thesis written in Typst.

Headers

`hydra` : Contextual headers

We have discussed in `Typst Basics` how to get current heading with `query(selector(heading).before(here()))` for headers. However, this works badly for nested headings with numbering and similar things. For these cases there is `hydra` :

Math

General

`physica`

> Physica (Latin for *natural sciences*) provides utilities that simplify > otherwise complex and repetitive mathematical expressions in natural > sciences.

> Its provides a full set of > demonstrations of how the package could be helpful.

Common notations

Below is a preview of those notations.

Matrices

In addition to Typst’s built-in `mat()` to write a matrix, physica provides a number of handy tools to make it even easier. Diagonal matrix `dmat( ... )` , antidiagonal matrix `admat( ... )` , identity matrix `imat(n)` , and zero matrix `zmat(n)` . Jacobian matrix with `jmat(func; ... )` or the longer name `jacobianmatrix` , Hessian matrix with `hmat(func; ... )` or the longer name `hessianmatrix` , and finally `xmat(row, col, func)` to build a matrix.

`mitex`

> MiTeX provides LaTeX support powered by WASM in Typst, including real-time > rendering of LaTeX math equations. You can also use LaTeX syntax to write `> \r ef` and `\l abel` .

> Please refer to the for more > details.

`i-figured`

Configurable equation numbering per section in Typst. There is also figure numbering per section, see more examples in its .

Theorems

`ctheorem`

A numbered theorem environment in Typst. See more examples in its .

`lemmify`

Lemmify is another theorem evironment generator with many selector and numbering capabilities. See documentations in its .

Wrapping figures

The better native support for wrapping is planned, however, something is already possible via package: Limitations: non-ideal spacing near warping, only top-bottom left/right are supported.

External

These are not official packages. Maybe once they will become one.

However, they may be very useful.

Treemap display

Graphs

`cetz`

Cetz comes with quite built-in support of drawing basic graphs. It is much more customizable and extensible then packages like `plotst` , so it is recommended to skim through its possibilities.

> See full manual .

Draw a graph in polar coords

`diagraph`

=== FFT

Labels are overridden manually. > See . Labels for nodes `big` and `sum` are overridden.

`bob-draw`

WASM plugin for to draw easily with ASCII,. Finite automata. See the for a full documentation.

Counting words

Wordometr

Excluding elements

You can exclude elements by name (e.g., `"caption"`), function (e.g., `figure.caption`), where-selector (e.g., `raw.where(block: true)`), or `label` (e.g., `< no-wc >`).

Misc

Formatting strings

== `nth` , Nth element

Custom boxes

== Theorems

See

Code

`codly`

> See docs

Codelst

Layouting

General useful things.

Pinit: relative place by pins

The idea of is pinning pins on the normal flow of the text, and then placing the content relative to pins. More complex example: > Get more info == Headings for actual current chapter

> See

Drawing

`cetz`

Cetz is an analogue of LaTeX's `tikz` . Maybe it is not as powerful yet, but certainly easier to learn and use.

It is the best choice in most of cases you want to draw something in Typst.

Scripting

== Split the string retrieving separators

Create selector matching any values in an array

This snippet creates a selector (that is then used in a show rule) that matches any of the values inside the array. Here, it is used to highlight a few raw lines, but it can be easily adapted to any kind of selector.

Synthesize show (or show-set) rules from dictionnary

This snippet applies a show-set rule to any element inside a dictionary, by using the key as the selector and the value as the parameter to set. In this example, it's used to give custom supplements to custom figure kinds, based on a dictionnary of correspondances. Additonnaly, as this is applied at the position where you write it, these show-set rules will appear as if they were added in the same place where you wrote this rule. This means that you can override them later, just like any other show-set rules.

Outlines

Outlines

> Lots of outlines examples are already available in You can use arbitrary selector there, so you can do any crazy things. == Replace default dots== Ignore citations and footnotes

That's a workaround a problem that if you make a footnote a heading, its number will be displayed in outline:

Page numbering

Separate page numbering for each chapter

Page setup

> See

Duplicate content

Notice that this implementation will mess up with labels and similar things. For complex cases see one below.

```
#set page(paper: "a4", flipped: true) #show: body => grid( columns:
(1fr, 1fr), column-gutter: 1cm, body, body, ) #lorem(200)
```

Advanced

Shaped boxes with text

(I guess that will make a package eventually, but let it be a snippet for now)

Multiline detection

Detects if figure caption (may be any other element) *has more than one line* .

If the caption is multiline, it makes it left-aligned.

Breaks on manual linebreaks.

Lines between list items

The same approach may be easily adapted to style the enums as you want.

Hiding things

Vectors & Matrices

You can easily note that the gap isn't necessarily even or the same in different vectors and matrices: That happens because `gap` refers to *spacing between* elements, not the distance between their centers.

To fix this, you can use this snippet:

Operations

Fractions= book/snippets/math/numbering.md

Math Numbering

Number by current heading

> See also built-in numbering in

Number only labeled equations

= **book/snippets/math/scripts.md**

Scripts

> To set scripts and limits see

Make every character upright when used in subscript

Fonts

Set math font

Important: The font you want to set for math should *contain* necessary math symbols. That should be a special font with math. If it isn't, you are very likely to get *an error* (remember to set `fallback: false` and check `typst fonts` to debug the fonts).

Calligraphic letters

Unfortunately, currently just `stylistic-set` for math creates bad spacing. Math engine detects if the letter should be correctly spaced by whether it is the default font. However, just making it "normal" isn't enough, because than it can be reduced. That's way the snippet is as hacky as it is (probably should be located in Typstonomicon, but it's not large enough).

Special documents

See .

Forms

> Presentation interactive forms are coming! They are currently under heavy > work by @tinger.

Fake italic & Text shadows

Skew

Individual language fonts

Passing arguments

Then, in template file: Writing templates is fairly easy if you understand .

See more information about writing templates in .

There is no official repository for templates yet, but there are a plenty community ones in .

Tutorial by Examples

The first section of Typst Basics is very similar to , with more specialized examples and less words. It is *highly recommended to read the official tutorial anyway* .

Numbering

Of course, there are lots of other cool properties that can be *set* , so feel free to dive into and explore them!

And now we are moving into something much more interesting…

Default values

What we made just now was inventing “default values”.

They are very common in styling, so there is a special syntax for them: You may have noticed that the argument became *named* now. In Typst, named argument is an argument *that has default value* .

Booleans

`if, else if, else` accept *only boolean* values as a switch. You can combine booleans as described in :

Scripting

Typst has a complete interpreted language inside. One of key aspects of working with your document in a nicer way

Optional argument

Currently the only way in Typst to create optional positional arguments is using `arguments` *object:*

TODO

Classes

> See

Each math symbol has its own “class”, the way it behaves. That’s one of the main reasons why they are layouted differently. = **book/basics/math/operators.md**

Operators

> See .

There are lots of built-in “text operators” in Typst math. This is a symbol that behaves very close to plain text. Nevertheless, it is different:

Predefined operators

Here are all text operators Typst has built-in:

Creating custom operator

Of course, there always will be some text operators you will need that are not in the list.

But don’t worry, it is very easy to add your own:

Limits for operators

When creating operators (upright text with proper spacing), you can set limits for *display mode* at the same time: This is roughly equivalent to Everything can be combined to create new operators:

Alignment

General alignment

By default display math is center-aligned, but that can be set up with `show` rule: Or using `align` element:

Alignment points

When equations include multiple alignment points (&), this creates blocks of alternatingly *right-* and *left-* aligned columns.

In the example below, the expression `(3x + y) / 7` is *right-aligned* and `= 9` is *left-aligned* . The word “given” is also left-aligned because `&&` creates two alignment points in a row, *alternating the alignment twice* .

`& &` and `&&` behave exactly the same way. Meanwhile, “multiply by 7” is left-aligned because just one `&` precedes it.

Each alignment point simply alternates between right-aligned/left-aligned.

Setting limits

Sometimes we want to change how the default attaching should work.

Limits

For example, in many countries it is common to write definite integrals with limits below and above. To set this, use `limits` function: You can set this by default using `show` rule:

Only display mode

Notice that this will also affect inline equations. To enable limits for display math only, use `limits(inline: false)` : Of course, it is possible to move them back as bottom attachments:

Operations

The same scheme works for operations. By default, they are attached to the bottom and top:

Vectors, matrices, semicolumn syntax

Vectors

> By vector we mean a column there. > To write arrow notations for letters, use `$ arrow ( v ) $` > I recommend to create shortcut for this, like `#let arr = math.arrow`

To write columns, use `vec` command:

Delimiter

You can change parentheses around the column or even remove them:

Gap

You can change the size of gap between rows:

Making gap even

You can easily note that the gap isn’t necessarily even or the same in different vectors: That happens because `gap` refers to *spacing between* elements, not the distance between their centers.

To fix this, you can use .

Matrix

> See

Matrix is very similar to `vec`, but accepts rows, separated by `;`:

Delimiters and gaps

You can specify them the same way as for vectors.

Specify the arguments either before the content, or **after the semicolon**. The code will panic if there is no semicolon!

Semicolon syntax

When you use semicolons, the arguments *between the semicolons* are merged into arrays. See yourself: If you miss some of elements, they will be replaced by `none`-s.

You can mix semicolon syntax and named arguments, but be careful! For example, this will not work:

Location and sizes

We talked already about display and inline math. They differ not only by aligning and spacing, but also by size and style: The size and style of current environment is described by Math Size, see .

There are for sizes:

Each time thing is used in fraction, script or exponent, it is moved several "levels lowers", becoming smaller and more "crapping". `sscript` isn't reduced father:

Setting sizes manually

Just use the corresponding command:

Grouping

Every grouping can be (currently) done by parenthesis. So the parenthesis may be both "real" parenthesis and grouping ones.

For example, these parentheses specify nominator of the fraction:

Left-right

> See .

If there are two matching braces of any kind, they will be wrapped as `lr` (left-right) group. You can disable it by escaping.

You can also match braces of any kind by using `lr` directly:

Fences

Fences *are not matched automatically* because of large amount of false- positives.

You can use `abs` or `norm` to match them:

book/basics/states/metadata.md

States & Query

This section is outdated. It may be still useful, but it is strongly recommended to study new context system (using the reference).

Typst tries to be a *pure language* as much as possible.

That means, a function can't change anything outside of it. That also means, if you call function, the result should be always the same.

Unfortunately, our world (and therefore our documents) isn't pure. If you create a heading №2, you want the next number to be three.

That section will guide you to using impure Typst. Don't overuse it, as this knowledge comes close to the Dark Arts of Typst!

Spreading
Spreading operators (see) may be especially useful for the tables:

Cyclic imports

Important: Typst *forbids* cyclic imports. That means you can't import `chapter_1` from `chapter_2` and `chapter_2` from `chapter_1` at the same time!

But the good news is that you can always create some other file to import variable from.

Must-know

This section contains things, that are not general enough to be part of "tutorial", but still are very important to know for proper typesetting.

Feel free to skip through things you are sure you will not use.