Welcome To Presentate!

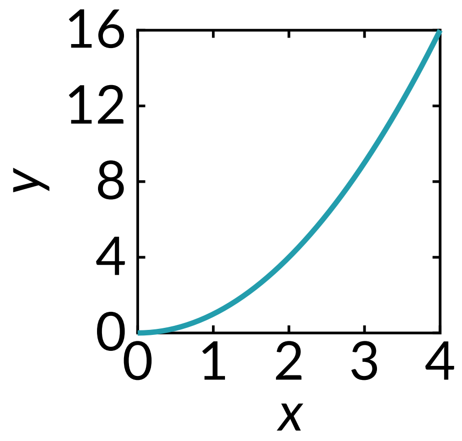
Tools for creating integrated dynamic slides.

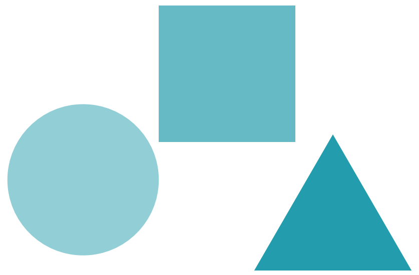
@pacaunt | 2026-02-04

Outline

1 Introduction

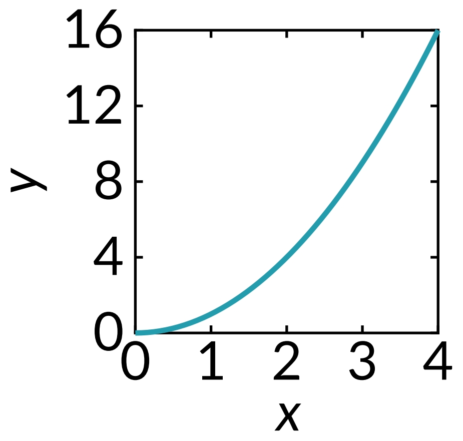
1.1 Presentation by Coding?

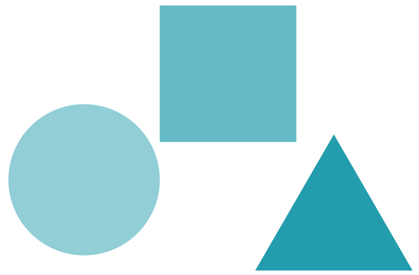




Presentation’s objective is to convey information to the audiences. **Slide deck** is one of visual media we can use for such tasks.

1.1 Presentation by Coding?

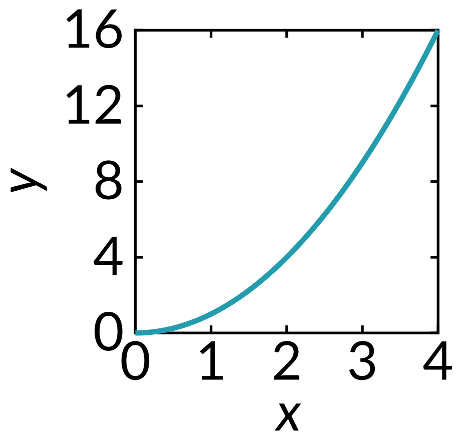


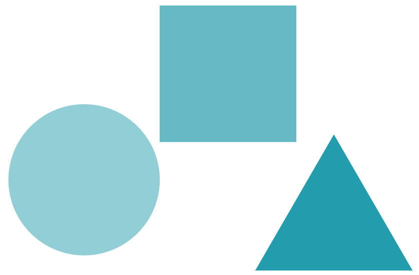


Presentation's objective is to convey information to the audiences. **Slide deck** is one of visual media we can use for such tasks.

Often, using visual tools is easy, as you can modify what you want. However, we have to admit that sometimes creating visual media is *easier in code*.

1.1 Presentation by Coding?



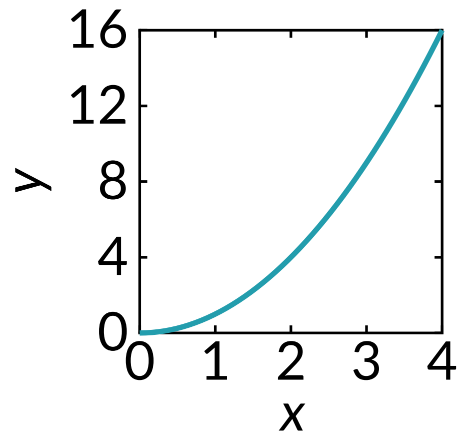


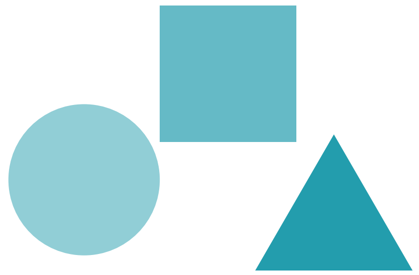
Presentation's objective is to convey information to the audiences. **Slide deck** is one of visual media we can use for such tasks.

Often, using visual tools is easy, as you can modify what you want. However, we have to admit that sometimes creating visual media is *easier in code*.

Imagine creating visual graphs that update directly from your source project.

1.1 Presentation by Coding?





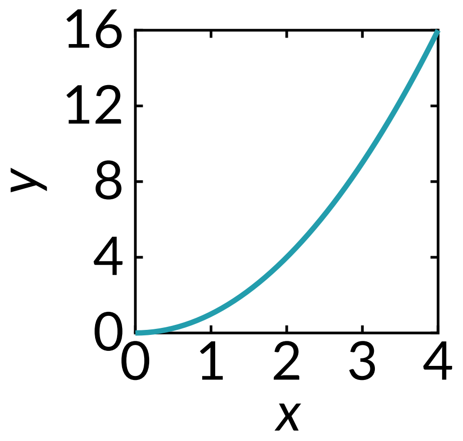
Presentation's objective is to convey information to the audiences. **Slide deck** is one of visual media we can use for such tasks.

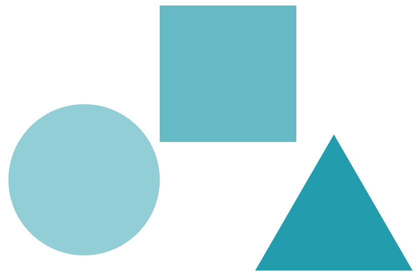
Often, using visual tools is easy, as you can modify what you want. However, we have to admit that sometimes creating visual media is *easier in code*.

Imagine creating visual graphs that update directly from your source project.

So you don't have to update them manually.

1.1 Presentation by Coding?





Presentation's objective is to convey information to the audiences. **Slide deck** is one of visual media we can use for such tasks.

Often, using visual tools is easy, as you can modify what you want. However, we have to admit that sometimes creating visual media is *easier in code*.

Imagine creating visual graphs that update directly from your source project.

So you *don't have to update them manually*.

1.2 A note about animating PDFs

Creating presentation in Typst, especially in PDF format, cannot provide the *actual animated* scenes like videos.

1.2 A note about animating PDFs

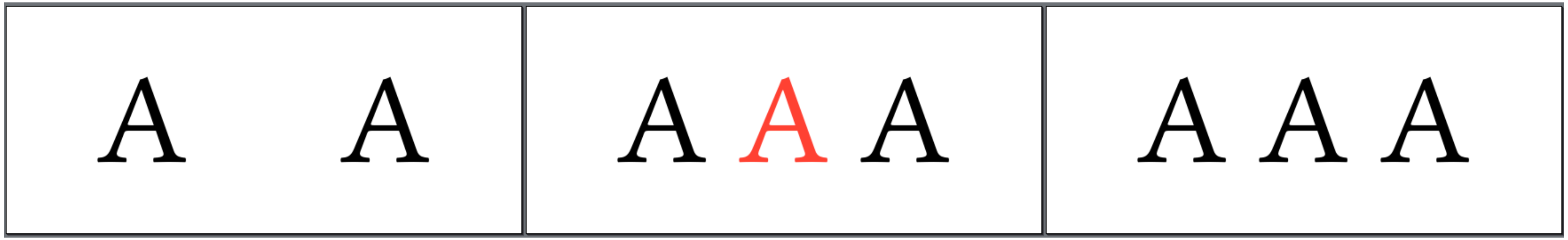
Creating presentation in Typst, especially in PDF format, cannot provide the *actual animated* scenes like videos.

However, the *dynamic* contents on the following examples are generated by *repeatedly printed* each page, which contains slightly different components.

1.2 A note about animating PDFs

Creating presentation in Typst, especially in PDF format, cannot provide the *actual animated* scenes like videos.

However, the *dynamic* contents on the following examples are generated by *repeatedly printed* each page, which contains slightly different components.



So that when you see on the screen, it *looks like* the contents are changing.

1.3 Integration of Tools

Presentate is another package written in Typst for creating slides.

1.3 Integration of Tools

Presentate is another package written in Typst for creating slides.

We already have other powerful presentation packages! So it rises a question:

1.3 Integration of Tools

Presentate is another package written in Typst for creating slides.

We already have other powerful presentation packages! So it rises a question:

Why creating another?

1.3 Integration of Tools

Presentate is another package written in Typst for creating slides.

We already have other powerful presentation packages! So it rises a question:

Why creating another?

The answer is **Package Integrations**.

1.3 Integration of Tools

Slide animation requires information in type `content`; however, most packages for creating visual data output requires *non*-`content` input, such as

1.3 Integration of Tools

Slide animation requires information in type `content`; however, most packages for creating visual data output requires *non*-`content` input, such as

- CeTZ: arrays of functions

1.3 Integration of Tools

Slide animation requires information in type `content`; however, most packages for creating visual data output requires *non*-`content` input, such as

- CeTZ: arrays of functions
- Fletcher: special metadata

1.3 Integration of Tools

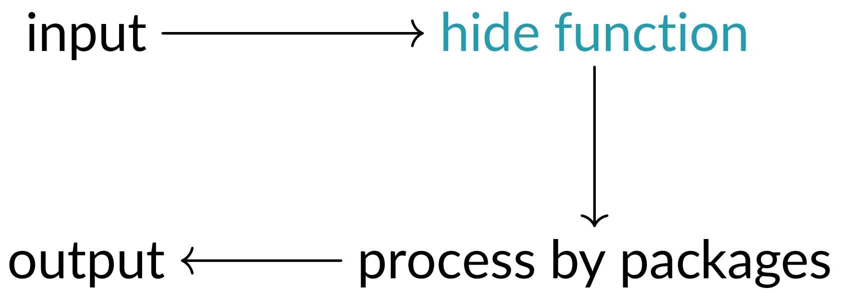
Slide animation requires information in type `content`; however, most packages for creating visual data output requires *non*-`content` input, such as

- CeTZ: arrays of functions
- Fletcher: special metadata
- Alchemist: arrays of dictionary

1.3 Integration of Tools

Slide animation requires information in type `content`; however, most packages for creating visual data output requires *non*-`content` input, such as

- CeTZ: arrays of functions
- Fletcher: special metadata
- Alchemist: arrays of dictionary



1.3 Integration of Tools

Slide animation requires information in type `content`; however, most packages for creating visual data output requires *non*-`content` input, such as

- CeTZ: arrays of functions
- Fletcher: special metadata
- Alchemist: arrays of dictionary

$$\begin{tikzcd}
\text{input} \arrow[r] & \text{hide function} \arrow[d] \\
\text{output} & \text{process by packages} \arrow[l]
\end{tikzcd}$$

So to create animation with those packages, we need some functionality to be able to *hide* the information *without* `content` generation.

1.3 Integration of Tools

Here is when presentate comes in.

[1] https://typst.app/universe/package/alchemist

1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:

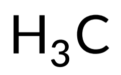
[1] https://typst.app/universe/package/alchemist

1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:



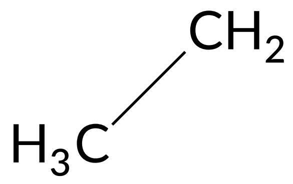
[1] https://typst.app/universe/package/alchemist

1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:



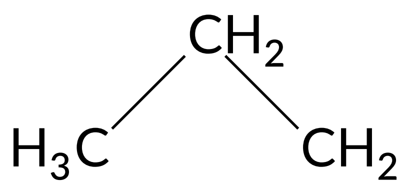
[1] https://typst.app/universe/package/alchemist

1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:



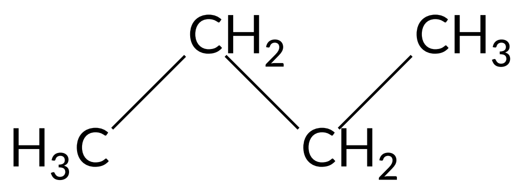
[1] https://typst.app/universe/package/alchemist

1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:



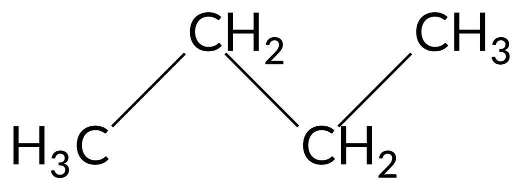
[1] https://typst.app/universe/package/alchemist

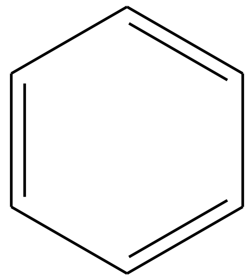
1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:





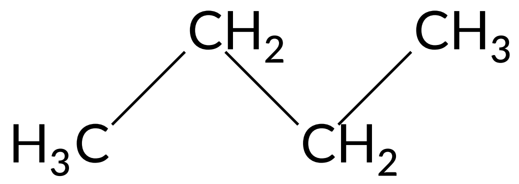
[1] https://typst.app/universe/package/alchemist

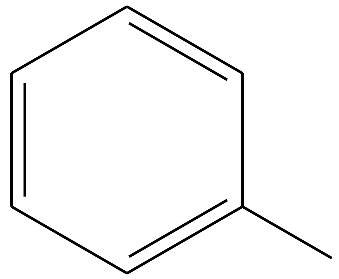
1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:





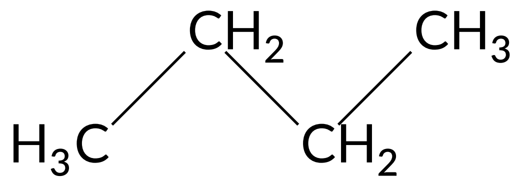
[1] https://typst.app/universe/package/alchemist

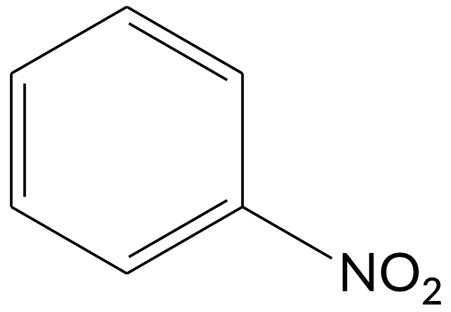
1.3 Integration of Tools

Here is when presentate comes in.

Presentate provides a framework for rendering input and output of *any kind*.

Like the following molecule drawing animation from Alchemist[1] package:





[1] https://typst.app/universe/package/alchemist

1.4 About Presentate

Presentate is a package aimimg to provide a *framework* for creating dynamic presentation animations that are *flexible* enough to be used with any packages.

1.4 About Presentate

Presentate is a package aimimg to provide a *framework* for creating dynamic presentation animations that are *flexible* enough to be used with any packages.

The package provides:

1.4 About Presentate

Presentate is a package aimimg to provide a *framework* for creating dynamic presentation animations that are *flexible* enough to be used with any packages.

The package provides:

- revealing content step-by-step from `#show: pause`,

1.4 About Presentate

Presentate is a package aimimg to provide a *framework* for creating dynamic presentation animations that are *flexible* enough to be used with any packages.

The package provides:

- revealing content step-by-step from `#show: pause`,
- revealing content specifically from `#uncover(..)` and `#only(..)`,

1.4 About Presentate

Presentate is a package aimimg to provide a *framework* for creating dynamic presentation animations that are *flexible* enough to be used with any packages.

The package provides:

- revealing content step-by-step from `#show: pause`,
- revealing content specifically from `#uncover(..)` and `#only(..)`,
- transform content by `#transform(..)`,

1.4 About Presentate

Presentate is a package aimimg to provide a *framework* for creating dynamic presentation animations that are *flexible* enough to be used with any packages.

The package provides:

- revealing content step-by-step from `#show: pause`,
- revealing content specifically from `#uncover(..)` and `#only(..)`,
- transform content by `#transform(..)`,
- relative index like `#auto` and `#none`,

1.4 About Presentate

Presentate is a package aimimg to provide a *framework* for creating dynamic presentation animations that are *flexible* enough to be used with any packages.

The package provides:

- revealing content step-by-step from `#show: pause`,
- revealing content specifically from `#uncover(..)` and `#only(..)`,
- transform content by `#transform(..)`,
- relative index like `#auto` and `#none`,
- render frame for package integration, with `#animation` module.

1.5 Acknowledgement

The package was created by mixing my original motivation and insprations from many existing presentation packages.

Thanks to: Polylux[2] for `subslide` implementation and pdfpc support,
Touying[3] for idea of render frame, fake frozen states, and
Minideck[4] for `#only`, and `#uncover` functions.

[2] https://github.com/polylux-typ/polylux
[3] https://github.com/touying-typ/touying
[4] https://github.com/knuesel/typst-minideck

2 Usage

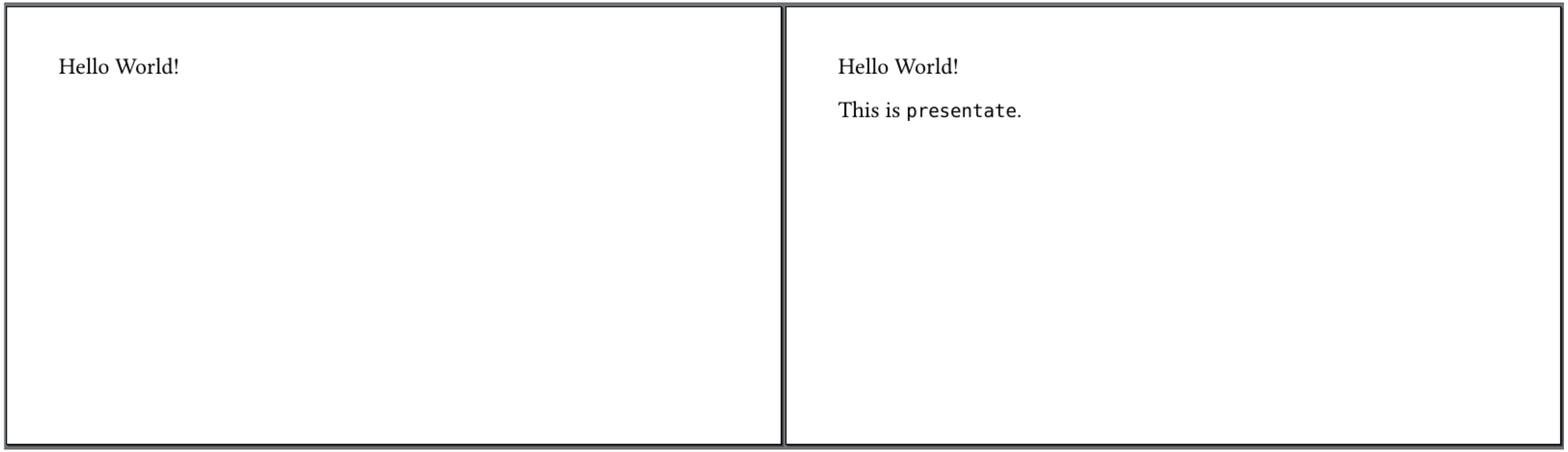
2.1 Getting Started

Start with the following snippets:

```typst
#import "@preview/presentate:0.2.4": *
#set text(size: 25pt) // of your choice

#slide[
    Hello World!
    #show: pause;

    This is `presentate`.
]
```

2.1 Getting Started

Then you will have:



Hello World!

Hello World!

This is `presentate`.

2.1 Getting Started

You may styling the way you want, for example:

```
#import "@preview/presentate:0.2.4": *
#set page(paper: "presentation-16-9")
#set text(size: 25pt, font: "FiraCode Nerd Font Mono")
#set align(horizon)
#slide[
  = Welcome to Presentate!
  \ A lazy author \
  #datetime.today().display()
]
```

2.1 Getting Started

(continued)

```
#set align(top)
#slide[
  == Tips for Typst.
  #set align(horizon)
  Do you know that $pi != 3.141592$?

  #show: pause
  Yeah. Certainly.

  #show: pause
  Also $pi != 22/7$.
]
```

2.1 Getting Started

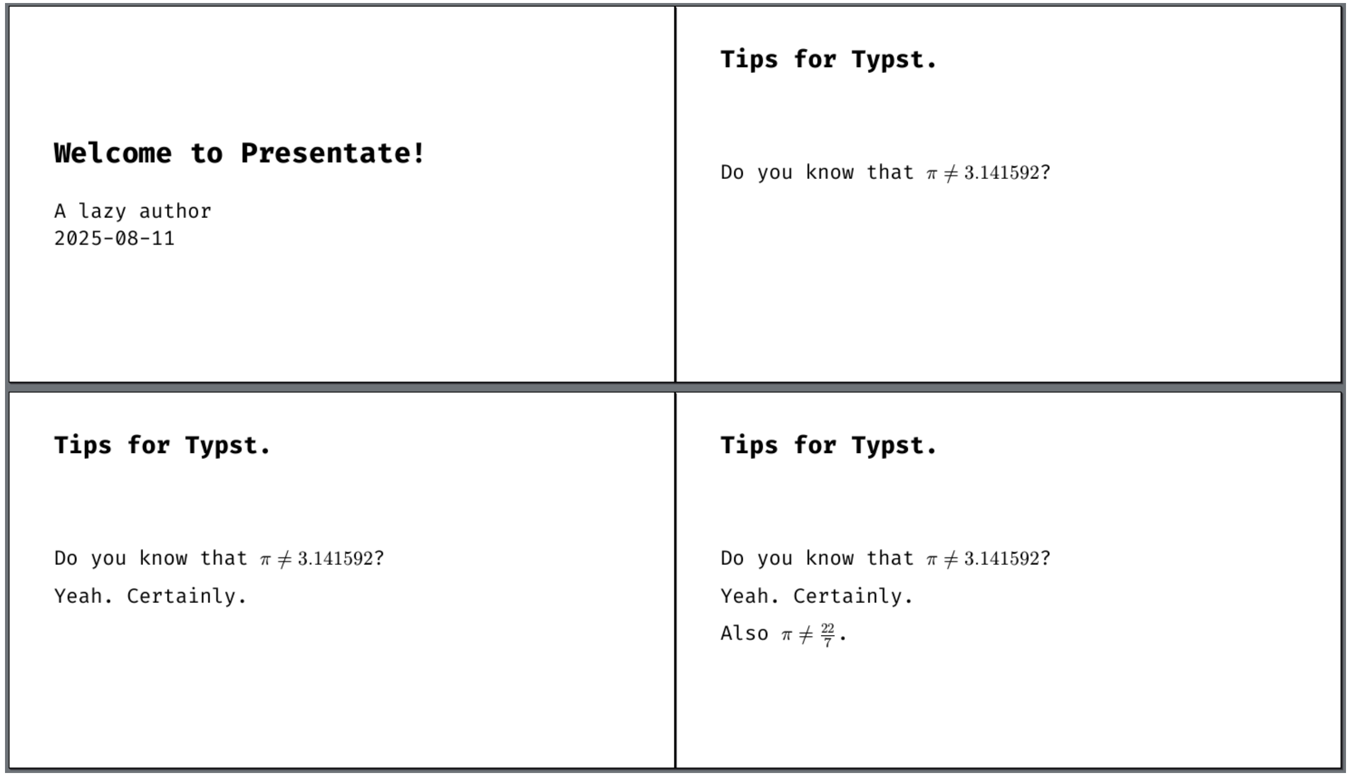
(continued)

```
#set align(top)
#slide[
  == Tips for Typst.
  #set align(horizon)
  Do you know that $pi !=
  3.141592$?

  #show: pause
  Yeah. Certainly.

  #show: pause
  Also $pi != 22/7$.
]
```

Presentate does not interfere Typst styling systems, so you can set and unset anything freely.

The results are on the next slide:

2.1 Getting Started



Welcome to Presentate!

A lazy author
2025-08-11

Tips for Typst.

Do you know that $\pi \neq 3.141592$?

Tips for Typst.

Do you know that $\pi \neq 3.141592$?

Yeah. Certainly.

Tips for Typst.

Do you know that $\pi \neq 3.141592$?

Yeah. Certainly.

Also $\pi \neq \frac{22}{7}$.

2.2 Dynamic Components

`#slide[..]` function provides a workspace for creating *animations*.
As the example showing the use of `#show: pause` functionality.

2.2 Dynamic Components

`#slide[..]` function provides a workspace for creating *animations* .
As the example showing the use of `#show: pause` functionality.

Presentate provides the following functions for creating dynamic slides:

2.2 Dynamic Components

`#slide[..]` function provides a workspace for creating *animations*.
As the example showing the use of `#show: pause` functionality.

Presentate provides the following functions for creating dynamic slides:

1. `#pause(..)` for basic reveal of content in chunks.

2.2 Dynamic Components

`#slide[..]` function provides a workspace for creating *animations*.
As the example showing the use of `#show: pause` functionality.

Presentate provides the following functions for creating dynamic slides:

1. `#pause(..)` for basic reveal of content in chunks.
2. `#uncover(..)` and `#only(..)` for precise steps of revealing content.

2.2 Dynamic Components

`#slide[..]` function provides a workspace for creating *animations*.
As the example showing the use of `#show: pause` functionality.

Presentate provides the following functions for creating dynamic slides:

1. `#pause(..)` for basic reveal of content in chunks.
2. `#uncover(..)` and `#only(..)` for precise steps of revealing content.
3. `#fragments(..)` for revealing content one-by-one.

2.2 Dynamic Components

`#slide[..]` function provides a workspace for creating *animations*.
As the example showing the use of `#show: pause` functionality.

Presentate provides the following functions for creating dynamic slides:

1. `#pause(..)` for basic reveal of content in chunks.
2. `#uncover(..)` and `#only(..)` for precise steps of revealing content.
3. `#fragments(..)` for revealing content one-by-one.
4. `#transform(..)` for transform the content by functions.

2.2 Dynamic Components

`#slide[..]` function provides a workspace for creating *animations*.
As the example showing the use of `#show: pause` functionality.

Presentate provides the following functions for creating dynamic slides:

1. `#pause(..)` for basic reveal of content in chunks.
2. `#uncover(..)` and `#only(..)` for precise steps of revealing content.
3. `#fragments(..)` for revealing content one-by-one.
4. `#transform(..)` for transform the content by functions.
5. `#render(..)` and `#animate(..)` for handling non-content type data.

2.3 `#pause` function

Basic usage of `#pause(..)` is usually in the form `#show: pause`.
Apart from that, you can put any content in the `(..)`, e.g. math equations.

2.3 #pause function

Basic usage of `#pause(..)` is usually in the form `#show: pause`.
Apart from that, you can put any content in the `(..)`, e.g. math equations.

$$(x + y)^2$$

2.3 `#pause` function

Basic usage of `#pause(..)` is usually in the form `#show: pause`.
Apart from that, you can put any content in the `(..)`, e.g. math equations.

$$(x + y)^2 = (x + y)(x + y)$$

2.3 #pause function

Basic usage of `#pause(..)` is usually in the form `#show: pause`.
Apart from that, you can put any content in the `(..)`, e.g. math equations.

$$\begin{aligned}(x+y)^2 &= (x+y)(x+y)\\ &= x^2+2xy+y^2\end{aligned}$$

2.3 #pause function

Basic usage of `#pause(..)` is usually in the form `#show: pause`.
Apart from that, you can put any content in the `(..)`, e.g. math equations.

$$
\begin{aligned}
(x+y)^2 &= (x+y)(x+y) \\
&= x^2 + 2xy + y^2
\end{aligned}
$$

as from

```
$ (x + y)^2 pause(&= (x + y)(x + y)) \
            pause(&= x^2 + 2 x y + y^2) $
```

2.4 `#fragments` function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively;

2.4 `#fragments` function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively; however, we have a better way.

2.4 `#fragments` function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively; however, we have a better way.

Indroducing `#fragments(..)`:

```
#fragments[+ A][+ B][+ C]
```

2.4 #fragments function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively; however, we have a better way.

Indroducing `#fragments(..)`:

```
#fragments[+ A][+ B][+ C]
```

Output:

1.
2.
3.

2.4 `#fragments` function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively; however, we have a better way.

Indroducing `#fragments(..)`:

```
#fragments[+ A][+ B][+ C]
```

Output:

1. A
2.
3.

2.4 `#fragments` function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively; however, we have a better way.

Indroducing `#fragments(..)`:

```
#fragments[+ A][+ B][+ C]
```

Output:

1. A
2. B
3.

2.4 `#fragments` function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively; however, we have a better way.

Indroducing `#fragments(..)`:

```
#fragments[+ A][+ B][+ C]
```

Output:

1. A
2. B
3. C

2.4 `#fragments` function

Imagine having to type

```
#pause[+ A]
#pause[+ B]
#pause[+ C]
```

to reveal `A` to `C` consecutively; however, we have a better way.

Indroducing `#fragments(..)`:

```
#fragments[+ A][+ B][+ C]
```

Output:

1. A
2. B
3. C

Note: default `#hide` function cannot hide the number or list markers.
To solve this, we will introduce the alternative way to 'hide' them.

2.5 The `#step-item` function

This function was created specifically for step-by-step revealing lists and enums, with ability to hide the markers and numbers.

2.5 The `#step-item` function

This function was created specifically for step-by-step revealing lists and enums, with ability to hide the markers and numbers.

```
#step-item[
    + First Item
    + Second Item
    + Third Item
]
```

Output:

2.5 The `#step-item` function

This function was created specifically for step-by-step revealing lists and enums, with ability to hide the markers and numbers.

```
#step-item[
    + First Item
    + Second Item
    + Third Item
]
```

Output:

1. First Item

2.5 The `#step-item` function

This function was created specifically for step-by-step revealing lists and enums, with ability to hide the markers and numbers.

```
#step-item[
    + First Item
    + Second Item
    + Third Item
]
```

Output:

1. First Item
2. Second Item

2.5 The `#step-item` function

This function was created specifically for step-by-step revealing lists and enums, with ability to hide the markers and numbers.

```
#step-item[
    + First Item
    + Second Item
    + Third Item
]
```

Output:

1. First Item
2. Second Item
3. Third Item

2.5 The `#step-item` function

It can be nested as long as you like.

```
#step-item[
    + First Item
      #step-item[
        - Sub-First
        - Sub-First-Second
      ]
    + Second Item
    + Third Item
]
```

Output:

2.5 The `#step-item` function

It can be nested as long as you like.

```
#step-item[
    + First Item
      #step-item[
        - Sub-First
        - Sub-First-Second
      ]
    + Second Item
    + Third Item
]
```

Output:

1. First Item

2.5 The `#step-item` function

It can be nested as long as you like.

```
#step-item[
    + First Item
    #step-item[
        - Sub-First
        - Sub-First-Second
    ]
    + Second Item
    + Third Item
]
```

Output:

1. First Item
    - Sub-First

2.5 The `#step-item` function

It can be nested as long as you like.

```
#step-item[
    + First Item
    #step-item[
        - Sub-First
        - Sub-First-Second
    ]
    + Second Item
    + Third Item
]
```

Output:

1. First Item
   - Sub-First
   - Sub-First-Second

2.5 The `#step-item` function

It can be nested as long as you like.

```
#step-item[
    + First Item
      #step-item[
        - Sub-First
        - Sub-First-Second
      ]
    + Second Item
    + Third Item
]
```

Output:

```
1.  First Item
      •  Sub-First
      •  Sub-First-Second
2.  Second Item
```

2.5 The `#step-item` function

It can be nested as long as you like.

```
#step-item[
    + First Item
    #step-item[
        - Sub-First
        - Sub-First-Second
    ]
    + Second Item
    + Third Item
]
```

Output:

1. First Item
    - Sub-First
    - Sub-First-Second
2. Second Item
3. Third Item

2.5 The `#step-item` function

It can be nested as long as you like.

```
#step-item[
    + First Item
      #step-item[
        - Sub-First
        - Sub-First-Second
      ]
    + Second Item
    + Third Item
]
```

Output:

1. First Item
    - Sub-First
    - Sub-First-Second
2. Second Item
3. Third Item

It works by modifying `item` input and the markers with varying timeline of `#pause`.

2.6 The `#hider` argument

Every function that can ‘hide’ and reveal content has a named argument called `#hider`. This argument has a default value of Typst’s native `#hide()` function.

2.6 The `#hider` argument

Every function that can ‘hide’ and reveal content has a named argument called `#hider`. This argument has a default value of Typst’s native `#hide()` function.

However, if you want other modes of *hiding* something? E.g. make it *transparent*. You can modify this with `#text.with(fill: gray.transparentize(50%))`:

2.6 The #hider argument

```
#let lg = gray.transparentize(50%)
#let pause = pause.with(hider: text.with(lg))

Hello!
#show: pause

It's gray
```

Output:

```
Hello!

It’s gray
```

2.6 The `#hider` argument

```
#let lg = gray.transparentize(50%)
#let pause = pause.with(hider: text.with(lg))

Hello!
#show: pause

It's gray
```

Output:

Hello!

It’s gray

2.7 `#only` and `#uncover`

So far, `#pause` and `#fragments` examples only show you to reveal the content *step-by-step*. How about *absolutely* reveal content? Say, at a given number of frames?

2.7 `#only` and `#uncover`

So far, `#pause` and `#fragments` examples only show you to reveal the content *step-by-step*. How about *absolutely* reveal content? Say, at a given number of frames?

> A **frame** or **subslide** is a page that contains fragments of slides' content, so that when all pages are viewed consecutively, we can see the *change* of content.

2.7 `#only` and `#uncover`

So far, `#pause` and `#fragments` examples only show you to reveal the content *step-by-step*. How about *absolutely* reveal content? Say, at a given number of frames?

> A **frame** or **subslide** is a page that contains fragments of slides' content, so that when all pages are viewed consecutively, we can see the *change* of content.

For a more complex animation, `#only` and `#uncover` functions can control when the content will be shown based on given number of frames, or *subslide number*.

2.7 #only and #uncover

```
Content Before

#only(2, 4)[
  This is _only_ shown on
  subslide 2 and 4.
]

Content After
```

Output: on subslide 1

Content Before

Content After

2.7 #only and #uncover

```
Content Before

#only(2, 4)[
  This is _only_ shown on
  subslide 2 and 4.
]

Content After
```

Output: on subslide 2

Content Before

This is *only* shown on subslide 2 and 4.

Content After

`#only(..n, body)` shows the #body *only* at the given subslide numbers #n.
For other frames, the content is vanished, with no preserved space.

2.7 `#only` and `#uncover`

```
Content Before

#only(2, 4)[
  This is _only_ shown on
  subslide 2 and 4.
]

Content After
```

Output: on subslide 3

```
Content Before

Content After
```

`#only(..n, body)` shows the `#body` *only* at the given subslide numbers `#n`.
For other frames, the content is vanished, with no preserved space.

2.7 `#only` and `#uncover`

```
Content Before

#only(2, 4)[
  This is _only_ shown on
  subslide 2 and 4.
]

Content After
```

Output: on subslide 4

Content Before

This is *only* shown on subslide 2 and 4.

Content After

`#only(..n, body)` shows the `#body` *only* at the given subslide numbers `#n`.
For other frames, the content is vanished, with no preserved space.

2.7 #only and #uncover

```
Content Before

#uncover(2, from: 4)[
  This is _uncovered_ on
  subslide 2 and 4 onwards.
]

Content After
```

Output: on subslide 1

Content Before

Content After

2.7 #only and #uncover

```
Content Before

#uncover(2, from: 4)[
    This is _uncovered_ on
    subslide 2 and 4 onwards.
]

Content After
```

Output: on subslide 2

```
Content Before

This is uncovered on subslide 2 and 4
onwards.

Content After
```

`#uncover(..n, from: int, body)` uncovers the `#body` in the same condition as `#only`, with an exception of having *space preserved*.

2.7 `#only` and `#uncover`

```
Content Before

#uncover(2, from: 4)[
    This is _uncovered_ on
    subslide 2 and 4 onwards.
]

Content After
```

Output: on subslide 3

Content Before

Content After

`#uncover(..n, from: int, body)` uncovers the `#body` in the same condition as `#only`, with an exception of having *space preserved*.

2.7 #only and #uncover

```
Content Before

#uncover(2, from: 4)[
  This is _uncovered_ on subslide 2 and 4 onwards.
]

Content After
```

Output: on subslide 4

Content Before

This is *uncovered* on subslide 2 and 4 onwards.

Content After

`#uncover(..n, from: int, body)` uncovers the `#body` in the same condition as `#only`, with an exception of having *space preserved*.

2.7 `#only` and `#uncover`

If you noticed the last example carefully, you will see the argument `#from` being introduced in the `#uncover(from: int, ..)`.

2.7 `#only` and `#uncover`

If you noticed the last example carefully, you will see the argument `#from` being introduced in the `#uncover(from: int, ..)`.

Both `#only` and `#uncover` can take the `#from` as integer to start revealing the content only after that subslide number `#from`.

2.7 `#only` and `#uncover`

If you noticed the last example carefully, you will see the argument `#from` being introduced in the `#uncover(from: int, ..)`.

Both `#only` and `#uncover` can take the `#from` as integer to start revealing the content only after that subslide number `#from`.

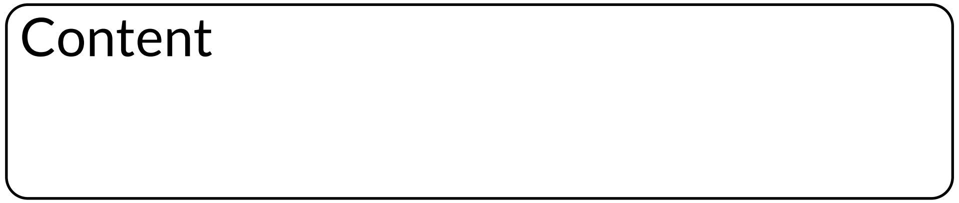
Not only integers can you use as subslide number, `#auto` and `#none` also can be used. What do they do?

2.8 Relative Indices

If you want to reveal a yellow box once in a frame after some stream of content, say the following code:

```
Content #show: pause; Content

#uncover(3, rect(
  fill: yellow, [BOX]
))
```

Output: on subslide 1


Content

2.8 Relative Indices

If you want to reveal a yellow box once in a frame after some stream of content, say the following code:

```
Content #show: pause; Content

#uncover(3, rect(
  fill: yellow, [BOX]
))
```

Output: on subslide 2

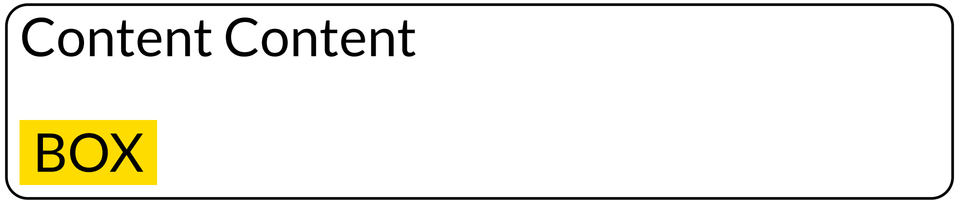
Content Content

2.8 Relative Indices

If you want to reveal a yellow box once in a frame after some stream of content, say the following code:

```
Content #show: pause; Content

#uncover(3, rect(
  fill: yellow, [BOX]
))
```

Output: on subslide 3



```
Content Content

BOX
```

You must know the current number of `#pause`s to determine the subslide number where the `BOX` must be shown.

2.8 Relative Indices

If you want to reveal a yellow box once in a frame after some stream of content, say the following code:

```
Content #show: pause; Content

#uncover(3, rect(
  fill: yellow, [BOX]
))
```

Output: on subslide 4

```
Content Content
```

You must know the current number of `#pause`s to determine the subslide number where the `BOX` must be shown. Is there an alternative? *Yes: Relative Indices*

2.8 Relative Indices

> **Index** (plural: Indices) is subslide number.

Index specified in `#uncover`, `#only`, and other arguments that requires it has 2 types:

2.8 Relative Indices

> **Index** (plural: Indices) is subslide number.

Index specified in `#uncover`, `#only`, and other arguments that requires it has 2 types:

1. **Absolute** index: the actual integer subslide number, and

2.8 Relative Indices

Index (plural: Indices) is subslide number.

Index specified in `#uncover`, `#only`, and other arguments that requires it has 2 types:

1. **Absolute** index: the actual integer subslide number, and
2. **Relative** index: `#auto`, `#none` and `(rel: int)`, relative to *number of pauses*

2.8 Relative Indices

> **Index** (plural: Indices) is subslide number.

Index specified in `#uncover`, `#only`, and other arguments that requires it has 2 types:

1. **Absolute** index: the actual integer subslide number, and
2. **Relative** index: `#auto`, `#none` and `(rel: int)`, relative to *number of pauses*
   - `#auto` means index *after* the current number of pauses.

2.8 Relative Indices

> **Index** (plural: Indices) is subslide number.

Index specified in `#uncover`, `#only`, and other arguments that requires it has 2 types:

1. **Absolute** index: the actual integer subslide number, and
2. **Relative** index: `#auto`, `#none` and `(rel: int)`, relative to *number of pauses*
   - `#auto` means index *after* the current number of pauses.
   - `#none` means index *as same as* the current number of pauses.

2.8 Relative Indices

> **Index** (plural: Indices) is subslide number.

Index specified in `#uncover`, `#only`, and other arguments that requires it has 2 types:

1. **Absolute** index: the actual integer subslide number, and
2. **Relative** index: `#auto`, `#none` and `(rel: int)`, relative to *number of pauses*
    - `#auto` means index *after* the current number of pauses.
    - `#none` means index *as same as* the current number of pauses.
    - `#(rel: int)` means index that is `int` subslides away from the current number of pauses. (e.g. `#(rel: 1)` is equivalent to `#auto`).

2.8 Relative Indices

Example: Uncover the yellow box on subslide 5 and after current pauses state, together with only show X on the same subslide as the current pauses.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
))

After Content #only(none, [X])
```

Output: on subslide 1

Content

2.8 Relative Indices

Example: Uncover the yellow box on subslide 5 and after current pauses state, together with only show X on the same subslide as the current pauses.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
))

After Content #only(none, [X])
```

Output: on subslide 2

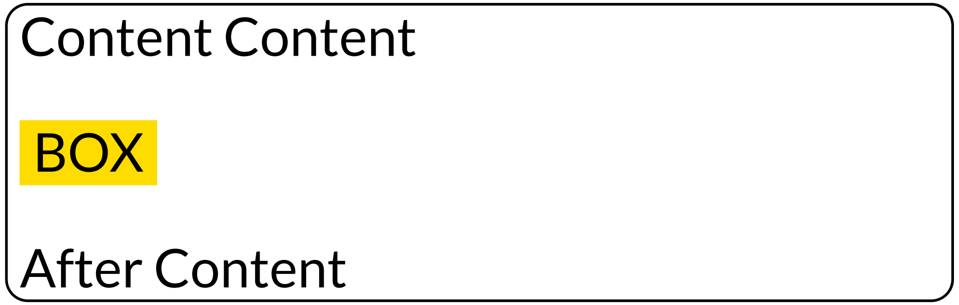
```
Content Content



After Content X
```

2.8 Relative Indices

Example: Uncover the yellow box on subslide 5 and after current pauses state, together with only show X on the same subslide as the current pauses.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
))

After Content #only(none, [X])
```

Output: on subslide 3



Content Content

BOX

After Content

2.8 Relative Indices

Example: Uncover the yellow box on subslide 5 and after current pauses state, together with only show X on the same subslide as the current pauses.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
))

After Content #only(none, [X])
```

Output: on subslide 4

```
Content Content



After Content
```

2.8 Relative Indices

Example: Uncover the yellow box on subslide 5 and after current pauses state, together with only show X on the same subslide as the current pauses.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
))

After Content #only(none, [X])
```

Output: on subslide 5

```
Content Content

BOX

After Content
```

2.9 Varying Timeline

If you look at the last example carefully, it is noticeable that when `After Content` appears, it follows the `#show: pause` function, as if there where no `#uncover` in between.

2.9 Varying Timeline

If you look at the last example carefully, it is noticeable that when `After Content` appears, it follows the `#show: pause` function, as if there where no `#uncover` in between.

However, what if we want to reveal some content afterwards, after every animation, without the need of specifying the subslide number?

2.9 Varying Timeline

If you look at the last example carefully, it is noticeable that when `After Content` appears, it follows the `#show: pause` function, as if there where no `#uncover` in between.

However, what if we want to reveal some content afterwards, after every animation, without the need of specifying the subslide number?

If only the `#pause` 'sees' the `#uncover`'s presence, it would be good, right?

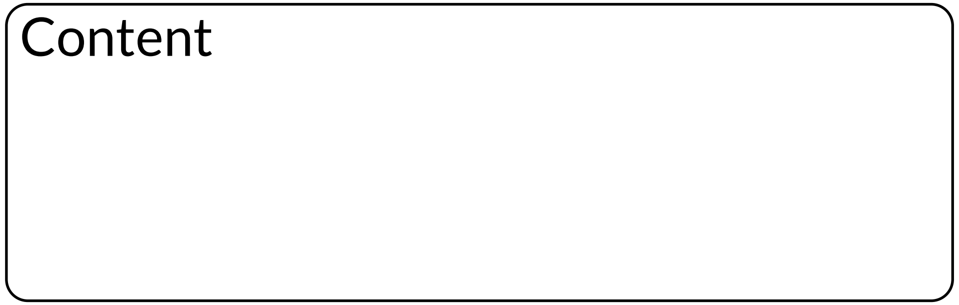
Yes, it can, by set the argument `#uncover(update-pause: true)`.

2.9 Varying Timeline

Example: The yellow box is revealed on subslide 5 and after the current pauses, with `After Content` appears after every animation.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
), update-pause: true)

#pause[After Content]
```

Output: on subslide 1



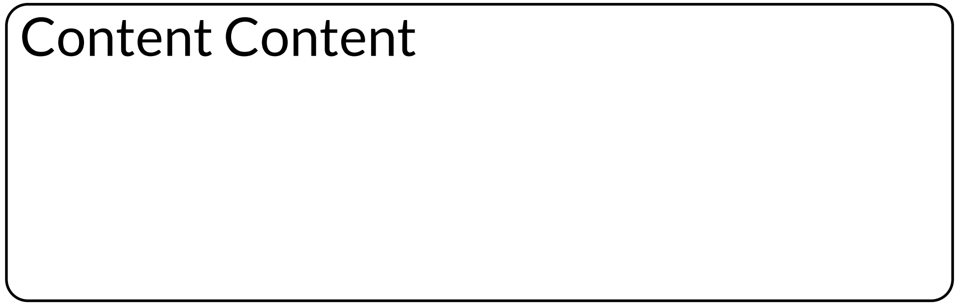
Content

2.9 Varying Timeline

Example: The yellow box is revealed on subslide 5 and after the current pauses, with `After Content` appears after every animation.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
), update-pause: true)

#pause[After Content]
```

Output: on subslide 2



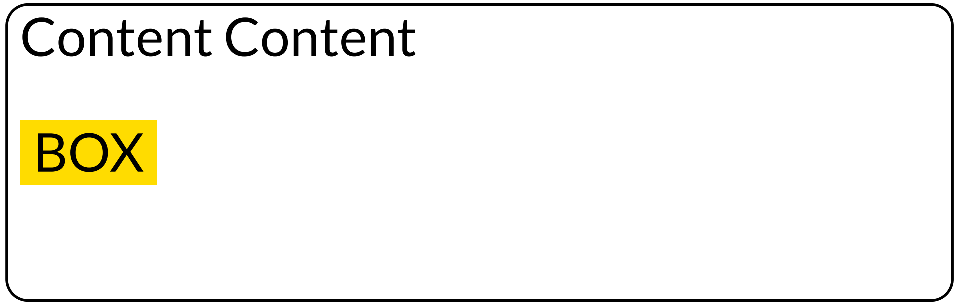
Content Content

2.9 Varying Timeline

Example: The yellow box is revealed on subslide 5 and after the current pauses, with `After Content` appears after every animation.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
), update-pause: true)

#pause[After Content]
```

Output: on subslide 3


Content Content

BOX

2.9 Varying Timeline

Example: The yellow box is revealed on subslide 5 and after the current pauses, with `After Content` appears after every animation.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
), update-pause: true)

#pause[After Content]
```

Output: on subslide 4

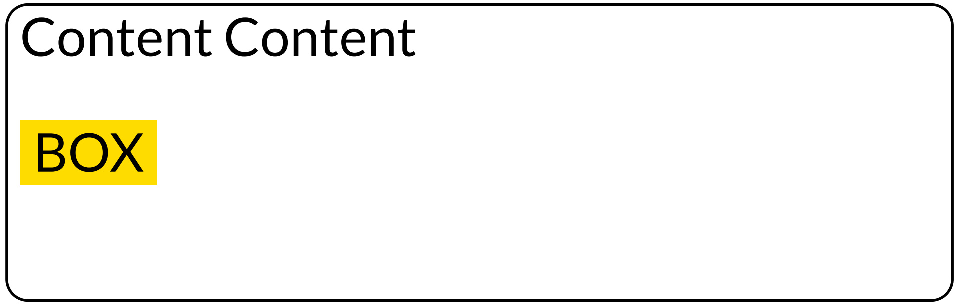
Content Content

2.9 Varying Timeline

Example: The yellow box is revealed on subslide 5 and after the current pauses, with `After Content` appears after every animation.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
), update-pause: true)

#pause[After Content]
```

Output: on subslide 5


Content Content

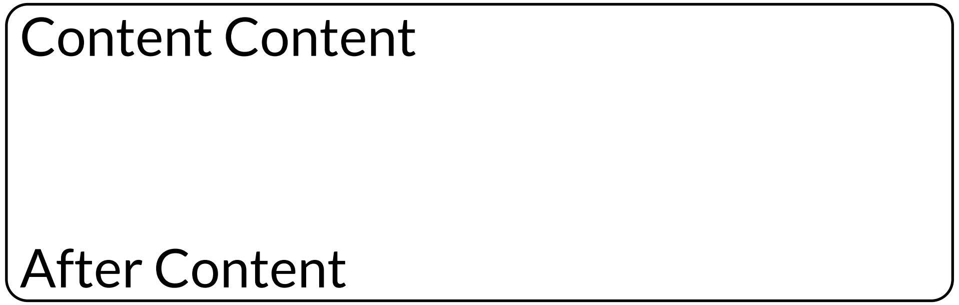
BOX

2.9 Varying Timeline

Example: The yellow box is revealed on subslide 5 and after the current pauses, with `After Content` appears after every animation.

```
Content #show: pause; Content

#uncover(auto, 5, rect(
  fill: yellow, [BOX]
), update-pause: true)

#pause[After Content]
```

Output: on subslide 6



Content Content

After Content

2.9 Varying Timeline

`#update-pause` argument updates the current pauses to the maxium index. In the example, `#auto` resolves to 3, so 5 is the maximum.

2.9 Varying Timeline

`#update-pause` argument updates the current pauses to the maxium index. In the example, `#auto` resolves to 3, so 5 is the maximum.

Both `#only` and `#uncover` have `#update-pause` argument, but they are set to be `#false` by default. So these functions reveal the content *independently* from `#pause(..)`.

2.9 Varying Timeline

`#update-pause` argument updates the current pauses to the maxium index. In the example, `#auto` resolves to 3, so 5 is the maximum.

Both `#only` and `#uncover` have `#update-pause` argument, but they are set to be `#false` by default. So these functions reveal the content *independently* from `#pause(..)`.

However, the ability to affect the `#pause(..)` progress unlocks one powerful key:

2.9 Varying Timeline

`#update-pause` argument updates the current pauses to the maxium index. In the example, `#auto` resolves to 3, so 5 is the maximum.

Both `#only` and `#uncover` have `#update-pause` argument, but they are set to be `#false` by default. So these functions reveal the content *independently* from `#pause(..)`.

However, the ability to affect the `#pause(..)` progress unlocks one powerful key:

Number of pauses can be varied and independent from actual number of `#pause`s.

If we use `#only` or `#uncover` to change them, for example:

2.9 Varying Timeline

Content can be revealed parallel on side by side.

```
#grid(columns: (1fr, 1fr))[
  First \ #show: pause;
  A #show: pause; B
][ // `[]` is a dummy content.
  #uncover(1, [], update-pause: true)
  Second \ #show: pause;
  A #show: pause; B
]
```

Output: on subslide 1

```
First                 Second
```

The content on both columns are shown synchronously, because the pauses are *set* to 1 (first subslide) by `#uncover`.

2.9 Varying Timeline

Content can be revealed parallel on side by side.

```
#grid(columns: (1fr, 1fr))[
  First \ #show: pause;
  A #show: pause; B
][ // `[]` is a dummy content.
  #uncover(1, [], update-pause: true)
  Second \ #show: pause;
  A #show: pause; B
]
```

Output: on subslide 2

First	Second
A	A

The content on both columns are shown synchronously, because the pauses are *set* to 1 (first subslide) by `#uncover`.

2.9 Varying Timeline

Content can be revealed parallel on side by side.

```
#grid(columns: (1fr, 1fr))[
  First \ #show: pause;
  A #show: pause; B
][ // `[]` is a dummy content.
  #uncover(1, [], update-pause: true)
  Second \ #show: pause;
  A #show: pause; B
]
```

Output: on subslide 3

First	Second
A B	A B

The content on both columns are shown synchronously, because the pauses are *set* to 1 (first subslide) by `#uncover`.

2.9 Varying Timeline

Moreover, you can set some content to be shown *before* one another by using *negative relative indices* such as `#(rel: -1)` in the example.

```
#step-item[
    - First
    - Second
    - Third
]
#only((rel: -1), [Second Too!],
update-pause: true)
#show: pause; After Second.
```

Output: on subslide 1

So the `After Second` is revealed at the same time as the last item.

2.9 Varying Timeline

Moreover, you can set some content to be shown *before* one another by using *negative relative indices* such as `#(rel: -1)` in the example.

```
#step-item[
    - First
    - Second
    - Third
]
#only((rel: -1), [Second Too!],
update-pause: true)
#show: pause; After Second.
```

Output: on subslide 2

- First

So the `After Second` is revealed at the same time as the last item.

2.9 Varying Timeline

Moreover, you can set some content to be shown *before* one another by using *negative relative indices* such as `#(rel: -1)` in the example.

```
#step-item[
    - First
    - Second
    - Third
]
#only((rel: -1), [Second Too!],
update-pause: true)
#show: pause; After Second.
```

Output: on subslide 3

- First
- Second

Second Too!

So the `After Second` is revealed at the same time as the last item.

2.9 Varying Timeline

Moreover, you can set some content to be shown *before* one another by using *negative relative indices* such as `#(rel: -1)` in the example.

```
#step-item[
    - First
    - Second
    - Third
]
#only((rel: -1), [Second Too!], update-pause: true)
#show: pause; After Second.
```

Output: on subslide 4

- First
- Second
- Third

After Second.

So the `After Second` is revealed at the same time as the last item.

2.10 Animated Decorations

Most of the functions we provide up until now can only create animations of hiding and showing stuff. How about *changing* its appearance? e.g. color?

2.10 Animated Decorations

Most of the functions we provide up until now can only create animations of hiding and showing stuff. How about *changing* its appearance? e.g. color?

You can emphasize your words by using `#alert` like in this sentence.
`#alert` can alert the audience by wrapping the input with its `#func` argument, which is `#emph` function by default.

2.10 Animated Decorations

Most of the functions we provide up until now can only create animations of hiding and showing stuff. How about *changing* its appearance? e.g. color?

You can *emphasize* your words by using `#alert` like in this sentence.
`#alert` can *alert* the audience by wrapping the input with its `#func` argument, which is `#emph` function by default.

```
Please #alert(auto)[FOCUS] me
and
#alert(auto, func: text.with(red))[Warn]
them.
```

Output:

Please FOCUS me
and Warn them.

2.10 Animated Decorations

Most of the functions we provide up until now can only create animations of hiding and showing stuff. How about *changing* its appearance? e.g. color?

You can emphasize your words by using `#alert` like in this sentence. `#alert` can alert the audience by wrapping the input with its `#func` argument, which is `#emph` function by default.

```
Please #alert(auto)[FOCUS] me
and
#alert(auto, func: text.with(red))[Warn]
them.
```

Output:

Please *FOCUS* me and Warn them.

2.10 Animated Decorations

Another functions for creating multiple *alerts* is called `#transform`.
This function wraps the content and change its through a series of functions.

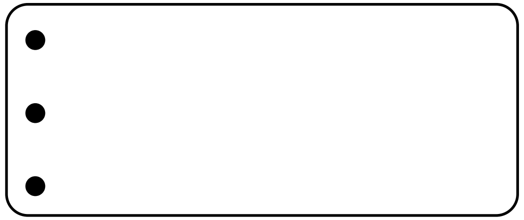
2.10 Animated Decorations

Another functions for creating multiple *alerts* is called `#transform`.
This function wraps the content and change its through a series of functions.

This is very useful for creating step-by-step list alerts or make the content dynamically changing its appearance. For example,

```
#let no(body) = body // original apperance
#let yes(body) = text(fill: red, body)
#transform([- First Item], yes, no)
#transform(start: none, [- Second Item], yes, no)
#transform(start: none, [- Third Item], yes, no)
```

Output:



2.10 Animated Decorations

Another functions for creating multiple *alerts* is called `#transform`.
This function wraps the content and change its through a series of functions.

This is very useful for creating step-by-step list alerts or make the content dynamically changing its appearance. For example,

```
#let no(body) = body // original apperance
#let yes(body) = text(fill: red, body)
#transform([- First Item], yes, no)
#transform(start: none, [- Second Item], yes, no)
#transform(start: none, [- Third Item], yes, no)
```

Output:

```
• First Item
•
•
```

2.10 Animated Decorations

Another functions for creating multiple *alerts* is called `#transform`.
This function wraps the content and change its through a series of functions.

This is very useful for creating step-by-step list alerts or make the content dynamically changing its appearance. For example,

```
#let no(body) = body // original apperance
#let yes(body) = text(fill: red, body)
#transform([- First Item], yes, no)
#transform(start: none, [- Second Item], yes, no)
#transform(start: none, [- Third Item], yes, no)
```

Output:

- First Item
- Second Item
-

2.10 Animated Decorations

Another functions for creating multiple *alerts* is called `#transform`.
This function wraps the content and change its through a series of functions.

This is very useful for creating step-by-step list alerts or make the content dynamically changing its appearance. For example,

```
#let no(body) = body // original apperance
#let yes(body) = text(fill: red, body)
#transform([- First Item], yes, no)
#transform(start: none, [- Second Item], yes, no)
#transform(start: none, [- Third Item], yes, no)
```

Output:

- First Item
- Second Item
- Third Item

2.10 Animated Decorations

Another functions for creating multiple *alerts* is called `#transform`.
This function wraps the content and change its through a series of functions.

This is very useful for creating step-by-step list alerts or make the content dynamically changing its appearance. For example,

```
#let no(body) = body // original apperance
#let yes(body) = text(fill: red, body)
#transform([- First Item], yes, no)
#transform(start: none, [- Second Item], yes, no)
#transform(start: none, [- Third Item], yes, no)
```

Output:

- First Item
- Second Item
- Third Item

2.10 Animated Decorations

~~~
#transform(
  codly(highlighted-lines: (1,)),
  codly(highlighted-lines: (4,)) )
```python
n = input("Number: ")
n = int(n)
for i in range(n):
  print("Hello World!")
```
~~~

You can use this to highlight different lines of code with Codly[5]. Output: on subslide 1

```
1  n = input("Number: ")
2  n = int(n)
3  for i in range(n):
4    print("Hello World!")
```

[5] https://typst.app/universe/package/codly/

2.10 Animated Decorations

~~~
#transform(
  codly(highlighted-lines: (1,)),
  codly(highlighted-lines: (4,)) )
```python
n = input("Number: ")
n = int(n)
for i in range(n):
  print("Hello World!")
```
~~~

You can use this to highlight different lines of code with Codly[5]. Output: on subslide 2

```
1  n = input("Number: ")
2  n = int(n)
3  for i in range(n):
4    print("Hello World!")
```

[5] https://typst.app/universe/package/codly/

2.11 The `#motion` workspace

I have dreamed about having an interface to modify the flow of animations as in Powerpoint or Keynote. This is the first attempt of how to do it, see the following example:

```
#motion(s => [#table(
    columns: (1fr, 1fr),
    tag(s, "A", [Ant]),
    tag(s, "B", [Bird]),
    tag(s, "C", [Cat]),
    tag(s, "D", [Dog])
)],
    controls: (
      "A",
      ("A", "B"),
      "C.start",
      "D",
      ("C.stop", "B")
))
```

2.11 The `#motion` workspace

The `#motion` function accepts a callback `s => [..]` where `[..]` can be any content.

Inside the callback function, the variable `s` called the *status* is read by the *named* group of content indicated by their `#tag` function. This function has the following anatomy:

```
#tag(/* status */, /* ID */, /* body */)
```

where `ID` means the name of the group. It should be a unique string that you call this group of content, and `body` is the content to be “tagged”.

2.11 The `#motion` workspace

Therefore, with the `#tag` function, the content has its own name. The sequence of displaying each content can be controlled precisely by specifying the `controls` argument of the `#motion` function using *their names*.

Just like moving around the control panel in Powerpoint.

Let’s see the result and how to control the sequence of these contents.

2.11 The #motion workspace

The `controls` argument must receive an array that specifies displaying sequence of showing the content, which can be

- A single name, like `"A"`, to show the content once.
- An array of names, like `("A", "B")` in line 10, to show both `"A"` and `"B"` once.
- Name + `".start"` to *start* showing the content, and
- Name + `".stop"` to *stop* showing (hide) the content.

Output: on subslide 1

Ant	

2.11 The `#motion` workspace

The `controls` argument must receive an array that specifies displaying sequence of showing the content, which can be

- A single name, like `"A"`, to show the content once.
- An array of names, like `("A", "B")` in line 10, to show both `"A"` and `"B"` once.
- Name + `".start"` to *start* showing the content, and
- Name + `".stop"` to *stop* showing (hide) the content.

Output: on subslide 2

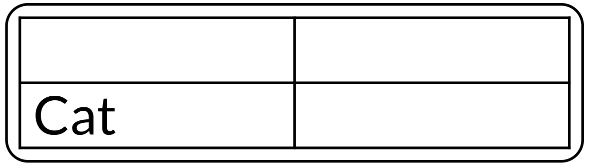
Ant	Bird

2.11 The `#motion` workspace

The `controls` argument must receive an array that specifies displaying sequence of showing the content, which can be

- A single name, like `"A"`, to show the content once.
- An array of names, like `("A", "B")` in line 10, to show both `"A"` and `"B"` once.
- Name + `".start"` to *start* showing the content, and
- Name + `".stop"` to *stop* showing (hide) the content.

Output: on subslide 3



Cat	

2.11 The `#motion` workspace

The `controls` argument must receive an array that specifies displaying sequence of showing the content, which can be

- A single name, like `"A"`, to show the content once.
- An array of names, like `("A", "B")` in line 10, to show both `"A"` and `"B"` once.
- Name + `".start"` to *start* showing the content, and
- Name + `".stop"` to *stop* showing (hide) the content.

Output: on subslide 4

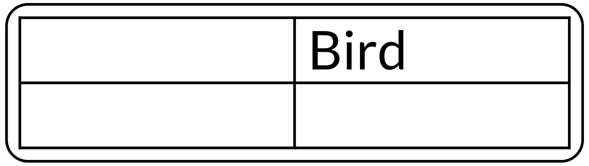
Cat	Dog

2.11 The `#motion` workspace

The `controls` argument must receive an array that specifies displaying sequence of showing the content, which can be

- A single name, like `"A"`, to show the content once.
- An array of names, like `("A", "B")` in line 10, to show both `"A"` and `"B"` once.
- Name + `".start"` to *start* showing the content, and
- Name + `".stop"` to *stop* showing (hide) the content.

Output: on subslide 5

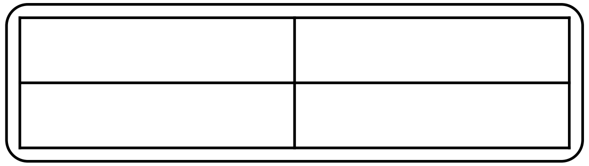


2.11 The `#motion` workspace

The `controls` argument must receive an array that specifies displaying sequence of showing the content, which can be

- A single name, like `"A"`, to show the content once.
- An array of names, like `("A", "B")` in line 10, to show both `"A"` and `"B"` once.
- Name + `".start"` to *start* showing the content, and
- Name + `".stop"` to *stop* showing (hide) the content.

Output: on subslide 6



Note that, you can select your own `#hider` in each `#tag` or all in `#motion` function. See next:

2.11 The `#motion` workspace

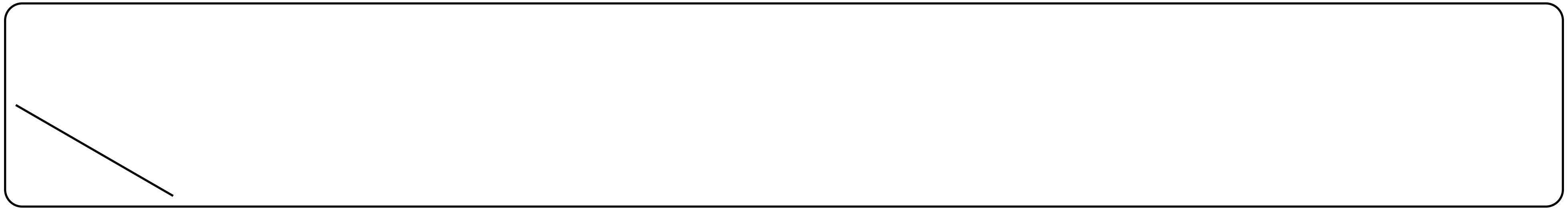
```
#import "@preview/cetz:0.4.2": canvas, draw
#motion(s => [
  #canvas({ import draw: *; scale(3)
    tag(s, "arc", arc((0, 0), start: 30deg, stop: 150deg, name: "R"))
    tag(s, "line1", line("R.start", "R.origin"))
    tag(s, "line2", line("R.end", "R.origin"))
  })
], hider: draw.hide.with(bounds: true), controls: (
  "line2.start", "line1.start", "arc.start"
))
```

2.11 The `#motion` workspace

The result is shown here. In this example, the `#hider` used is native `#draw.hide` of CeTZ module, so the elements can be covered compatibly.

With the `controls` argument, **the content can be shown regardless of their definition order** in the source code, like drawing this fan shape.

Output: on subslide 1

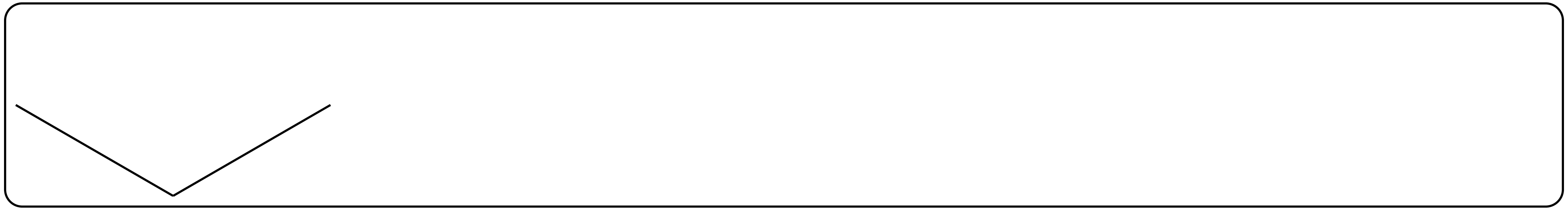


2.11 The `#motion` workspace

The result is shown here. In this example, the `#hider` used is native `#draw.hide` of CeTZ module, so the elements can be covered compatibly.

With the `controls` argument, **the content can be shown regardless of their definition order** in the source code, like drawing this fan shape.

Output: on subslide 2

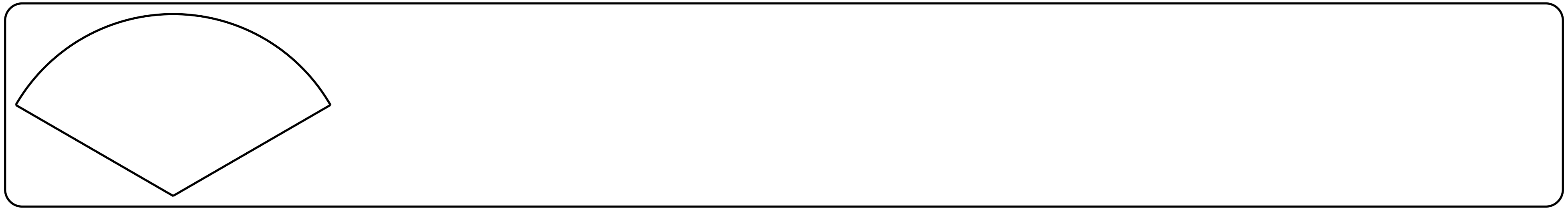


2.11 The `#motion` workspace

The result is shown here. In this example, the `#hider` used is native `#draw.hide` of CeTZ module, so the elements can be covered compatibly.

With the `controls` argument, **the content can be shown regardless of their definition order** in the source code, like drawing this fan shape.

Output: on subslide 3



2.12 Rendering Stuffs

Here comes the most powerful, but most complex utilization of Presentate: `#render` function and `#animation` module.

2.12 Rendering Stuffs

Here comes the most powerful, but most complex utilization of Presentate: `#render` function and `#animation` module.

As we have told, package integration on presentation animation is sometimes tricky, as they are not happy with `content` input data.

2.12 Rendering Stuffs

Here comes the most powerful, but most complex utilization of Presentate: `#render` function and `#animation` module.

As we have told, package integration on presentation animation is sometimes tricky, as they are not happy with `content` input data.

So presentate provides a *workspace* for rendering stuffs that are not necessary to be in content type, with *non-content updates* for number of frames needed.

2.12 Rendering Stuffs

Here comes the most powerful, but most complex utilization of Presentate: `#render` function and `#animation` module.

As we have told, package integration on presentation animation is sometimes tricky, as they are not happy with `content` input data.

So presentate provides a *workspace* for rendering stuffs that are not necessary to be in content type, with *non-content updates* for number of frames needed.

So you can focus on the animation, without worrying about number of subslides.

2.12 Rendering Stuffs

Structure of `#render`

```
#render(s => ({
  import animation: *
  // your stuff goes here.
}, s))
```

2.12 Rendering Stuffs

Structure of #render

```
#render(s => ({
  import animation: *
  // your stuff goes here.
}, s))
```

#render only accepts one positional argument: **a function.**

2.12 Rendering Stuffs

Structure of #render

```
#render(s => ({
    import animation: *
    // your stuff goes here.
}, s))
```

#render only accepts one positional argument: **a function**.

This function accepts the current animation states, and returns *an array*, of length 2 which

- **first** member is the shown output,
- **second** member is the updated states.

2.12 Rendering Stuffs

Structure of `#render`

```
#render(s => ({
  import animation: *
  // your stuff goes here.
}, s))
```

`#render` only accepts one positional argument: **a function**.

This function accepts the current animation states, and returns *an array*, of length 2 which

- **first** member is the shown output,
- **second** member is the updated states.

This way, Presentate can both show your output, and update the states, so the other elements on the slide react automatically.

2.12 Rendering Stuffs

The first member's area only accepts `content`, intended for updating internal states.

However, to create animation with `#render` without generating `content` during the way, Presentate provides the same set of functionality like `#pause`, `#only`, `#fragments`, `#alert`, `#uncover`, and so on, with some key differences:

2.12 Rendering Stuffs

The first member's area only accepts `content`, intended for updating internal states.

However, to create animation with `#render` without generating `content` during the way, Presentate provides the same set of functionality like `#pause`, `#only`, `#fragments`, `#alert`, `#uncover`, and so on, with some key differences:

1. These functions must be imported from `#animation` module.

2.12 Rendering Stuffs

The first member's area only accepts `content`, intended for updating internal states.

However, to create animation with `#render` without generating `content` during the way, Presentate provides the same set of functionality like `#pause`, `#only`, `#fragments`, `#alert`, `#uncover`, and so on, with some key differences:

1. These functions must be imported from `#animation` module.
2. The functions will always accepts the *state* (`#s`) as first positional argument.

2.12 Rendering Stuffs

The first member's area only accepts `content`, intended for updating internal states.

However, to create animation with `#render` without generating `content` during the way, Presentate provides the same set of functionality like `#pause`, `#only`, `#fragments`, `#alert`, `#uncover`, and so on, with some key differences:

1. These functions must be imported from `#animation` module.
2. The functions will always accepts the *state* (`#s`) as first positional argument.
3. **You have to update the state variable (`#s`) manually**.

2.12 Rendering Stuffs

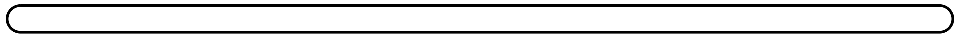
Example 1: Animated CeTZ[6] diagram. Create an animation drawing two circles, in green and red.

```
#import "@preview/cetz:0.4.2": canvas, draw
#render(s => ({
  import animation: *
  canvas({
    import draw: *
    pause(s, circle((0, 0), fill: green,))
    s.push(auto) // update s
    pause(s, circle((1, 0), fill: red))
  })
}, s))
```

[6] https://typst.app/universe/package/cetz

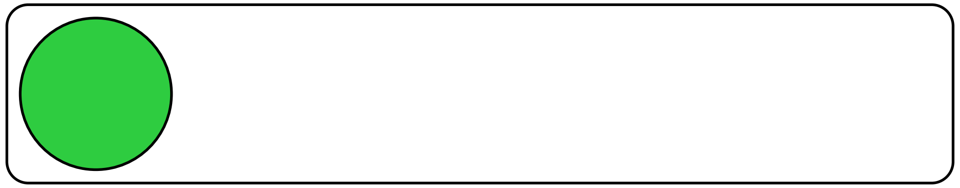
2.12 Rendering Stuffs

Output: on subslide 1



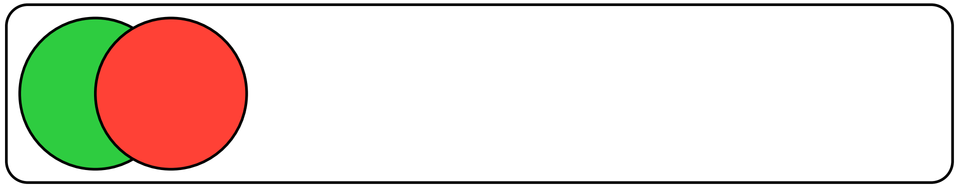
2.12 Rendering Stuffs

Output: on subslide 2



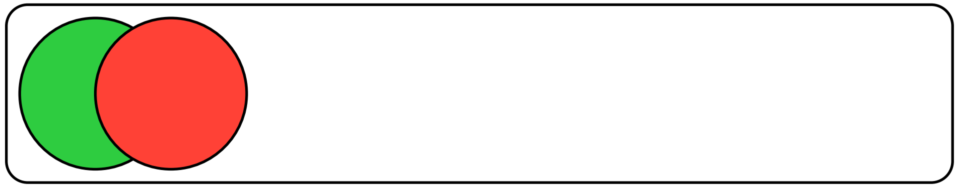
2.12 Rendering Stuffs

Output: on subslide 3



2.12 Rendering Stuffs

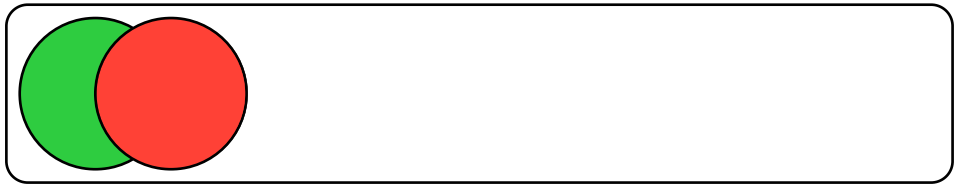
Output: on subslide 4



The default hider of `animation.pause` is `it => none`, so it *does not* preserve space.

2.12 Rendering Stuffs

Output: on subslide 5



The default hider of `animation.pause` is `it => none`, so it *does not* preserve space.

However, you can change this by the `#draw.hide.with(bounds: true)` from native CeTZ to preserve space, by adding the following line before `#canvas`:

```
let pause = pause.with(hider: draw.hide.with(bounds: true))
```

Similarly, you can change the default hider functions to suit your package.

2.12 Rendering Stuffs

You can change the default `#hider` by using `#settings` functions, which will return a dictionary containing the functions:

```
// import "@preview/cetz:0.4.2": canvas, draw
let (uncover, pause) = settings(hider: draw.hide.with(bounds: true))
```

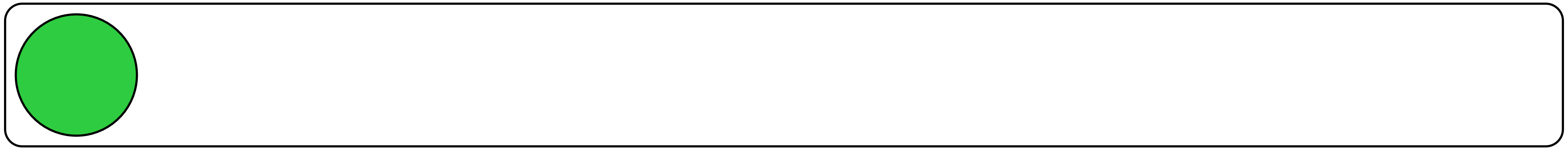
For this change, the last example would become the Output:

2.12 Rendering Stuffs

You can change the default #hider by using #settings functions, which will return a dictionary containing the functions:

```
// import "@preview/cetz:0.4.2": canvas, draw
let (uncover, pause) = settings(hider: draw.hide.with(bounds: true))
```

For this change, the last example would become the Output:

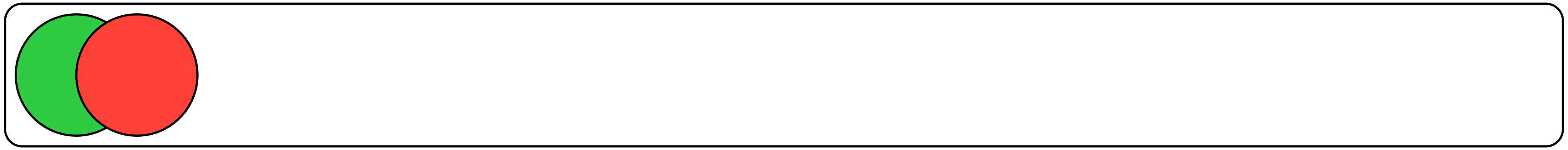


2.12 Rendering Stuffs

You can change the default `#hider` by using `#settings` functions, which will return a dictionary containing the functions:

```
// import "@preview/cetz:0.4.2": canvas, draw
let (uncover, pause) = settings(hider: draw.hide.with(bounds: true))
```

For this change, the last example would become the Output:



2.12 Rendering Stuffs

Updating States: In render function, the state variable `#s` is the sole information about the number of subslides needed to render all of the animations.

2.12 Rendering Stuffs

Updating States: In render function, the state variable `#s` is the sole information about the number of subslides needed to render all of the animations.

So updating it is crucial to produce the correct number of subslides. But how?

2.12 Rendering Stuffs

Updating States: In render function, the state variable `#s` is the sole information about the number of subslides needed to render all of the animations.

So updating it is crucial to produce the correct number of subslides. But how?

The state variable `s` is an *array*, so updating it is basically *push* the new information to it. The infomation added determine the current animation states as

2.12 Rendering Stuffs

Updating States: In render function, the state variable `#s` is the sole information about the number of subslides needed to render all of the animations.

So updating it is crucial to produce the correct number of subslides. But how?

The state variable `s` is an *array*, so updating it is basically *push* the new information to it. The infomation added determine the current animation states as

- `#auto` is pushed to **increase the number of pauses by 1.**

2.12 Rendering Stuffs

Updating States: In render function, the state variable `#s` is the sole information about the number of subslides needed to render all of the animations.

So updating it is crucial to produce the correct number of subslides. But how?

The state variable `s` is an *array*, so updating it is basically *push* the new information to it. The infomation added determine the current animation states as

- `#auto` is pushed to **increase the number of pauses by 1.**
- `(rel: int)` relative index is pushed to **modify the number of pauses by `int`**.

2.12 Rendering Stuffs

Updating States: In render function, the state variable `#s` is the sole information about the number of subslides needed to render all of the animations.

So updating it is crucial to produce the correct number of subslides. But how?

The state variable `s` is an *array*, so updating it is basically *push* the new information to it. The infomation added determine the current animation states as

- `#auto` is pushed to **increase the number of pauses by 1.**
- `(rel: int)` relative index is pushed to **modify the number of pauses by `int`**.
- `1, 2, 3, ..` integers are pushed to set the **current number of pauses**.

2.12 Rendering Stuffs

Updating States: In render function, the state variable `#s` is the sole information about the number of subslides needed to render all of the animations.

So updating it is crucial to produce the correct number of subslides. But how?

The state variable `s` is an *array*, so updating it is basically *push* the new information to it. The infomation added determine the current animation states as

- `#auto` is pushed to **increase the number of pauses by 1.**
- `(rel: int)` relative index is pushed to **modify the number of pauses by `int`**.
- `1, 2, 3, ..` integers are pushed to set the **current number of pauses**.
- `(1, 2,..)` array of integers are pushed to set the **minimum number of subslides**, *without* updating pauses.

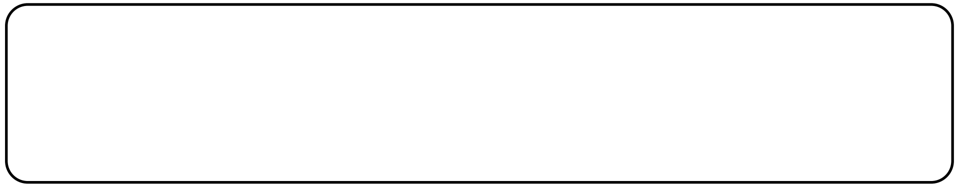
2.12 Rendering Stuffs

Example 2: CeTZ drawings with `#uncover` and `#only`

```
#import "@preview/cetz:0.4.2": canvas, draw
#render(s => ({
  import animation: *
  canvas({
    import draw: *
    let (uncover, pause) = settings(hider: draw.hide.with(bounds: true))
    pause(s, circle((0, 0)))
    s.push(auto)
    uncover(s, 3,
      rect((-1, -1), (1, 1)))
    s.push((3,))
    only(s, 4, circle((1, 1)))
    s.push(4)
  })
}, s))
```

2.12 Rendering Stuffs

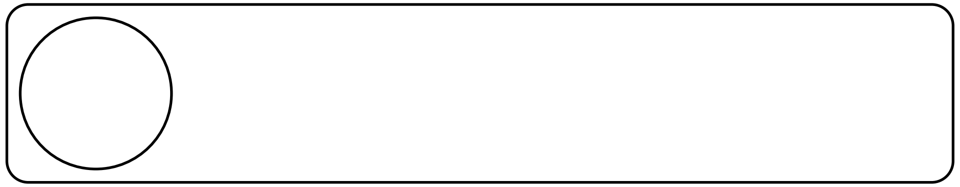
Output: on subslide 1



Notice that the circle produced by `only()` does not preserve space, as it uses `it => none` as hider.

2.12 Rendering Stuffs

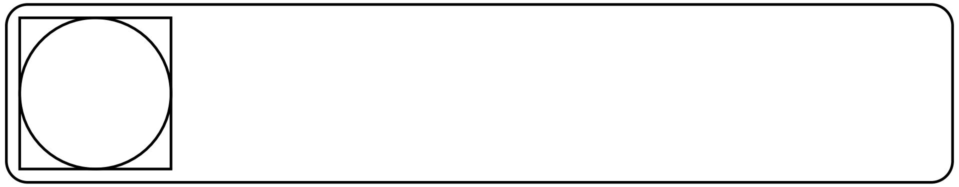
Output: on subslide 2



Notice that the circle produced by `only()` does not preserve space, as it uses `it => none` as hider.

2.12 Rendering Stuffs

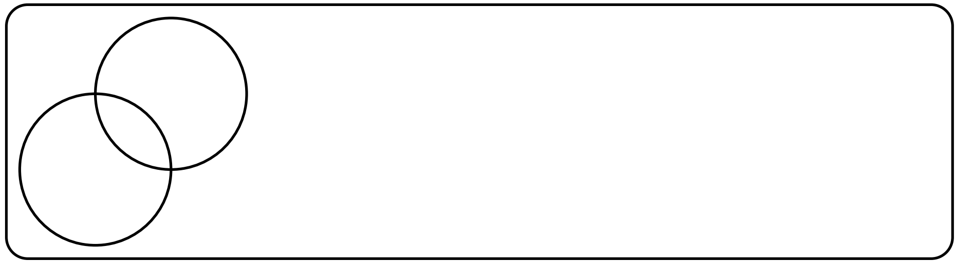
Output: on subslide 3



Notice that the circle produced by `only()` does not preserve space, as it uses `it => none` as hider.

2.12 Rendering Stuffs

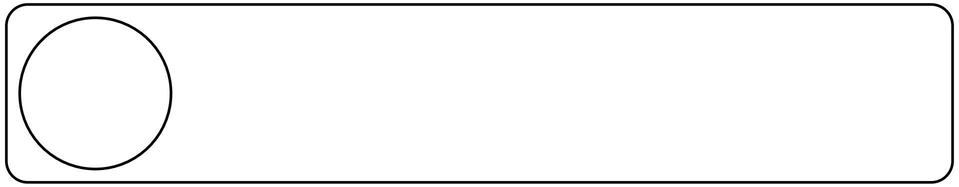
Output: on subslide 4



Notice that the circle produced by `only()` does not preserve space, as it uses `it => none` as hider.

2.12 Rendering Stuffs

Output: on subslide 5

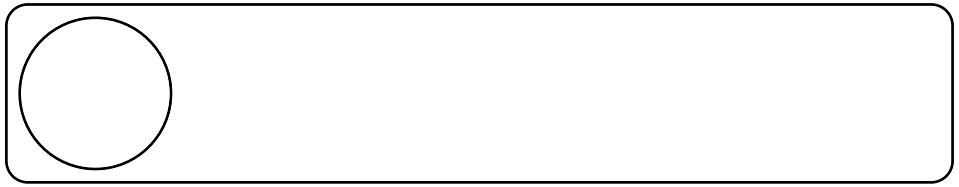


Notice that the circle produced by `only()` does not preserve space, as it uses `it => none` as hider.

The updates: The first `#auto` *increments* the pauses, the second `#(3,)` set the *minimum subslides* to at least 3 and `#4` set the number of *pauses* to 4.

2.12 Rendering Stuffs

Output: on subslide 6



Notice that the circle produced by `only()` does not preserve space, as it uses `it => none` as hider.

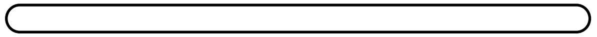
The updates: The first `#auto` *increments* the pauses, the second `#(3,)` set the *minimum subslides* to at least 3 and `#4` set the number of *pauses* to 4.

All you need to do is to update the `#s` for each animation.
For total number of subslides needed, Presentate will do the job *automatically*.

2.12 Rendering Stuffs

Example 3: Fletcher in math mode diagram, with `it => none` as hider.

Output: on subslide 1

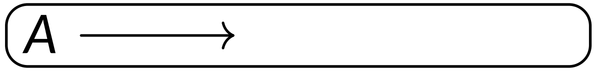


```
#render(s => ({
  import animation: *
  diagram($
      pause(#s, A edge(->)) #s.push(auto)
        & pause(#s, B edge(->)) #s.push(auto)
          pause(#s, edge(->, "d") & C) \
        & pause(#s, D)
    $,)
}, s,))
```

2.12 Rendering Stuffs

Example 3: Fletcher in math mode diagram, with `it => none` as hider.

Output: on subslide 2



```
#render(s => ({
  import animation: *
  diagram($
      pause(#s, A edge(->)) #s.push(auto)
        & pause(#s, B edge(->)) #s.push(auto)
          pause(#s, edge(->, "d") & C) \
        & pause(#s, D)
    $,)
}, s,))
```

2.12 Rendering Stuffs

Example 3: Fletcher in math mode diagram, with `it => none` as hider.

Output: on subslide 3

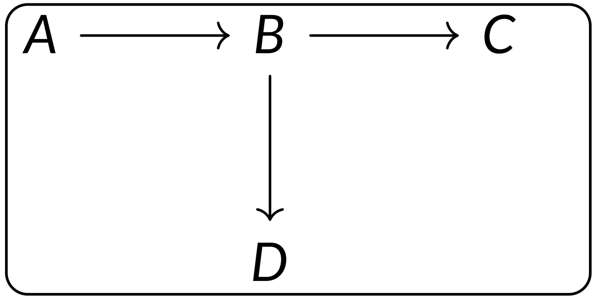
$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & {} \end{tikzcd}$$

```
#render(s => ({
 import animation: *
 diagram($
   pause(#s, A edge(->)) #s.push(auto)
    & pause(#s, B edge(->)) #s.push(auto)
     pause(#s, edge(->, "d") & C) \
    & pause(#s, D)
  $,)
}, s,))
```

2.12 Rendering Stuffs

Example 3: Fletcher in math mode diagram, with `it => none` as hider.

Output: on subslide 4



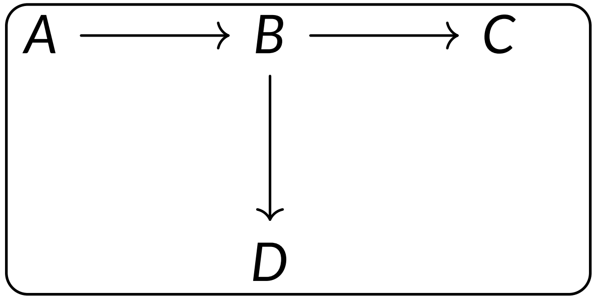
$$\begin{tikzcd}A \arrow[r] & B \arrow[r] \arrow[d] & C \\ & D & \end{tikzcd}$$

```
#render(s => ({
  import animation: *
  diagram($
      pause(#s, A edge(->)) #s.push(auto)
        & pause(#s, B edge(->)) #s.push(auto)
          pause(#s, edge(->, "d") & C) \
        & pause(#s, D)
    $,)
}, s,))
```

2.12 Rendering Stuffs

Example 3: Fletcher in math mode diagram, with `it => none` as hider.

Output: on subslide 5


$$\begin{tikzcd} A \arrow[r] & B \arrow[r] \arrow[d] & C \\ & D & \end{tikzcd}$$

Although not perfect, it is doable.

```
#render(s => ({
  import animation: *
  diagram($
      pause(#s, A edge(->)) #s.push(auto)
        & pause(#s, B edge(->)) #s.push(auto)
          pause(#s, edge(->, "d") & C) \
        & pause(#s, D)
    $,)
}, s,))
```

2.13 Animate the inanimate

Last examples show us how to hack for drawing stuff that has its own `#hider`, either provided by the package or we created it.

[7] https://typst.app/universe/package/alchemist/

2.13 Animate the inanimate

Last examples show us how to hack for drawing stuff that has its own `#hider`, either provided by the package or we created it.

However, I admitted that using `#pause(s, ..)` a lot is tedious, do we have a better way?

[7] https://typst.app/universe/package/alchemist/

2.13 Animate the inanimate

Last examples show us how to hack for drawing stuff that has its own `#hider`, either provided by the package or we created it.

However, I admitted that using `#pause(s, ..)` a lot is tedious, do we have a better way? How about making the input *reactive* to the states?

[7] https://typst.app/universe/package/alchemist/

2.13 Animate the inanimate

Last examples show us how to hack for drawing stuff that has its own `#hider`, either provided by the package or we created it.

However, I admitted that using `#pause(s, ..)` a lot is tedious, do we have a better way? How about making the input *reactive* to the states?

Introducing `#animation.animate` function, together with a package for drawing molecular strcuture: Alchemist[7].

[7] https://typst.app/universe/package/alchemist/

2.13 Animate the inanimate

Alchemist does not provide any hider functions to hide the structure. However, we came up with an idea: setting the hidden bond’s stroke to `#0pt` should effectively hide the bonds, right?

2.13 Animate the inanimate

Alchemist does not provide any hider functions to hide the structure. However, we came up with an idea: setting the hidden bond's stroke to `#0pt` should effectively hide the bonds, right?

So we use the hider as a `#modifier` the function's argument.

```
#import "@preview/alchemist:0.1.8": *
#let (single,) = animation.animate(
  single, modifier: (func, ..args) => func(stroke: 0pt, ..args)
)
```

Note! The animated functions require `#s` as the first positional argument.

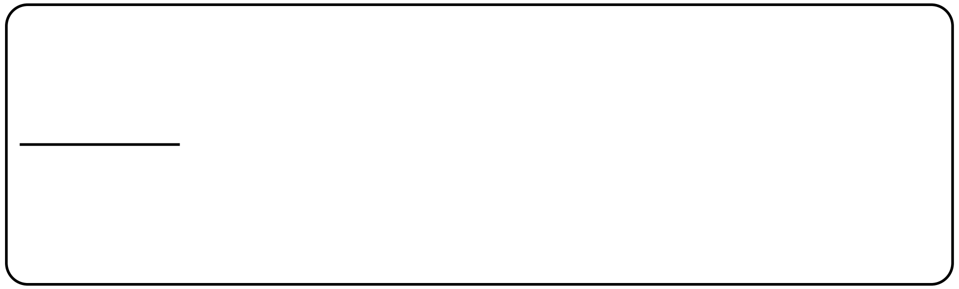
2.13 Animate the inanimate

Output: on subslide 1

```
#render(s => ({
  skeletize({
    single(s) // Note the `s`!
    branch({
      s.push(auto)
      single(s, angle: -1)
    })
    s.push(auto)
    single(s, angle: 1)
  })
}, s))
```

2.13 Animate the inanimate

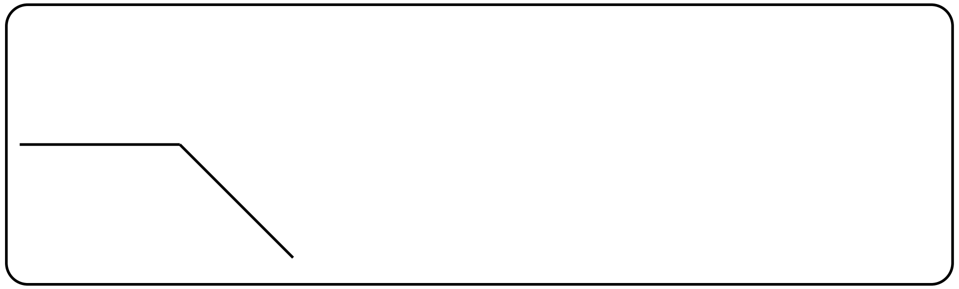
Output: on subslide 2



```
#render(s => ({
  skeletize({
    single(s) // Note the `s`!
    branch({
      s.push(auto)
      single(s, angle: -1)
    })
    s.push(auto)
    single(s, angle: 1)
  })
}, s))
```

2.13 Animate the inanimate

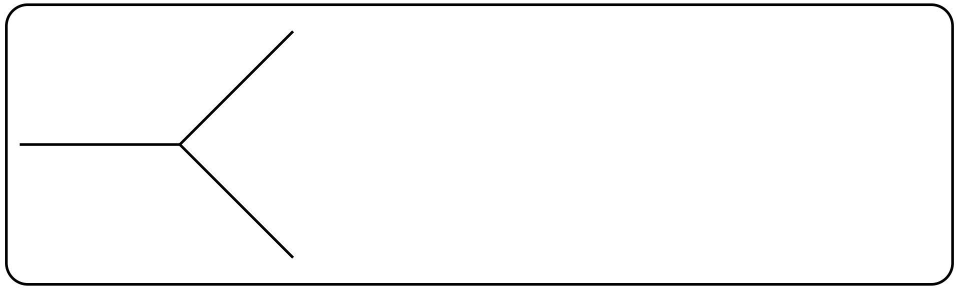
Output: on subslide 3



```
#render(s => ({
  skeletize({
    single(s) // Note the `s`!
    branch({
      s.push(auto)
      single(s, angle: -1)
    })
    s.push(auto)
    single(s, angle: 1)
  })
}, s))
```

2.13 Animate the inanimate

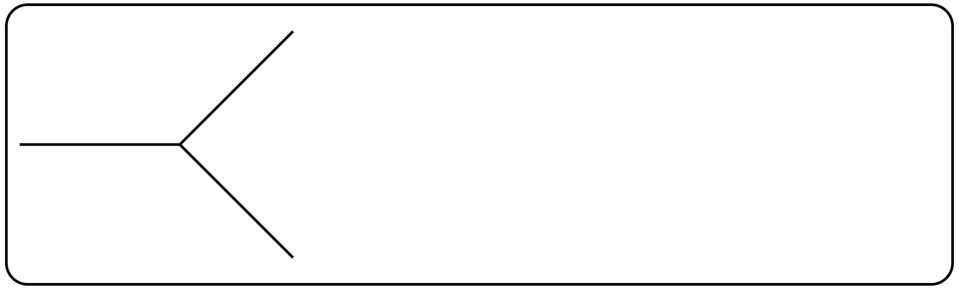
Output: on subslide 4



```
#render(s => ({
  skeletize({
    single(s) // Note the `s`!
    branch({
      s.push(auto)
      single(s, angle: -1)
    })
    s.push(auto)
    single(s, angle: 1)
  })
}, s))
```

2.13 Animate the inanimate

Output: on subslide 5

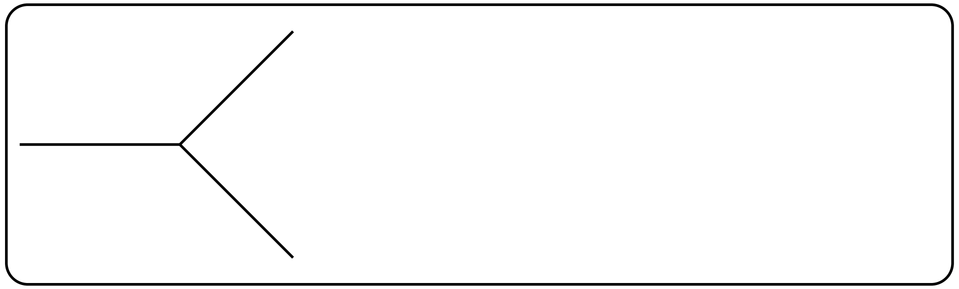


Now the molecule is drawn!

```
#render(s => ({
  skeletize({
    single(s) // Note the `s`!
    branch({
      s.push(auto)
      single(s, angle: -1)
    })
    s.push(auto)
    single(s, angle: 1)
  })
}, s))
```

2.13 Animate the inanimate

Output: on subslide 6



Now the molecule is drawn!
The `#animate` is like *modifier* to make the function *aware* to the `#s` updates.

```
#render(s => ({
  skeletize({
    single(s) // Note the `s`!
    branch({
      s.push(auto)
      single(s, angle: -1)
    })
    s.push(auto)
    single(s, angle: 1)
  })
}, s))
```

2.14 Modes and Utility

Presentate provides three modes for different purposes:

2.14 Modes and Utility

Presentate provides three modes for different purposes:

- **Normal** for animated slides. [Default]

2.14 Modes and Utility

Presentate provides three modes for different purposes:

- **Normal** for animated slides. [Default]
- **Handout** for disabling all animations.

2.14 Modes and Utility

Presentate provides three modes for different purposes:

- **Normal** for animated slides. [Default]
- **Handout** for disabling all animations.
- **Drafted** for showing the subslide number.

2.14 Modes and Utility

Presentate provides three modes for different purposes:

- **Normal** for animated slides. [Default]
- **Handout** for disabling all animations.
- **Drafted** for showing the subslide number.

Normal mode is to do nothing, for the last two options, you can set them via

```
#set-options(handout: true, drafted: true)
```

2.15 Themes

The slide you are viewing is the *simple* theme of Presentate. You can use it by typing the following lines:

```
#import themes.simple: *
#show: template.with(
  author: [Pacaunt], // change to yours!
  title: [Welcome To Presentate!],
  subtitle: [Slides Tools.],
)
```

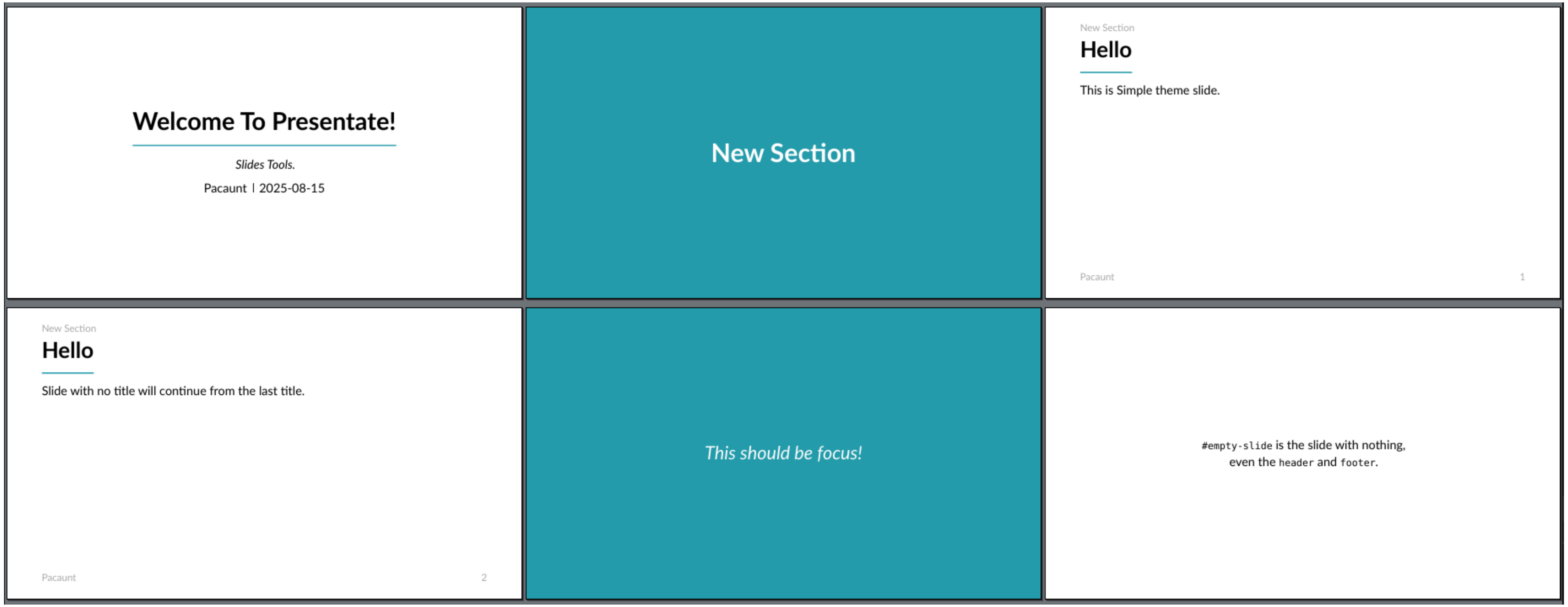
2.15 Themes

The theme provides the following slides:

- `#slide(title, body)` which if no title, it will repeat the last topic.
- `#empty-slide(body)` which is empty and has no margin, header, and footer.
- `#focus-slide(body)` which is colored, vibrant slide for getting attention.

The preview is on the next slide:

2.15 Themes



Welcome To Presentate!

Slides Tools.

Pacaunt | 2025-08-15

New Section

New Section

Hello

This is Simple theme slide.

Pacaunt 1

New Section

Hello

Slide with no title will continue from the last title.

Pacaunt 2

This should be focus!

`#empty-slide` is the slide with nothing, even the `header` and `footer`.

2.15 Themes

```
= New Section
#slide[Hello][
  This is Simple theme slide.
]

#slide[
  Slide with no title will continue from the last title.
]

#focus-slide[
  This should be focus!
]

#empty-slide[
  #set align(center + horizon)
  `#empty-slide` is the slide with nothing, \
  even the `header` and `footer`.
]
```

2.15 Themes

Another theme is *default* theme. It is very minimal, as it sets the paper and text font and sizes, provided with new section slides.

2.15 Themes

Another theme is *default* theme. It is very minimal, as it sets the paper and text font and sizes, provided with new section slides.

You can import it with

```
#import themes.default: *
#show: template.with(
  aspect-ratio: "16-9"
)
```

2.15 Themes

Another theme is *default* theme. It is very minimal, as it sets the paper and text font and sizes, provided with new section slides.

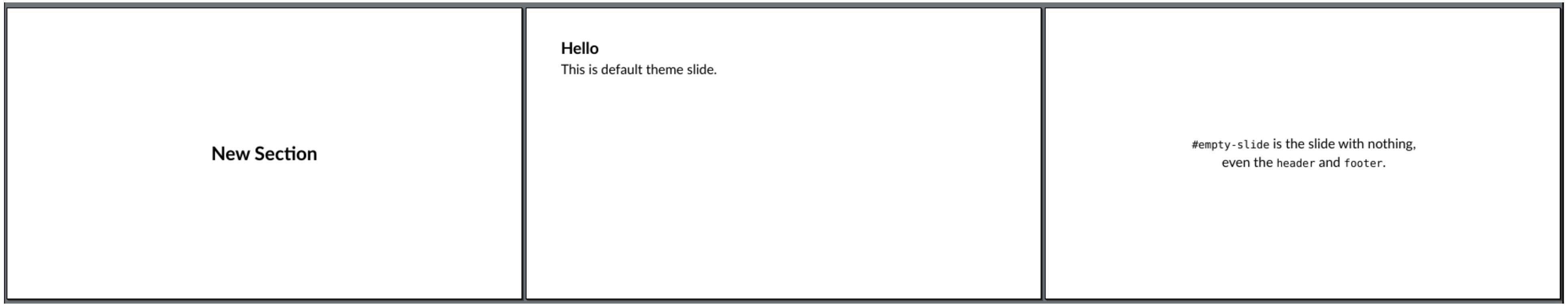
You can import it with

```
#import themes.default: *
#show: template.with(
  aspect-ratio: "16-9"
)
```

and then you will have `#slide(body)`, which are normal slide function, and `#empty-slide(body)` for a slide with no header, footer, and margins.

2.15 Themes

Some example of the Default theme.



New Section

Hello

This is default theme slide.

`#empty-slide` is the slide with nothing,
even the `header` and `footer`.

2.15 Themes

```
= New Section

#slide[
  == Hello
  This is default theme slide.
]

```

```
#empty-slide[
  #set align(center + horizon)
  `#empty-slide` is the slide with nothing, \
  even the `header` and `footer`.
]
```

2.16 Structured Themes

Integration with navigator[8] package provides a structured theme that have an animation of the outlines. Big thanks to David Hajage for providing all of the structured themes and a complete theme guide of this package.

You can visit the examples of the structured themes here:

- minimal
- progressive-outline
- sidebar
- split
- miniframes

[8]https://typst.app/universe/package/navigator

For more information, you can contact us at
Presentate's github
(https://github.com/pacaunt/typst-presentate/)

Enjoy making presentation!