Scrutinize

v0.2.0 March 16, 2024

https://github.com/SillyFreak/typst-scrutinize

Clemens Koza

Abstract

Scrutinize is a library for building exams, tests, etc. with Typst. It provides utilities for common task types and supports creating grading keys and sample solutions.

Contents

I Introduction

Scrutinize has three general areas of focus:

- It helps with grading information: record the points that can be reached for each task and make them available for creating grading keys.
- It provides a selection of task authoring tools, such as multiple choice or true/false questions.
- It supports the creation of sample solutions by allowing to switch between the normal and "pre-filled" exam.

Right now, providing a styled template is not part of this package's scope.

II Tasks and task metadata

Let's start with a really basic example that doesn't really show any of the benefits of this library yet:

```typ
#import "@preview/scrutinize:0.2.0": grading, task, solution, task-kinds
// you usually want to alias this, as you'll need it often
#import task: t

= Task
#t(points: 2)
#lorem(20)
```

Task

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

After importing the library's modules and aliasing an important function, we simply get the same output as if we didn't do anything. The one peculiar thing here is `t(points: 2)`: this adds some metadata to the task. Any metadata can be specified, but `points` is special insofar as it is used by the `grading` module by default.

A lot of scrutinize's features revolve around using that metadata, and we'll soon see how. A task's metadata is a dictionary with the following fields:

- `data`: he explicitly given metadata of the task, such as `(points: 2)`.
- `heading`: the heading that identifies the task, such as `= Task`.
- `subtasks`: an array of nested tasks, identified by nested headings. When getting task metadata, you can limit the depth; this is only present as long as the depth is not exceeded.

Let's now look at how to retrieve metadata. Let's say we want to show the points in each task's header:

```typ
#show heading: it => {
  // here, we need to access the current task's metadata
  block[#it.body #h(1fr) / #task.current().data.points P.]
}
```

Task **/ 2 P.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Here we're using the `task.current()` function to access the metadata of the current task. This function requires context to know where in the document it is called, which a show rule already provides. The function documentation contains more details on how task metadata can be retrieved.

II.a Subtasks

Often, exams have not just multiple tasks, but those tasks are made up of several subtasks. Scrutinize supports this, and reuses Typst's heading hierarchy for subtask hierarchy.

Let's say some task's points come from its subtasks points. This could be achieved like this:

```typ
#show heading.where(level: 1): it => {
  let t = task.current(level: 1, depth: 2)
  block[#it.body #h(1fr) / #grading.total-points(t.subtasks) P.]
}
#show heading.where(level: 2): it => {
  let t = task.current(level: 2)
  block[#it.body #h(1fr) / #t.data.points P.]
}

= Task
#lorem(20)

== Subtask A
#t(points: 2)
#lorem(20)

== Subtask B
#t(points: 1)
#lorem(20)
```

Task **/ 3 P.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Subtask A **/ 2 P.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Subtask B **/ 1 P.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

In this example `task.current()` is used in conjunction with `grading.total-points()`, which recursively adds all points of a list of tasks and its subtasks. More about this function will be said in the next section, and of course in the function's reference.

III Grading

The next puzzle piece is grading. There are many different possibilities to grade an exam; Scrutinize tries not to be tied to specific grading strategies, but it does assume that each task gets assigned points and that the grade results from looking at some kinds of sums of these points. If your test does not fit that schema, you can simply use less of the related features.

The first step in creating a typical grading scheme is determining how many points can be achieved in total, using `grading.total-points()`. We also need to use `task.all()` to get access to the task metadata distributed throughout the document:

```typ
#context [
  #let ts = task.all()
  #let total = grading.total-points(ts)
  #let hard = grading.total-points(ts, filter: t => t.data.points >= 5)
  Total points: #total \ Points from hard tasks: #hard
]

= Hard Task
#t(points: 6)
#lorem(20)

= Task
#t(points: 2)
#lorem(20)
```

Total points: 8
Points from hard tasks: 6

Hard Task / 6

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Task / 2

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Once we have the total points of the exam figured out, we need to define the grading key. Let's say the grades are in a three-grade system of "bad", "okay", and "good". We could define these grades like this:

```typ
#context [
  #let total = grading.total-points(task.all())
  #grading.grades(
    [bad],
    total * 2/4, [okay],
    total * 3/4, [good],
  )
]
```

```
(
  (body: [bad], lower-limit: none, upper-limit: 4.0),
  (body: [okay], lower-limit: 4.0, upper-limit: 6.0),
  (body: [good], lower-limit: 6.0, upper-limit: none),
)
```

Hard Task / 6

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Task / 2

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Obviously we would not want to render this representation as-is, but `grading.grades()` gives us a convenient way to have all the necessary information, without assuming things like inclusive or exclusive point ranges. The `test.typ` example in the gallery has a more complete demonstration of a grading key.

One thing to note is that `grading.grades()` does not process the limits of the grade ranges. If you prefer to ignore total points and instead show percentages, or want to use both, that is also possible:

```typ
#grading.grades(
  [bad],
  (points: total * 2/4, percent: 50%), [okay],
  (points: total * 3/4, percent: 75%), [good],
)
```

```
(
  (
    body: [bad],
    lower-limit: none,
    upper-limit: (points: 4.0, percent: 50%),
  ),
  (
    body: [okay],
    lower-limit: (points: 4.0, percent: 50%),
    upper-limit: (points: 6.0, percent: 75%),
  ),
  (
    body: [good],
    lower-limit: (points: 6.0, percent: 75%),
    upper-limit: none,
  ),
)
```

IV Task templates and sample solutions

With the test structure out of the way, the next step is to actually define tasks. There are endless ways of posing tasks, but some recurring formats come up regularly.

> *Note:* customizing the styles is currently very limited/not possible. I would be interested in changing this, so if you have ideas on how to achieve this, contact me and/or open a pull request. Until then, feel free to "customize using copy/paste".

Tasks have a desired response, and producing sample solutions can be made very convenient if they are stored with the task right away. To facilitate this, this package provides

- `solution._state`: this boolean state controls whether solutions are currently shown in the document. Some methods have convenience functions on the module level to make accessing them easier: `solution.get()`, `solution.update()`.
- `solution.with()`: this function sets the solution state temporarily, before switching back to the original state. The `small-example.typ` example in the gallery uses this to show a solved example task at the beginning of the document.

Additionally, the solution state can be set using the Typst CLI using `--input solution=true` (or `false`, which is already the default), or by regular state updates. Within context expressions, a question can use `#solution.get()` to find out whether solutions are shown. This is also used by Scrutinize's task templates.

Let's look at a free text question as a simple example:

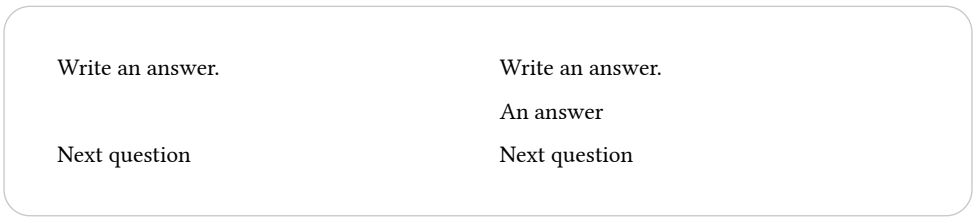
IV.a Free form questions

In free form questions, the student simply has some free space in which to put their answer:

```typ
#import "@preview/scrutinize:0.2.0": grading, task, solution, task-kinds
#import task-kinds: free-form

// toggle the following comment or pass `--input solution=true`
// to produce a sample solution
// #solution.update(true)

Write an answer.
#free-form.plain[An answer]
Next question
```



Left is the unanswered version, right the answered one. Note that the answer occupies the same space regardless of whether it is displayed or not, and that the height can also be overridden - see `free-form.plain()`. The content of the answer is of course not limited to text.

IV.b single and multiple choice questions

These taks types allow making a mark next to one or multiple choices. See `choice.single()` and `choice.multiple()` for details.

```typ
#import "@preview/scrutinize:0.2.0": grading, task, solution, task-kinds
#import task-kinds: choice

Which of these is the fourth answer?
#choice.single(
  range(1, 6).map(i => [Answer #i]),
  // 0-based indexing
  3,
)

Which of these answers are even?
#choice.multiple(
  range(1, 6).map(i => ([Answer #i], calc.even(i))),
)
```

Which of these is the fourth answer?

Answer 1	☐
Answer 2	☐
Answer 3	☐
Answer 4	☐
Answer 5	☐

Which of these answers are even?

Answer 1	☐
Answer 2	☐
Answer 3	☐
Answer 4	☐
Answer 5	☐

Which of these is the fourth answer?

Answer 1	☐
Answer 2	☐
Answer 3	☐
Answer 4	☒
Answer 5	☐

Which of these answers are even?

Answer 1	☐
Answer 2	☒
Answer 3	☐
Answer 4	☒
Answer 5	☐

V Module reference

V.a `scrutinize.task`

scope(body: `content`) -> `content`

Encloses the provided body with labels that can be used to limit what tasks are considered during lookup. This is helpful if there are independent exams/sections in the same document, or for this documentation (separate examples).

The lookup functions in this package properly handle nested scopes.

Example:

```
#task.scope[                                    typ
  Number of tasks: #context task.all().len()
  #task.scope[
    = A (counts) #t()
  ]
  = B (counts too) #t()
]
= C (doesn't) #t()
```

Number of tasks: 2

A (counts)

B (counts too)

C (doesn't)

Parameters:

`body` (`content`) – the scope's body

t(..args: `arguments`) -> `content`

Sets the task data for the most recent heading. This must be called only once per heading; subsequent calls won't do anything. The task metadata can later be accessed using the other functions in this module.

Parameters:

`..args` (`arguments`) – only named parameters: values to be added to the task's metadata

current(
  level: `int`,
  depth: `int`,
  from: `location` `auto` `none`,
  to: `location` `auto` `none`,
  loc: `location` `auto`
) -> `dictionary`

Looks up the most recently defined heading of a certain level and retrieves the task data, in a dictionary of the following form:

- `heading` is the heading that starts the task; through this location information can also be retrieved.
- `data` contains any data passed to `t()`.
- `subtasks` recursively contains more such dictionaries.

`subtasks` only exists down to the specified `depth`: if the depth is 1, no subtasks are retrieved, `auto` means an unlimited depth.

Only tasks within the specified range are considered. For example, if this is called within a `scope()` but the enclosing level 1 heading is *outside* the scope, this function will not find anything and fail. To still succeed, you need to either increase the range via `from` and `to`, or provide a location via `loc` that is outside the problematic scope.

This function is contextual and must appear within a `context` expression.

Examples:

```typ
= Example task
#t(points: 2)
#context [
  #let name = task.current().heading.body
  #let points = task.current().data.points
  "#name" is worth #points points,
  or up to #(points + 1) points for great answers!
]
```

Example task

"Example task" is worth 2 points, or up to 3 points for great answers!

```typ
= Example task
#context [
  #let subtasks = task.current(depth: 2).subtasks
  #let total = subtasks.map(t => t.data.points).sum()
  This task gives #total points total.
]
== Subtask 1 #t(points: 2)
== Subtask 2 #t(points: 2)
```

Example task

This task gives 4 points total.

Subtask 1

Subtask 2

Parameters:

`level` (`int` = `1`) – the level of heading to look up

`depth` (`int` = `1`) – the depth to which to also fetch subtasks

`from` (`location` or `auto` or `none` = `auto`) – the location at which to start looking; `auto` means start of the enclosing `scope()`, `none` means start of the document

`to` (`location` or `auto` or `none` = `auto`) – the location at which to stop looking; `auto` means end of the enclosing `scope()`, `none` means end of the document

`loc` (`location` or `auto` = `auto`) – the location at which to look; `auto` means `here()`

```
all(
    level: int,
    depth: int,
    from: location auto none,
    to: location auto none,
    loc: location auto
  ) -> array
```

Locates all tasks in the document (or scope, or region of the document), which can then be used to create grading keys etc. The return value is an array with elements as described in current().

Headings of the specified `level` are treated as top-level tasks; any higher level headings, even if they have an associated t(), are ignored.

This function is contextual and must appear within a `context` expression.

Examples:

```typ
#context [
  #let tasks = task.all(level: 2)
  Number of tasks: #tasks.len() \
  Total points: #tasks.map(t => t.data.points).sum()
]
== A #t(points: 2)
== B #t(points: 3)
=== B1 #t(points: 1)
```

Number of tasks: 2
Total points: 5

A

B

B1

```typ
#context [
  #let tasks = task.all(level: 3)
  Number of tasks: #tasks.len() \
  Total points: #tasks.map(t => t.data.points).sum()
]
== Part 1
=== A #t(points: 2)
=== B #t(points: 3)
== Part 2
=== C #t(points: 1)
```

Number of tasks: 3
Total points: 6

Part 1

A

B

Part 2

C

Parameters:

`level` (`int` = 1) – the level of heading to look up

`depth` (`int` = 1) – the depth to which to also fetch subtasks

`from` (`location` or `auto` or `none` = `auto`) – the location at which to start looking; `auto` means start of the enclosing scope(), `none` means start of the document

`to` (`location` or `auto` or `none` = `auto`) – the location at which to stop looking; `auto` means end of the enclosing scope(), `none` means end of the document

`loc` (`location` or `auto` = `auto`) – the location at which to look; `auto` means `here()`

V.b `scrutinize.grading`

- total-points()

- grades()

total-points(tasks: `array`, filter: `function`, field: `string`) -> `integer` `float`

Takes an array of task metadata and returns the sum of their points, recursivly including subtasks. Tasks without points count as zero points.

Parameters:

`tasks` (`array`) – an array of task metadata dictionaries

`filter` (`function` = `none`) – an optional filter function for determining which tasks to sum up. Subtasks of tasks that didn't match are ignored.

`field` (`string` = `"points"`) – the field in the metadata over which to calculate the sum

grades(..args: `any`) -> `array`

A utility function for generating grades with upper and lower point limits. The parameters must alternate between grade names and threshold scores, with grades in ascending order. These will be combined into dictionaries for each grade with keys `body`, `lower-limit`, and `upper-limit`. The first (lowest) grade will have a `lower-limit` of `none`; the last (highest) grade will have an `upper-limit` of `none`.

Example:

```typ
#let total = 8
#let (bad, okay, good) = grading.grades(
  [bad], total * 2/4, [okay], total * 3/4, [good]
)
You will need #okay.lower-limit points to pass,
everything below is a _#(bad.body)_ grade.
```

You will need 4 points to pass, everything below is a *bad* grade.

Parameters:

`..args` (`any`) – only positional: any number of grade names interspersed with scores

V.c `scrutinize.solution`

- get()
- update()
- with()

Variables:

- _state

get() -> `boolean`

A direct wrapper around `state.get()` for the solution state `_state`.

update(value: `boolean`) -> `content`

A direct wrapper around `state.update()` for the solution state `_state`.

Parameters:

`value` (`boolean`) – the new solution state

with(value: `boolean`, body: `content`) -> `content`

Sets whether solutions are shown for a particular part of the document.

Example:

```typ
Before: #context solution.get() \
#solution.with(true)[
  Inside: #context solution.get() \
]
After: #context solution.get()
```

Before: false
Inside: true
After: false

Parameters:

`value` (`boolean`) – the solution state to apply for the body

`body` (`content`) – the content to show

_state `state`

The boolean state storing whether solutions should currently be shown in the document. This can be set using the Typst CLI using `--input solution=true` (or `false`, which is already the default) and accessed by the regular methods of the `state` type or by the convenience functions of this module: `get()`, `update()`. Additionally, `with()` can be used to change the solution state temporarily.

V.d `scrutinize.task-kinds.choice`

- checkbox()
- multiple()
- single()

checkbox(correct: `boolean`) -> `content`

A checkbox which can be ticked by the student. If the checkbox is a correct answer and the document is in solution mode, it will be ticked.

Example:

```typ
#import task-kinds.choice: checkbox
Correct: #checkbox(true) -- Incorrect: #checkbox(false)
```

Correct: □ – Incorrect: □ Correct: ☒ – Incorrect: □

Parameters:

`correct` (`boolean`) – whether the checkbox is of a correct answer

multiple(options: `array`) -> `content`

A table with multiple options that can each be true or false. Each option is a tuple consisting of content and a boolean for whether the option is correct or not.

Example:

```typ
#import task-kinds: choice
#choice.multiple(
  range(1, 6).map(i => ([Answer #i], calc.even(i))),
)
```

Answer 1	□
Answer 2	□
Answer 3	□
Answer 4	□
Answer 5	□

Answer 1	□
Answer 2	☒
Answer 3	□
Answer 4	☒
Answer 5	□

Parameters:

`options` (`array`) – an array of (option, correct) pairs

single(options: `array`, answer: `integer`) -> `content`

A table with multiple options of which one can be true or false. Each option is a content, and a second parameter specifies which option is correct.

Example:

```typ
#import task-kinds: choice
#choice.single(
  range(1, 6).map(i => [Answer #i]),
  // 0-based indexing
  3,
)
```

Answer 1	☐
Answer 2	☐
Answer 3	☐
Answer 4	☐
Answer 5	☐

Answer 1	☐
Answer 2	☐
Answer 3	☐
Answer 4	☒
Answer 5	☐

Parameters:

`options` (`array`) – an array of contents

`answer` (`integer`) – the index of the correct answer, zero-based

V.e `scrutinize.task-kinds.free-form`

- plain()
- lines()
- grid()

plain(answer: `content`, height: `auto` `relative`, stroke) -> `content`

An answer to a free form question. If the document is not in solution mode, the answer is hidden but the height of the element is preserved.

Example:

```
#import task-kinds: free-form
Write an answer.
#free-form.plain(stroke: 0.5pt, pad(x: 0.5em, y: 1em)[
  an answer
])
Next question
```

typ

Write an answer.

Next question

Write an answer.

an answer

Next question

Parameters:

`answer` (`content`) – the answer to (maybe) display

`height` (`auto` or `relative` = `auto`) – the height of the region where an answer can be written

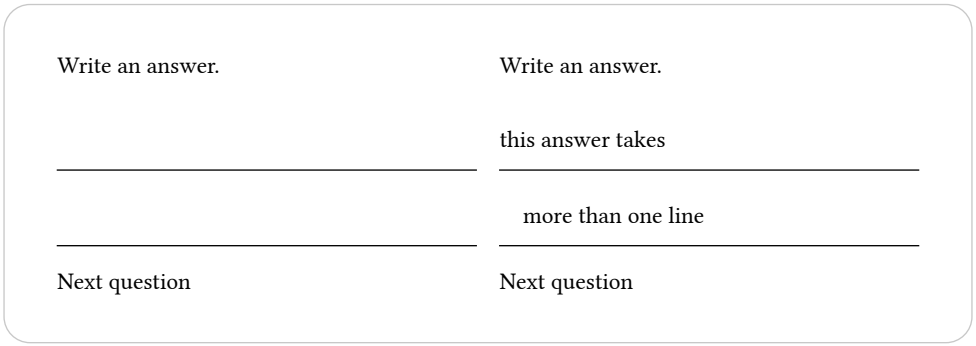
lines(answer: `content`, count: `auto` `int`, line-height: `relative`) -> `content`

An answer to a free form question. If the document is not in solution mode, the answer is hidden but the height of the element is preserved.

This answer type is meant for text questions and shows a lines to write on. By default, the lines are spaced to match a regular paragraph (assuming the text styles are not changed within the answer) and the number of lines depends on what is needed for the sample solution. If a `line-height` is explicitly set, the `par.leading` is adjusted to make the sample solution fit the lines. Paragraph spacing is currently not adjusted, so for the answer to look nice, it should only be a single paragraph.

Example:

```typ
#import task-kinds: free-form
Write an answer.
#free-form.lines(line-height: 1cm)[this answer takes \ more than one line]
Next question
```



Parameters:

`answer` (`content`) – the answer to (maybe) display

`count` (`auto` or `int` = `auto`) – the number of lines to show; defaults to however many are needed for the answer

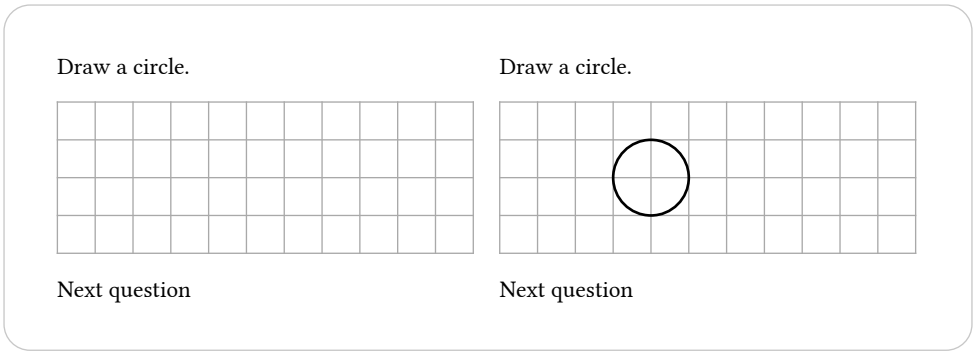
`line-height` (`relative` = `auto`) – the line height; defaults to what printed lines naturally take

grid(answer: `content` , height: `auto` `relative` , size: `relative` `dictionary`) -> `content`

An answer to a free form question. If the document is not in solution mode, the answer is hidden but the height of the element is preserved.

This answer type is meant for math and graphing tasks and shows a grid of lines. By default, the grid has a 5mm raster, and occupies enough vertical space to contain the answer. The grid fits the available width; use padding or similar to make it more narrow.

```typ
#import task-kinds: free-form
Draw a circle.
#free-form.grid(height: 20mm, {
  place(dx: 15mm, dy: 5mm, circle(radius: 5mm))
})
Next question
```



Parameters:

`answer` (`content`) – the answer to (maybe) display

`height` (`auto` or `relative` = `auto`) – the height of the grid region

`size` (`relative` or `dictionary` = `5mm`) – grid size, or a dictionary containing `width` and `height`