Callisto Reference Manual

Version 0.3.0

Contents

1. Introduction

The main functionality of Callisto is exposed through reading-rendering and export-execution functions. These functions accept keyword arguments called *settings*. All functions accepts the same settings and can be preconfigured together using a single `config` call. For example, the following configures the `render` and `errors` functions to read from the file `notebook.ipynb`:

```
#import "@preview/callisto:0.3.0"

#let (render, errors) = callisto.config(nb: path("notebook.ipynb"))

// Check that there are no errors
#if errors().len() > 0 { panic("Notebook has errors") }

// Render the whole notebook
#render()
```

These functions also accept a *cell specification* as positional argument, such as a cell index, or a cell label or tag or an array of such values. Examples:

```
// Render the first two cells
#render((0, 1))
// Get the errors of the cell cell with label or tag "plot"
#errors("plot")
```

See Cell specification for all the ways that cells can be specified.

Some notebooks contain Markdown that includes images from external files. To correctly render such Markdown, a *path handler* must be defined to give Callisto access to the files in the project directory:

```
#let (render, errors) = callisto.config(
  nb: path("notebook.ipynb"),
  handlers: (path: (x, ..args) => path(x)),
)
```

1.1. Settings Overview

Making full sense of all the settings requires some familiarity with the reading-rendering and export-execution functions, so these functions are documented first, and the settings are presented in detail later in Section 5. Here is however a brief overview of the available settings:

`nb`: The notebook to read from. Default: `none`.

`cell-header-pattern`: The pattern of header lines in code cells. Default: `"#| %key: %value"`.

`keep-cell-header`: Whether header lines in code cell source should be kept/rendered. Default: `false`.

`count`: Whether to count cells by index or by execution count. Default: `"index"`.

`name-path`: Where to look for cell names/labels in cell metadata. Default: `auto` which resolves to `("metadata.callisto.header.label", "id", "metadata.tags")`.

`cell-type`: The type(s) of cells to process. Default: `"all"`.

`lang`: The language of code cells. Default: `auto` to use notebook metadata.

`raw-lang`: The language of raw cells. Default: `none`.

`item`: The output item to keep. Default: `"unique"`, which raises an error if several are found.

`output-type`: The types of output to keep. Default: `"all"`.

`format`: The output format(s) to use, in order of preference. Default: `auto` for the default list.

`ignore-wrong-format`: Whether to ignore outputs without suitable format, rather than raise an error. Default: `false`.

`stream`: The text stream(s) to include in cell outputs. Default: `"all"`.

`result`: Whether each output should be returned as a simple value or a dict with metadata. Default: `"value"`.

`handlers`: Handlers to override the processing of certain types of values. Default: `(:)`.

`new-handlers`: User-defined handlers. Default: `(:)`.

`input`: Whether to render the input of code cells. Default: `true`.

`output`: Whether to render the output of code cells. Default: `true`.

`cmarker`: Additional configuration for the `cmarker` package, such as `h1-level`. Default: `(:)`.

`gather-latex-defs`: Whether LaTeX command definitions should be gathered from the whole notebook and made available to every formula. Default: `true`.

`console-text`: How to process text outputs that might contain ANSI escape sequences. Default: `auto` to process values that contain such sequences.

`apply-theme`: Whether to apply theme handlers even outside of rendering. Default: `false`.

`theme`: Theme for rendering the notebook cells. Default: `"notebook"`.

`export-name`: Identifier to use for this notebook export. Default: `"notebook"`.

`export-label`: Label for the metadata holding the exported notebook. Default: `auto` to derive the label from `export-name`.

`cell-header`: Keys and values to add to the header of exported cells. Default: `none`.

`kernel`: Name of Jupyter kernel to use in exported notebook. Default: `none`.

`transform`: Transformation function to apply to every output item. Default: `none`.

`placeholder`: Configuration for the placeholder to use when an output item or cell is missing. Default: `auto` to enable default placeholders when reading from an exported notebook.

2. Reading and Rendering

There are many functions for extracting items from a notebook and for rendering notebook content as Typst content. However they are all variants of four main functions: `render`, `outputs`, `sources` and `cells`.

2.1. Main Functions

`render`

Used to render notebook cells as Typst content. A `render` call can be used to render anything from a single cell to a whole notebook.

Markdown cells are converted to Typst content: Markdown headings to Typst headings, LaTeX equations to Typst equations, etc. By default, both the source and outputs of code cells are rendered (in a style that depends on the selected theme), and raw cells are rendered as simple raw blocks.

```
render(cell-spec, ..args) → content
```

The positional argument is a cell specification. For the other arguments, see Configuration.

Examples:

```
// Render whole notebook
#render()
// Render first 10 cells
#render(range(10))
// Render last cell
#render(-1)
```

`outputs`

Returns the outputs of the selected cells, as an array of output items: strings, images, etc. It operates only on code cells; other cell types are ignored.

```
outputs(cell-spec, ..args) → array
```

The positional argument is a cell specification. For the other arguments, see Configuration.

Example:

```
// Get all outputs that are errors or error streams
#outputs(output-type: ("error", "stream"), stream: "stderr")
```

`sources`

Returns the source of cells as an array of `raw` elements. The value of the `lang` field on the `raw` element depends on the cell type:

- For Markdown: `"markdown"`.
- For code cells: the notebook language (configured `lang` value, or language name from notebook metadata if `lang` is not configured).
- For raw cells: the configured `raw-lang`, or `none` if not configured.

```
sources(cell-spec, ..args) → array
```

The positional argument is a cell specification. For the other arguments, see Configuration.

Example:

```
// Get the source of all Markdown cells
#sources(cell-type: "markdown")
```

`cells`

Returns the cells themselves, as Typst dictionaries holding the JSON data found in the notebook file. This is a low-level function to be used for further processing.

```
cells(cell-spec, ..args) → array
```

The positional argument is a cell specification. For the other arguments, see Configuration.

Example:

```
// Get all Markdown cells
#cells(cell-type: "markdown")
```

2.2. Variants

The functions in the previous section have many variants that work similarly, but for only a single cell or a single output, or for a specific output type.

`Cell`, `In`, `Out`

The `render` function has three variants that work on a single cell, raising an error if zero or more than one are found:

`Cell`: Renders one cell. This is like `render` except that an error is raised if several matching cells are found, and an error is raised (or a `placeholder` is used if enabled) when no match is found.

`In`: Renders the source of a code cell. Equivalent to a `Cell` call with `input: true` and `output: false`.

`Out`: Renders the outputs of a code cell. Equivalent to a `Cell` call with `input: false` and `output: true`.

Examples:

```
// Render cell with "plot" label/tag, checking there is only one
#Cell("plot")
// Render only its output
#Out("plot")
```

`output`, `source`, `cell`

The `outputs`, `sources` and `cells` functions have "singular" variants that return a single value (raising an error if zero or more than one are found):

`output`: Returns one output. An error is raised if several outputs are found, unless the `item` setting is used to pick one. An error is also raised if no output is found, unless placeholders are enabled.

`source`: Returns a single cell's source. An error is raised if zero or several matching cells are found.

`cell`: Returns a single cell. An error is raised if zero or several matching cells are found.

Examples:

```
// Unique output of first cell
#output(0)
// Last output of first cell
#output(0, item: -1)
// Unique cell with label or tag "plot", returned as dict
#cell("plot")
// Get all outputs of that cell, raising an error if 0 or >1 cells are found
#outputs(cell("plot"))
```

`displays`, `results`, `streams`, `full-streams`, `errors`

The `outputs` function has further variants to target a specific type of output:

`displays`: Returns the "display" outputs of the selected cells (a single cell can produce several display outputs on top of its "execution result"). Equivalent to `outputs` with `output-type: "display"`.

`results`: Returns the execution results of the selected cells (generally the value produced by the last line in the cell code). Equivalent to `outputs` with `output-type: "result"`.

`streams`: Returns the text streams produced by the cells. Equivalent to `outputs` with `output-type: "stream"`.

`full-streams`: Works like `streams` but merging all stream items produced by the same cell into a single value.

`errors`: Returns the errors produced by the cells. Equivalent to `outputs` with `output-type: "error"`.

Examples:

```
// Get the results of the first 10 cells
#results(range(10))
// Get all errors found in the notebook
#errors()
```

`display`, `result`, `stream`, `full-stream`, `error`

Type-specific output functions also come in singular versions:

`display`: Returns a single display output.

`result`: Returns a single result output.

`stream`: Returns a single stream output.

`full-stream`: Returns the merged stream items for a single cell.

`error`: Returns a single error output.

These functions behave like `output` when one or several matches are found.

Examples:

```
// Get the result of the first cell
#result(0)
// Get full stream of the single cell with label/tag "plot" that has streams
// items in its outputs.
#full-stream("plot")
```

3. Export and Execution

Callisto can be used to export raw elements (e.g. code blocks) from the Typst document into a Jupyter notebook file. This notebook can be executed outside of Typst, for example with `jupyter-nbconvert`. This notebook can also be used as the input file for Callisto, to automatically include execution results in the Typst document.

Export is done by compiling with `typst eval --input callisto-export=true` instead of `typst compile`. During export, most functions are disabled: `outputs`, `sources` and `cells` return empty

arrays, `render`, `source` and `cell` return `none`, and `Cell`, `In`, `Out` and `output` and its variants return a `placeholder`.

When rendering the execution result, the notebook works like a portable cache holding the result of every code block. You can share the Typst document together with the notebook and your collaborators will be able to edit the Typst code and recompile the document (though this will include outdated results if the executed blocks are changed). The notebook can also be uploaded to the Web App to edit the document online.

See Section 3.3.2 for the steps required to export and execute a notebook.

3.1. Cell Functions

The following functions are used to export individual cells. While the reading-rendering functions accept all kinds of cell specifications, the functions shown here accept only a raw element. This element is used as source for the exported cell. In the case of `execute` and `evaluate`, this element is also used together with the cell header and cell position to find the executed cell when reading from the notebook.

`export`

Wraps the given raw element in a metadata element labeled for export. The return value should be inserted in the document so that `stage-notebook` and `make-notebook` can find it.

```
export(raw-spec, ..args) → content
```

The positional argument is a raw element. For the other arguments, see Configuration.

The `export` function exports a cell without rendering it. This can be used to add silent "setup code" in the notebook. Example:

````
// Export a cell that configures the Python pandas library
export(`pd.set_option('display.max_columns', None)`)
````

Cell header key-value pairs can be specified either as header lines or with the `cell-header` setting so the following are equivalent:

````
#export(```
#| label: calc
2+2
```)

#export(
  cell-header: (label: "calc"),
  ```
  2+2
  ```
)
````

`execute`

Exports the given raw element and renders it from the notebook file. It is essentially equivalent to calling `export` + `Cell`.
````

```
execute(raw-spec, ..args) → content
```

The positional argument is a raw element. For the other arguments, see Configuration.

Examples:

```
// Export the code "2+2" as a cell and render it
#execute(`2+2`)

// Find every raw block with `py-x` lang, export them and render them
#show raw.where(block: true, lang: "py-x"): execute

```py-x
2 + 2
```
```

`evaluate`

Exports the given raw element and returns the single output of the corresponding cell in the notebook file. It is essentially equivalent to calling `export` + `output`.

```
evaluate(raw-spec, ..args) → content
```

The positional argument is a raw element. For the other arguments, see Configuration.

Example:

```
The sum of 3 and 4 is #evaluate(`3+4`).
```

The return value of `evaluate` is not the cell output itself, but content made of the export metadata and the cell output. To work with the output value, the easiest is generally to use the `transform` setting: The transform function gets the actual output value (generally as string) and can transform it before it is joined with the export metadata. For example in the following we get the result of `3+4` (as string), convert it to integer and use that in a for loop to produce seven squares:

```
#evaluate(
  `3+4`,
  transform: x => for i in range(int(x)) { sym.square },
)
```

Note for advanced usage: Usually, `evaluate` returns simple content (metadata + cell output) that can be introspected and manipulated, as long as care is taken to ensure that the export metadata is included in the document when "compiling" for export (during a `typst eval` call from CLI). However when several exported cells have the exact same source and same header, Callisto will need a query with context to disambiguate the cells, and in that case the return value will be opaque. To get non-opaque return values in such cases, each cell can be given a unique header. Example:

```
// Make sure the return values can be introspected/manipulated
#evaluate(`3+4`)
#evaluate(`3+4`, cell-header: (dedup: "2"))
#evaluate(`3+4`, cell-header: (dedup: "3"))
```

Thankfully this is generally not needed: instead of manipulating a return value from outside the `evaluate` call, one can usually use `transform` to manipulate it during the call, before it becomes opaque.

3.2. Notebook Functions

To export a notebook, individual cells must be exported using the functions in the previous section. These cells can be gathered into a whole notebook using the following functions.

`stage-notebook`

Make a notebook out of all the raw elements exported with the specified `export-name` and return the result as a metadata element labelled with `export-label`. The metadata value is a dictionary that can be converted to JSON to obtain a valid Jupyter notebook. This JSON can be read from the command-line using `typst eval` for storing as an `.ipynb` file.

```
stage-notebook(..args) → content
```

For the arguments, see Configuration.

If the `kernel` setting is not specified, the value recorded in the raw elements at the time of the `export`/`execute`/`evaluate` calls is used.

Example:

````
#let (execute, stage-notebook) = callisto.config(
  nb: path("export.ipynb"),
  kernel: "python3",
)

// Insert the notebook metadata in the document so that typst eval can find it
#stage-notebook()

// Export and render all Python code blocks
#show raw.where(lang: "py-x"): execute

```py-x
2+2
```
````

See Section 3.3.2 for a full example including the terminal commands used to export and execute a notebook.

`make-notebook`

Looks for the exported raw elements with matching `export-name` and returns a notebook dictionary. This function must be called with `context` (unlike `stage-notebook`).

```
make-notebook(..args) → dictionary
```

For the arguments, see Configuration.

If the `kernel` setting is not specified, the value recorded in the raw elements at the time of the `export`/`execute`/`evaluate` calls is used.

This function is used by `stage-notebook` to prepare the notebook dictionary, but it can be useful for other purposes as shown in the examples below.

3.2.1. Example: Notebook as PDF attachment

Here Callisto is not used to render notebooks but to add a notebook as attachment to the compiled PDF. The notebook contains a cell for every Python code block found in the document.

```
#import "@preview/callisto:0.3.0"

#context pdf.attach(
  "notebook.ipynb",
  bytes(json.encode(callisto.make-notebook(kernel: "python3"))),
  mime-type: "application/x-ipynb+json",
  relationship: "supplement",
  description: "Notebook of all code blocks in the document",
)

#show raw.where(block: true, lang: "py"): it => callisto.export(it) + it

```py
2+2
```
```

3.2.2. Example: Check for outdated export

Here a notebook is prepared for export, but the result is used only to compare the cell sources with those found in the notebook version that is actually on disk: if they differ, it means that the exported file is outdated.

```
#context {
  // Get cell sources from external notebook (on disk)
  let external = sources()
  // Get cell sources from raw elements marked for export in this document
  let internal = sources(nb: make-notebook())
  // Check for any difference
  if external.len() != internal.len() or array.zip(external, internal).any(
    ((a, b)) => a.text != b.text
  ) {
    // Cell sources differ: the file on disk is outdated
    rect(width: 100%, inset: 1em, stroke: red)[Exported notebook outdated!]
  }
}
```

3.3. Putting It All Together

This section presents the typical workflow for executing code blocks of a Typst document through a Jupyter kernel and having the result integrated in the compiled document.

How code blocks are selected for export, and read back from the exported notebook to show the result, is largely up to the user. Some examples:

- Wrap raw blocks manually in `export` calls, then call functions such as `Cell` and `output` explicitly to render the cells:

```
#export(```
#| label: plot
plot(x, y)
```)

#Cell("plot")
```

- Do the same thing with an explicit `execute` or `evaluate` call:

```
#execute(```
plot(x, y)
```)
```

- Use a show rule to apply `execute` or `evaluate` automatically:

```
// Select blocks by lang
#show raw.where(lang: "py-x"): execute

```py-x
plot(x, y)
```

// Select blocks by label
#show <exec>: execute

```py
plot(x, y)
```<exec>
```

3.3.1. Issues with Show Rules

The "show rule" method described above allows for the nicest syntax but comes with a significant downside: the `execute` function runs in the context of the raw element that is being replaced by the show rule, while the "raw style" is active. This means the execution result is subject to the default rules on `raw` as well as any `show raw: set ...` rule added by the user.

Problems appear also when the `execute` function produces a raw element: the raw style is applied again to the new element, so a font size in "em" units ends up scaling the text twice (see this issue).

These issues might get resolved if/when Typst adds support for replacing show rules. In the meantime, it is best to "revoke" the raw style directly in the show rule. By default, raw elements have styling equivalent to `set text(0.8em)` so we write:

```
#show raw.where(lang: "py-x"): it => {
  set text(1em/0.8) // Revoke raw style
  execute(it)
}
```

The raw style also applies a monospace font which is often not a problem. When it is undesired, we can write something like `set text(1em/0.8, font: "Libertinus Serif")`.

Finally, when selecting blocks with `#show raw.where(lang: ..): execute`, it is generally a bad idea to use the standard language name (e.g. "python") for the raw blocks: the show rule can inadvertently

select blocks that should not be executed, in particular blocks that are produced by the `execute` call itself when it renders the code input, which can lead to infinite recursion. Prefer a non-standard `lang` value, for example `"py-x"` instead of `"py"`.

3.3.2. Complete Workflow with the Command Line

Here is a complete workflow using show rules to select code blocks for execution:

1. In the document:

   - Use `config` to configure the functions you want to use. Set the notebook path and Jupyter kernel with the `nb` and `kernel` settings. You can choose the notebook path but it must correspond to the file created during export. This is also the notebook that Callisto will read from to show the code blocks and results.
   - Choose a `lang` tag for code blocks that should be executed and add a show rule to execute them.

   Example:

   ```
   #import "@preview/callisto:0.3.0"

   #let (execute, stage-notebook) = callisto.config(
     nb: path("export.ipynb"),
     kernel: "python3",
   )

   // Execute all code blocks that have "py-x" lang
   #show raw.where(lang: "py-x"): it => {
     set text(1em/0.8) // Revoke raw style
     execute(it)
   }

   // Make notebook from exported (executed) code blocks
   #stage-notebook()

   Some computation with Python:
   ```py-x
   2 + 3
   ```

   Another computation:
   ```py-x
   2 + 4
   ```
   ```

   Note: until you do the first export, Typst will complain that it doesn't find the notebook file. To avoid this error, comment-out the `nb: ...` line in the `config` call until you are ready to export the notebook.

2. Export and execute the notebook:

   ```
   typst eval --input callisto-export=true --in document.typ \
       'query(<notebook>).first().value' > export.ipynb
   jupyter-nbconvert --to notebook --execute --inplace export.ipynb
   ```

This will create (or overwrite) the file `export.ipynb`. Make sure you use the same filename as was specified in the `nb: ...` line of the `config` call.

Note the flag `--input callisto-export=true` used during export: This tells Callisto to disable most functions so they don't produce an error while the notebook is unavailable for reading.

That's it: the next time you compile `document.typ`, Callisto will read the execution results from `export.ipynb` and include them in the compiled document.

In this example, we just exported some code blocks using `execute`. Here is a more comprehensive example that also uses `export` and `evaluate`:

```
#import "@preview/callisto:0.3.0"

#let (execute, export, evaluate, stage-notebook) = callisto.config(
  nb: path("export.ipynb"),
  kernel: "python3",
)

// Execute all code blocks that have "py-x" lang
#show raw.where(lang: "py-x"): it => {
  set text(1em/0.8) // Revoke raw style
  execute(it)
}

// Make notebook from exported (executed) code blocks
#stage-notebook()

// Export some code as cell but without rendering the cell
#export(`a = 2`)

Some computation with Python:
```py-x
a + 3
```

The same, getting just the result: #evaluate(`a + 3`).
```

Here we included some "setup code" using `export`, to create a notebook cell that will be executed along with the other cells but not rendered in the document.

3.4. Automatic Export/Execution with a Makefile or justfile

The calls to `typst eval` and `jupyter-nbconvert` can be automated using a simple Makefile:

```
EVAL := typst eval --input callisto-export=true --in document.typ

default: export execute

export:
  $(EVAL) "query(<notebook>).first().value" > export.ipynb

execute:
  jupyter-nbconvert --to notebook --execute --inplace export.ipynb

watch:
  watchexec -w . -f '**/*.typ' make export execute
```

```
.PHONY: default export execute watch
```

Running the `make` command will then export and execute the notebook. This also defines a `watch` target: assuming you have `watchexec` installed, you can:

1. run `make watch` in a terminal to monitor the current directory for changes to the `.typ` files: whenever one of these files changes it will run `make export execute` for us,
2. run `typst watch document.typ` in another terminal (or use the preview feature in your editor) to see the execution results automatically included in the Typst output.

And here is an equivalent `justfile` to use with `just` instead of `make`:

```
default: export execute

EVAL := "typst eval --input callisto-export=true --in document.typ"

export:
  {{EVAL}} 'query(<notebook>).first().value' > export.ipynb

execute:
  jupyter-nbconvert --to notebook --execute --inplace export.ipynb

watch:
  watchexec -w . -f '**/*.typ' just export execute
```

3.5. Several Notebooks from a Single Typst Document

Several notebooks can be used independently of each other in the same Typst document. This can be used to execute code using different kernels, or to have each chapter run code blocks in its own execution context.

The `nb` and `export-name` settings must have distinct values for each export. Here is a complete example:

```
#import "@preview/callisto:0.3.0"

#let (execute: execute-python, stage-notebook: stage-python) = callisto.config(
  nb: path("export-python.ipynb"),
  kernel: "python3",
  export-name: "python",
)

#let (execute: execute-julia, stage-notebook: stage-julia) = callisto.config(
  nb: path("export-julia.ipynb"),
  kernel: "julia-1.11",
  export-name: "julia",
)

#stage-python()
#stage-julia()

#show raw.where(lang: "py-x"):    it => { set text(1em/0.8); execute-python(it) }
#show raw.where(lang: "julia-x"): it => { set text(1em/0.8); execute-julia(it) }

A computation in Python:

```py-x
2 + 2
```

And one in Julia:

```julia-x
2 + 3
```
```

The following justfile can be used to automate the export and execution of both notebooks:

```
default: export execute

EVAL := "typst eval --input callisto-export=true --in document.typ"
EXEC := "jupyter-nbconvert --to notebook --execute --inplace"

export:
  {{EVAL}} 'query(<python>).first().value' > export-python.ipynb
  {{EVAL}} 'query(<julia>).first().value'  > export-julia.ipynb

execute:
  {{EXEC}} export-python.ipynb
  {{EXEC}} export-julia.ipynb

watch:
  watchexec -w . -f '**/*.typ' just export execute
```

3.5.1. Exporting Multiple Notebooks Together

By default, `stage-notebook` derives the label for the exported notebook from the `export-name` value, so each notebook can be extracted by its own `typst eval` call as shown above. For large documents it can be interesting to get all the notebooks with a single `tyst eval` command for performance reasons.

Something like `query(selector.or(<python>,<julia>))` could be used in `typst eval` to retrieve several notebooks. Maybe a better way is to export all notebooks under the same label using the `export-label` setting:

```
#stage-notebook(export-name: "python", export-label: <notebook>)
#stage-notebook(export-name: "julia",  export-label: <notebook>)
```

Using `query(<notebook>)` in `typst eval` will then return a JSON array of notebooks. The export name of each notebook is available in the field `metadata.callisto.export-name` of each array value. A Bash command could be used to write each array value to its own notebook file:

```
typst eval --input callisto-export=true --in document.typ \
  'query(<notebook>).map(x => x.value)' |
  jq -c '.[]' | while read -r nb; do
    filename=$(jq -r '.metadata.callisto."export-name"' <<<"$nb")
    echo "$nb" > "export-${filename}.ipynb"
  done
```

Even better: we can use the `callisto.export-names` function to automatically find and stage all the notebooks under the same label. Here is a complete example:

```
#import "@preview/callisto:0.3.0"

#let (execute: execute-a,) = callisto.config(
  nb: path("export-python-a.ipynb"),
  kernel: "python3",
  export-name: "python-a",
)
#let (execute: execute-b,) = callisto.config(
  nb: path("export-python-b.ipynb"),
  kernel: "python3",
  export-name: "python-b",
)

// Stage all exported notebooks under <notebook> label
#context for name in callisto.export-names() {
  callisto.stage-notebook(export-name: name, export-label: <notebook>)
}

#execute-a(`a = 1`)
#execute-b(`a = 10`)
#execute-a(`a += 3; print(a)`)
#execute-b(`print(a)`)
```

And here is a justfile to automatically retrieve all the notebooks and write them to separate files, and to execute them:

```
default: export execute

EVAL := "typst eval --input callisto-export=true --in document.typ"

SPLIT := '''
import json
for nb in json.load(open(0, encoding="utf-8")):
    name = nb["metadata"]["callisto"]["export-name"]
    with open(f"export-{name}.ipynb", "w", encoding="utf-8") as f:
        json.dump(nb, f)
'''

EXEC := '''
import subprocess, glob
args = ["jupyter-nbconvert", "--to", "notebook", "--execute", "--inplace"]
for file in glob.glob("export-*.ipynb"):
    subprocess.run(args + [file], check=True)
'''

export:
    @echo "Exporting notebooks..."
    @{{EVAL}} 'query(<notebook>).map(x => x.value)' | python3 -c '{{SPLIT}}'

execute:
    @echo "Executing notebooks..."
    @python3 -c '{{EXEC}}'

watch:
    watchexec -w . -f '**/*.typ' just export execute
```

Notes:
- This should work on Linux, macOS and Windows as long as Python is installed.
- It assumes that notebook paths are of the form `export-xxx.ipynb` where `xxx` is the `export-name`.

4. Cell Specification

The Callisto functions that operate on cells accept a positional argument to select the cells that should be processed. This argument is called the *cell specification*.

The functions `render`, `outputs`, `sources`, `cells` and all their variants accept all the forms listed below.

The functions `export`, `execute` and `evaluate` only accept the "raw element" form.

One form of specification is notably missing from the list: `none`. Indeed the `none` value is used internally to represent that no specification was provided by the user (meaning that all cells should be selected). To specify "no cell", use the empty array `()`.

4.1. Allowed Values

`int`: By default this refers to the cell index in the notebook, counting from the end if the value is negative. The `count` setting can be used to interpret the value as an execution count (the number shown in square brackets to the left of the cell in the Jupyter notebook interface). Examples:

```
#render(0)  // render first cell
#render(-1) // render last cell
#render(1, count: "execution") // render cell with execution count equal to 1
```

`str`: By default this can be either a cell label, cell ID or cell tag. The cell ID and tags are standard cell attributes in Jupyter notebooks. The cell label is specific to Callisto: it refers to the `label` field in the cell's `metadata.callisto.header` dict. A cell can be labeled by adding a header line at the top of the cell source:

```
#| label: two-and-two
2+2
```

Callisto will automatically convert the header line to a cell label in the metadata. See Section 7 for details.

The `name-path` setting can be used to change where Callisto will look in the cell dict to check for matching cell names.

Examples:

```
// Render cell(s) with label or tag "plot1"
#render("plot1")
// Render cell(s) with `metadata.callisto.header.type` field set to "scatter"
// (for example a cell with `#| type: scatter` in the header).
#render("scatter", name-path: "metadata.callisto.header.type")
```

`content` (a single `raw` element): This finds all the code cells in the notebook that have exactly the same source code and header as the raw element.

The header for the provided raw element is computed by merging the `cell-header` setting with the header rows found in the raw element text. The result is compared to the header built from the source of each code cell in the notebook. Examples:

```
// Render the code cell(s) with label "calc" and single line of code: `2 + 2`
#render(
  ```typ
  #| label: calc
  2 + 2
  ```
)
// Another way to do the same thing:
#render(`2 + 2`, cell-header: (label: "calc"))
```

`label`: This finds cells that were exported from a Typst document using raw blocks with the given Typst label. Example:

```
#show raw.where(lang: "python-x"): export

```python-x
sum(range(5))
```<sum-calc>

Here is how to compute $1 + 2 + 3 + 4 + 5$ in Python:
#render(<sum-calc>)
```

See Section 3 for more information about this functionality.

Note: Typst labels should not be confused with cell labels (which are strings defined in the cell header). They are two independent concepts with different features and limitations. For example cell labels are meant to be unique, while the same Typst label can be used on many cells, e.g. to select many inline raw elements for export. Ideally we would use different names for the two concepts, but we try to maintain some compatibility with Quarto chunk options.

`function`: The function is passed a cell dict and must return `true` for desired cells, `false` otherwise. Example:

```
// Results of cells with execution count larger than 3
#results(c => c.execution_count > 3)
```

`dictionary` (cell): The dictionary must be a cell dict as returned by a `cells` call. Example:

```
// Get the cell with label "nice-plot"
#let c = cell("nice-plot")
// Render it with the "plain" theme
#render(c, theme: "plain")
```

`array`: An array of the above. Cells that match any of the array elements are included in the result. Examples:

```
// Render the first 10 cells
#render(range(10))
// Render first cell, "plot1" cell and all cells that have an error
#render((
    0,
    "plot1",
    c => c.at("outputs", default: ()).any(x => x.output_type == "error"),
))
```

5. Configuration

The functions in the previous sections all accept the same settings which can be applied when the function is called, for example `render(nb: path("notebook.ipynb"))`. However it is usually more convenient to configure the desired functions ahead of time using `config`.

5.1. Preconfiguring Several Functions

`config`

A single `config` call can apply the same settings to several functions:

```
config(..args) → dictionary
```

Arguments can include any of the settings defined in the next section.

The returned dictionary includes preconfigured versions of all the functions from the previous sections, as well as a `template` function as defined by the selected theme (or a do-nothing template if the theme defines no template).

Full example:

```
#import "@preview/callisto:0.3.0"

#let (output, render, template) = callisto.config(
  nb: path("notebook.ipynb"),
  output-type: ("display", "result"),
  item: 0,
  theme: "neat",
)

#show: template

#figure(
  caption: [Output of last cell],
  output(-1),
)

= Complete Notebook
#render()
```

Here `output` and `render` are configured to use `notebook.ipynb` as notebook, keep only display and result outputs (ignoring errors and streams), and in case of multiple outputs to keep only the first one (item 0). Additionally the "neat" theme is selected, and the corresponding template applied to the document.

5.2. Settings

This section lists all the settings that can be used as arguments to `config` or directly when calling `render`, `outputs`, `sources`, `cells`, `export`, `execute`, `evaluate` or any of their variants.

`nb` `path` `bytes` `dictionary` `none`

The notebook to read from (default: `none`). This can be the path to a notebook file, or the content of a notebook either as `bytes` or as a dict as returned by the `json` function, or `none` (for example when exporting without reading back). Examples:

```
// Specify the notebook by path
#let (output, render) = callisto.config(nb: path("notebook.ipynb"))

// Give the notebook content directly, as dictionary
#let (output, render) = callisto.config(nb: json("notebook.ipynb"))
```

When using export and execution, the path form is required so that a `typst eval` can succeed when creating the exported notebook the first time, when the file doesn't exist yet.

`cell-header-pattern` `str` `dictionary` `auto` `none`

The pattern that defines which lines at the start of a code cell constitute a header holding metadata (default: `auto`).

If given as string, the pattern must include the "words" `%key` and `%value`, and any whitespace in the string is considered as representing any amount of whitespace (possibly none).

If `auto`, a default pattern string is used: `"# | %key: %value"`

For more control, a dictionary with fields `regex` and/or `writer` can be specified. The regular expression must define a first capture group for the key and a second one for the value. The `writer` field must be a function that takes key and value strings as positional arguments and returns a header line without trailing newline. If the `regex` field is missing or `none`, cells are treated as having no header. If the `writer` field is missing or none, attempts to generate a cell dict (e.g. for export) will fail.

A value `none` is equivalent to `(regex: none, writer: none)`.

The default pattern matches lines of the form `#| key: value` and `# | key: value` (a space between `#` and `|` is allowed as it might be added by code formatters and expected by linters). This is appropriate for kernels that recognize `#` as starting a line comment. For other kernels the pattern must be set manually. Examples:

```
// Header pattern for languages that start line comments with //
#let (render,) = callisto.config(
   nb: path("notebook.ipynb"),
   cell-header-pattern: "//| %key: %value",
)

// Header pattern with strict format `#| key: value` without whitespace
// between `#` and `|`
#let (render,) = callisto.config(
   nb: path("notebook.ipynb"),
   cell-header-pattern: (
      regex: regex("^#\|\s+(.*?):\s+(.*?)\s*$"),
      writer: (key, value) => "#| " + key + ": " + value,
   ),
)
```

`keep-cell-header` `bool`

When `true`, the cell header is not removed from the cell source. The default is `false`. Example:

```
// Render cells while preserving the cell metadata headers
#render(keep-cell-header: true)
```

`count` `str`

Can be `"index"` or `"execution"`, to select if a cell number refers to its position in the notebook (zero-based) or to its execution count. The default is `"index"`. Example:

```
// Cells with execution count between 5 and 9
#cells(range(5, 10), count: "execution")
```

`name-path` `str` `array` `auto`

If given as string, it defines the "path" in the cell dict where Callisto looks for cell names when cells are specified by string. A string of the form `x.y.z` refers to the field `z` of the field `y` of the field `x` in the cell dict. The cell is selected if the value found in this path is either the cell specification string, or an array containing that string.

The cell dict always includes a `metadata` dict with cell metadata, which itself always includes a `callisto.header` dict with all the key-value pairs defined in the cell header. Example:

```
// Render cells that have `#| type: scatter` in the cell header
#render("scatter", name-path: "metadata.callisto.header.type")
```

An array of strings can be given, in which case each value is considered as a possible path.

The value `auto` (the default) corresponds to the array `("metadata.callisto.header.label", "id", "metadata.tags")`.

`cell-type` `str` `array`

Can be `"markdown"`, `"raw"`, `"code"`, an array of these values, or `"all"`. The default is `"all"`. This setting can be used to restrict the cell selection to specific cell types. This filtering out has no effect on the cell indices, which refer always to the position in the unfiltered notebook. Examples:

```
// Render all Markdown cells
#render(cell-type: "markdown")

// Get the source of all code cells
#sources(cell-type: "code")

// Get non-raw cells among the notebook's first 10
#cells(range(10), cell-type: ("markdown", "code"))

// For comparison: get first 10 of the non-raw cells
#cells(cell-type: ("markdown", "code")).slice(0, count: 10)
```

`lang` `str` `auto` `none`

The language of the notebook's code cells. This is used as language tag for the raw blocks holding the source of code cells. If `auto` (the default), the language is read from the notebook metadata. Example:

```
// Get source of all cells, setting the language to python for code cells
#sources(lang: "python")
```

`raw-lang` `str` `none`

The language of the notebook's raw cells. This is used as language tag for the raw blocks holding the source of raw cells. The default is `none`. Example:

```
// Get source of all cells, setting `lang` to `typ` for raw cells
#sources(raw-lang: "typ")
```

`item` `int` `str`

This controls wich item should be returned by the "singular" output functions: `output`, `display`, `result`, `stream`, `full-stream`, `error`.

The default is `"unique"`. In this case an error is raised when zero or more than one items are found. An integer value can be specified to pick one item in case of multiple matches. A negative value can be used to count from the end.

Examples:

```
// Get the only "result" output, raising an error if there are more than one
#result()
// Get the notebook result, picking the last one if there are more than one
#result(item: -1)
```

`output-type` `str` `array`

Selects which output types should be included in the result. Can be `"display"`, `"result"`, `"stream"`, `"error"`, an array of these values, or `"all"` (the default).

A `result` output is usually the value returned by the last line in a code cell; this output is usually missing if the last line returns nothing. It is possible to have other cell lines generate results (for example using `sys.displayhook` in Python) but that is uncommon and not recommended. In the notebook file, a single result is often stored as multiple values in different formats to let the reader choose a preferred format. Use the `format` setting to choose a particular format.

A `display` output is a display object generated by a code cell, in addition to (or instead of) the cell result. A cell can have many display outputs. Just like `result` outputs, each `display` output can be stored in multiple formats in the notebook.

A `stream` is a piece of text written by the cell either on the standard output or the standard error (the `stream` setting can be used to filter for a particular stream). When a cell produces interleaved messages on both streams, each message is stored in the notebook as a separate stream item so that the order of messages is preserved. Use the `result: "dict"` setting to see the stream name of stream items.

An `error` stores information on an error raised during the execution of a cell. Use the `result: "dict"` setting to get detailed information on the error, including a backtrace.

Example:

```
// Get the results and/or errors of all code cells
#outputs(output-type: ("result", "error"))
```

`format` `str` `array` `auto`

Used to select the format for an output items, as Jupyter notebooks can store the same output in several formats to let the viewer choose a format.

This can be a MIME string such as `"image/png"`, or an array of such strings. The order of the array sets the preference: the first match is used. Every listed format must have a corresponding handler.

The value `auto` (the default) represents the default array `callisto.default-formats`:

```
(
  "text/vnd.typst",
  "image/svg+xml",
  "image/png",
  "image/gif",
  "image/jpeg",
  "text/markdown",
  "text/latex",
  "text/html",
  "text/plain",
  "application/json",
)
```

The value `auto` can also be used as one element of an array of values; in this case the default array will be inserted at that position. Example:

```
// Get PNG version of all display outputs, error if a display has no PNG
#outputs(output-type: "display", format: "image/png")
// Get PNG version where available, use default precedence otherwise
#outputs(output-type: "display", format: ("image/png", auto))
```

Note that the format with highest precedence is `text/vnd.typst`. This is the MIME type for Typst source code. The default handler for this format renders the value as Typst markup using the `eval` function.

For example, the following Python cell produces a `text/vnd.typst` value with `text/plain` fallback:

```
#| label: typst-markup
from IPython.display import display

typst_code = """
= Basel Problem

$ sum_(i=1)^oo = pi^2/6 $
"""

display({
    'text/vnd.typst': typst_code,
    'text/plain': '<Typst Document>'
}, raw=True)
```

Writing `#output("typst-markup")` will include the Typst heading and equation in the document. The text fallback will be used in notebook viewers that don't know how to render Typst source code.

`ignore-wrong-format` `bool`

By default an error is raised if a selected output is not available in one of the desired formats (see `format` setting). Set this to `true` to skip the output silently. Example:

```
// Get PNG version of all display outputs, ignoring displays without PNG
#outputs(format: "image/png", ignore-wrong-format: true)
```

`stream` `str`

For stream outputs, this selects the type of streams that should be returned. Can be `"stdout"`, `"stderr"` or `"all"` (the default). Example:

```
// Get all errors, and all messages written to stderr
#outputs(output-type: ("error", "stream"), stream: "stderr")
```

`result` `str`

How the function should return its result: `"value"` (the default) to return each result as a simple value, or `"dict"` to return it as a dictionary that contains a `"value"` field plus other fields holding metadata.

The additional fields depend on the function called but include at least a `cell` dict holding the cell index, ID, metadata, type and (for code cells) execution count.

Examples:

```
// Get for each code cell a a dict with source and various metadata
#sources(cell-type: "code", result: "dict")

// Get the traceback of the first error
#error(item: 0, result: "dict").traceback
```

`handlers` `dictionary`

A dictionary mapping "MIME types" to handler functions. A "MIME type" can be an actual MIME type like `image/png`, or a pseudo MIME type used for internal processing such as `cell`. A handler is a function called to process a value of a particular type. Each handler should accept a positional argument for the data to process and any keyword argument, and return the processed data. The dict passed to `handlers` is merged with the dict of default handlers, overriding default values with the specified ones. The default is an empty dict: `(:)`.

Example:

```
// Show all text outputs in uppercase
// (writing the field name in quotes since it contains a slash)
#outputs(handlers: ("text/plain": (data, ..args) => upper(data)))
```

Each field in the dictionary can be a function, or `auto` (representing the default handler), or `none` (equivalent to a function that returns `none`), or an array of such values.

In the case of an array, the handler functions are chained by calling the first one, then the second one with the result of the first, etc. Example:

```
// Render each cell in a frame by calling the default handler, then putting
// the result in a rect
#let frame(it, ..args) = rect(it, width: 100%)
#render(handlers: (cell: (auto, frame)))
```

This setting can only be used to redefine handlers for known "MIME types": an error is raised if a dict field doesn't correspond to a known handler, to catch typos. New "MIME types" can be registered with the `new-handlers` setting.

new-handlers `dictionary`

This works like handlers, but allows registering new handlers, for example to support additional output formats. Default: `(:)`.

For example one can register a handler for the `model/obj` MIME type to render cell outputs that hold 3D models in the OBJ format:

```
#import "@preview/callisto:0.3.0"
#import "@preview/maquette:0.1.0": render-obj

#let obj-handler(data, ..args) = render-obj(
  camera: (2, 3, 3),
  stroke: (color: "#000000", width: 1),
  data,
)

#callisto.render(
  nb: path("notebook.ipynb"),
  // Register new handler
  new-handlers: ("model/obj": obj-handler),
  // Add OBJ format as first in the list of desired formats
  format: ("model/obj", auto),
)
```

input `bool` `auto`

Whether `render` should render the input (source code) of code cells.

The default is `auto`. In this case the value is taken from the `echo` field of the cell header, defaulting to `true` if the field is absent.

Example:

```
// Render the first five cells without showing the source of code cells
#render(range(5), input: false)
```

output `bool` `auto`

Whether `render` should render the output of code cells.

The default is `auto`. In this case the value is taken from the `output` field of the cell header, defaulting to `true` if the field is absent.

Example:

```
// Render the first five cells without showing the outputs of code cells
#render(range(5), output: false)
```

cmarker `dictionary`

Additional settings to pass to `cmarker.render` when converting Markdown to Typst. Default: `(:)`. Notable settings:

`h1-level`: The Typst heading level corresponding to top-level headings in the notebook. All other levels are shifted accordingly. The value can be negative, in which case a heading shifted to level 0 is converted to a `title` element. Default: `1`.

`task-list-marker`: A callback for rendering the markers in Markdown task lists. Default: `none`.

See the cmarker documentation for more information on the available settings.

Example for `h1-level`:

```
// Render notebook with top-level heading converted to document title,
// second-level headings converted to level 1, etc.
#render(cmarker: (h1-level: 0))
```

A value larger than 1 can be useful to include a notebook as a chapter/section of a larger document:

```
= Chapter 1

// Render notebook with top-level headings converted to level 2, second-level
// headings converted to level 3, etc.
#render(cmarker: (h1-level: 2))
```

`gather-latex-defs` `bool`

Whether all the LaTeX command definitions (of the form `\newcommand`) in the notebook should be gathered into into a single "preamble" to be used when rendering any part of the notebook.

Jupyter allows defining a LaTeX command in an equation and using it in another equation (contrary to actual LaTeX, where a definition in some equation is local to that equation). This can cause difficulties in Callisto: the default LaTeX renderer (MiTeX) doesn't allow a command local to one equation to be used in another. Also the user might render a single cell that uses a command defined in another. To address these issues, by default Callisto gathers all command definitions in a single preamble string: at render time, such definitions are removed from the equation (to avoid duplicate definitions) and the whole preamble is inserted at the beginning of the equation. This whole processing can be disabled by setting `gather-latex-defs: false`.

```
// Leave LaTeX equations as-is
#render(gather-latex-defs: false)
```

`console-text` `bool` `auto` `str` `dictionary`

How to process text that might be meant for a terminal. This setting affects every value that is processed through the `text-console-block` handler. By default this is all stream, error and `text/plain` outputs.

Text shown in a terminal might require special handling to render correctly, in particular:

- Conversion of ANSI escape sequences to text styles (colors, underline, etc.).
- Adjustments to the paragraph leading and text edges to avoid gaps in the background color between rows of text, and to have box-drawing characters connect properly across rows.

Use `true` or `false` to enable or disable the processing unconditionally.

Use `auto` (the default) to have Callisto look for ANSI escape sequences in the text and enable processing if any are found.

Use `"strip"` to have Callisto simply remove all escape sequences, without applying any styling.

Use a dictionary for more control on the processing:

- The `render` field can hold `true`/`false`/`auto`/`"strip"`. These values have the same meaning as when passed directly to `console-text`.
- Additional fields are passed as arguments to `ansi.render`. Supported fields include `palette` (an array of 16 standard ANSI colors), `fg` and `bg` (foreground and background colors) and `bold-is-bright` (whether bold text in a standard normal color should be rendered in the corresponding bright color).

Examples:

```
// Render cells, removing any escape sequence found in text outputs
#render(console-text: "strip")

// Render cells, treating all text outputs as console text with black background
#render(console-text: (render: true, bg: black))

// Render cells using a gruvbox palette for ANSI text
#let gruvbox = (
  rgb("#282828"), rgb("#cc241d"), rgb("#98971a"), rgb("#d79921"),
  rgb("#458588"), rgb("#b16286"), rgb("#689d6a"), rgb("#a89984"),
  rgb("#928374"), rgb("#fb4934"), rgb("#b8bb26"), rgb("#fabd2f"),
  rgb("#83a598"), rgb("#d3869b"), rgb("#8ec07c"), rgb("#ebdbb2"),
)
#render(console-text: (palette: gruvbox, bg: gruvbox.at(0), fg: gruvbox.at(-1)))
```

See the `ansi` module for more information on ANSI rendering.

`apply-theme` `bool`

Whether the theme handlers should be used also for processing the items returned by `outputs`.

A theme is a set of handlers used in place of the defaults while rendering cells. The `outputs` function (and variants) is meant to extract values for further processing by the user, so in principle the theme is not used by `outputs` but this can be changed by setting `apply-theme: true`.

For example, text outputs can contain ANSI escape codes used in terminals to colorize the text. A string with such codes can look like gibberish. When rendering a cell with ANSI codes in the output, the codes are converted to text styles. We can use `apply-theme` to apply the same conversion to outputs that we extract manually:

```
// Get the outputs of the first cell, transforming them through
// the theme handlers.
#outputs(0, apply-theme: true)
```

`theme` `str` `dictionary`

The theme used for rendering content (default: `"notebook"`). This can be the name of a built-in theme as string, or a theme dictionary (see Section 9 for more information). By default the theme has no effect on the `outputs` function and its variants, but this can be changed using the `apply-theme` setting.

Currently available built-in themes:

- `notebook`: renders cells in a style similar to the Jupyter graphical interface, with code cell execution counts shown in square brackets in the left margin.
- `neat`: a simpler theme, more appropriate for regular documents such such as reports. This theme provides a document template.
- `plain`: a theme that adds no additional styling on the notebook content, except for the rendering of ANSI escape sequences (see the `console-text` setting).

Example:

```
#let (template, render) = callisto.config(
  nb: path("notebook.ipynb"),
  theme: "neat",
)
#show: template
#render()
```

`export-name` `str`

The name used to identify raw elements belonging to a particular notebook export. The default is `"notebook"`. If you want to export several notebooks from the same document, you must give each of them a unique export name.

By default, the export name is also used to label the metadata holding the exported notebook. This can be changed with the `export-label` setting.

The export name is independent from the notebook file: the filename can be unwieldy, or undefined if the notebook is exported but not read back.

Example:

```
#let (execute, stage-notebook) = callisto.config(
  nb: path("notebooks/export-python.ipynb"),
  kernel: "python3",
  export-name: "python",
)

#stage-notebook()
```

The notebook staged in this example can be retrieved from the command line and written to a file using the following command:

```
typst eval --input callisto-export=true --in document.typ \
    'query(<python>).first().value' > notebooks/export-python.ipynb
```

`export-label` `label` `auto`

The label for the metadata returned by `stage-notebook`. The default is `auto`, to derive a label from the `export-name` value. Example:

```
#let (execute, stage-notebook) = callisto.config(
  nb: path("notebooks/export-python.ipynb"),
  kernel: "python3",
  export-name: "python",
  export-label: <notebook>, // instead of the default <python>
)
#stage-notebook()
```

This setting is often used to exported multiple notebooks under the same label, so they can be retrieved from the command line with a single `typst eval`. See Section 3.5.1 for a complete example.

`cell-header` `dictionary` `none`

If a dictionary is given, each key-value pair is added as a header line in the source of exported cells. All field values in the dictionary must be strings.

This setting is especially useful when generating cells programmatically. Example:

```
#let exprs = ("1+1", "2+2")

// Export a cell for each expression
#for (i, expr) in exprs.enumerate() {
  // Give a unique label to each cell
  export(raw(expr), cell-header: (label: "calc" + str(i)))
}

The result of the second computation is #output("calc2").
```

`kernel` `str` `none`

The name of the Jupyter kernel to use when exporting a notebook (default: `none`).

Setting the kernel name enables the export and execution functionality.

To see the list of kernels available in your Jupyter installation, use the command `jupyter kernelspec list` which prints a list like the following:

```
$ jupyter kernelspec list
Available kernels:
  ir            /home/user/.local/share/jupyter/kernels/ir
  julia-1.11    /home/user/.local/share/jupyter/kernels/julia-1.11
  python3       /usr/share/jupyter/kernels/python3
```

We see that on this system there is an R kernel (`"ir"`), a Julia kernel (`"julia-1.11"`) and a Python kernel (`"python3"`).

Note: In a standard Jupyter installation the kernel name is given by the directory holding the kernel description. For example the Python kernel is described in the file `/usr/share/jupyter/kernels/python3/kernel.json` so its name is `python3`.

Example:

```
#let (execute, stage-notebook) = callisto.config(
  nb: path("export.ipynb"),
  kernel: "python3",
)

#stage-notebook()

#show raw.where(lang: "py-x"): execute

```py-x
2+2
```
```

And a minimal example without rendering (the notebook is prepared for export but not read back):

```
#import "@preview/callisto:0.3.0"

#callisto.export(`2+2`)
#callisto.export(`2+3`)
#callisto.stage-notebook(kernel: "python3")
```

In both cases, the exported notebook can be obtained from the command-line using the command

```
typst eval --input callisto-export=true --in document.typ \
    'query(<notebook>).first().value' > export.ipynb
```

`transform` `function` `none`

If defined, this function is called on every output item. The function must accept the item value as positional parameter and return the transformed value.

Example:

```
// Put a rect around every output
#output("plot1", transform: rect)
```

This setting is useful for manipulating cell outputs in documents that use export and execution: During export (with `typst eval`) functions such as `output` and `evaluate` return a `placeholder` instead of the real output value. Furthermore, `evaluate` doesn't return the bare value (or placeholder) but a content value that includes metadata. Working with such values that have different types during `typst compile` vs `typst eval` can be cumbersome. The `transform` function is applied directly on the real cell outputs, sparing us this complexity.

For example, we can transform a NumPy vector into a Typst math `vec`, to typeset as part of a formula. Here is a complete example:

````
#import "@preview/callisto:0.3.0"

#let (output, execute, stage-notebook) = callisto.config(
  nb: path("export.ipynb"),
  kernel: "python3",
)
#show raw.where(lang: "py-x"): execute

#stage-notebook()

// Transform "array([1., 2., 3.])" into math.vec("1", "2", "3")
#let py-vec = output.with(
  transform: s => math.vec(..s.slice(7, -2)
                           .split(", ")
                           .map(x => str(float(x)))),
)

We solve the linear system with NumPy:

```py-x
#| label: linear-system
import numpy as np
A = np.array([[1, 1, 1], [0, 1, 1], [0, 0, 1]])
b = np.array([6, 5, 3])
np.linalg.solve(A, b)
```

The solution is:
$ arrow(x) = #py-vec("linear-system") $
````

`placeholder` `auto` `bool` `function` `any`

The value to use in place of a missing value.

Placeholders are used by functions that extract a single output and functions that render a single cell. For example `output` and `result` should always return a single output. If none is found, we know that one is missing, so a placeholder is used if this feature is enabled. On the other hand, `outputs` and `results` can very well return an empty array if the target cells have no such outputs; this doesn't mean that a value is missing, so no placeholder is used. Similarly, `Cell` uses placeholders since it is supposed to find exactly one cell, while `render` doesn't since any number of matches is valid.

Placeholders are particularly useful when using export and execution: it is annoying to get an error in the editor every time a code block is added/edited and the corresponding execution result not yet available. With placeholders, Callisto can render the source of the code block as "work in progress" instead of raising an error.

During an export run (`typst eval` with `--input callisto-export=true`), placeholders are always enabled when `kernel` is set, to prevent functions such as `Cell` and `output` from raising an error.

Here is how the possible values for this setting are interpreted when a value is missing:

`true`: A placeholder is obtained by calling the corresponding handler:

- `placeholder-output` for `output` (and its variants) and `evaluate`,
- `placeholder-Cell` for `Cell` and `execute`,
- `placeholder-In` for `In`,

- `placeholder-Out` for `Out`.

`false`: During normal compilation: no placeholder is used and the missing value causes a panic. During export when `kernel` is set, placeholders are always enabled so this behaves like `true`.

`auto`: The default. Behaves like `false` when reading from a regular notebook, and `true` when export is enabled (`kernel` not `none`) or when the notebook is undefined (`nb: none`).

A function: A placeholder is obtained by calling the function with the cell specification of the missing value as positional argument. For code blocks passed to `execute` and `evaluate`, the cell specification is the code block (raw element) itself (except when the raw text + cell header is ambiguous, in which case the cell ID is used as cell spec). For calls such as `#output("my-cell")`, the specification is `"my-cell"`.

Any other value: The value itself is used as placeholder. The value can be for example a string, number, content or `none`.

Examples:

```
// Show placeholder text if execution result no available
#evaluate(`1+1`, placeholder: [(computation)])

// Show source in red if execution result no available
#evaluate(`1+1`, placeholder: text.with(red))
```

By default a placeholder for `execute`, `Cell`, `In` or `Out` is rendered as a block element, while for `output` and `evaluate` a heuristic is used: the placeholder is rendered as block if the cell specification is an inline raw element, and as block otherwise.

6. Modules and Utility Functions

Callisto also exports various variables and utility functions.

6.1. Module `callisto`

Apart from the functions described in the previous sections, the top-level Callisto module exports the following functions and variables:

`themes`

A dictionary holding the built-in themes: `plain`, `notebook` and `neat`.

`default-formats`

A dictionary holding the default formats.

`default-handlers`

A dictionary holding the default handlers.

`export-names`

This function returns all the `export-name` values used in the current document.

```
export-names() → array
```

This function must be called with context. It can be used to automatically stage all the exported notebooks. Example:

```
#context for name in callisto.export-names() {
  callisto.stage-notebook(export-name: name, export-label: <notebook>)
}
```

See Section 3.5.1 for a complete example.

`handle`

This function is used in handlers to defer processing of a value to another handler.

```
handle(
  mime: str,
  ctx: dictionary,
  ..args,
  any,
) → any
```

`mime:` The "MIME type" of the handler to call to process the value passed as positional argument.

`ctx:` The handler context. This dict is usually received by the handler that makes the `handle` call, and simply forwarded to `handle`. The dict contains the user configuration as well as contextual values relevant to the output item or cell being processed.

The positional argument is the data to process. Additional arguments are forwarded to the handler called by `handle`.

For example, here is the definition of the default handler for Markdown cells:

```
#let markdown-cell(cell, ctx: none, ..args) = {
  parbreak()
  handle(cell.source, mime: "markdown-generic", ctx: ctx, ..args)
  parbreak()
}
```

In this case the data to process is a cell. This handler defers the Markdown conversion to the `markdown-generic` handler, but makes sure that the result is rendered as a standalone paragraph.

And here is the default handler for raw cells:

```
#let raw-cell(cell, ctx: none, ..args) = handle(
  cell.source,
  mime: "source-code-generic",
  ctx: ctx,
  lang: ctx.raw-lang,
  ..args,
)
```

This renders the source of the raw cell using the `source-code-generic` handler. The `source-code-generic` handler accepts a `lang` argument, which is set here using the user-configured `raw-lang`.

6.2. Submodule `configuration`

`settings`

A dictionary holding all available settings with their default values. After importing Callisto, this dict can be accessed as `callisto.configuration.settings`.

6.3. Submodule `header-pattern`

This module holds the logic for converting a `cell-header-pattern` string into a regular expression and writer function, as well as utility functions to read and write cell headers.

`parse-text`

Parses a cell source to find the header and convert it to a dictionary. The returned value is a dict with `header` field holding the header dict and `code` field holding the rest of the cell source as a string.

Header fields are not processed in any way, they are returned as strings as found according to the header pattern. This means that `echo` and `output` fields if present will hold `"true"` or `"false"` rather than boolean values.

```
parse-text(
  pattern: str dictionary auto none,
  str,
) → dictionary
```

`pattern:` The header pattern as configured with `cell-header-pattern`.

The positional argument is the string to parse.

Example:

```
// Get a dict with field:
//   header: (label: "x")
//   code: "a = 2"
#callisto.header-pattern.parse-text("#| label: x\na = 2", pattern: auto)
```

`make-text`

Builds a cell header string for the given header dictionary.

```
make-text(
  pattern: str dictionary auto none,
  dictionary,
) → str
```

`pattern:` The header pattern as configured with `cell-header-pattern`.

The positional argument is the header dict to convert to a header string. All fields in the dictionary must be strings.

Example:

```
// Make string "# | label: x\n"
#callisto.header-pattern.make-text((label: "x"), pattern: auto)
```

6.4. Submodule `ansi`

This module holds the code for dealing with ANSI escape sequences in textual outputs.

`render`

Converts a string with ANSI escape sequences into styled text. How a particular ANSI sequence is rendered can be configured using the parameters that accept a `function`: the function will receive content as positional argument, and `fg` and `bg` keyword arguments for the current colors, which are always of type `color` or `none`.

When colors are reversed and one of `fg` or `bg` is `none`, the `none` value is replaced with `white` or `black` to guarantee good contrast. (When both are `none`, the reversing has no effect.)

```
render(
  palette: auto array,
  fg: color none,
  bg: color none,
  bold-is-bright: bool,
  apply-fg: function,
  apply-bg: function,
  bold: function,
  italic: function,
  overline: function,
  underline: function,
  strike: function,
  dim: function,
  conceal: function,
  str,
) → content
```

`palette:` An array of 16 colors to use for the standard ANSI colors. The default is `auto` which currently resolves to a palette based on Tango colors.

`fg:` Initial color for text (default `none`). How this color is used can be changed with `apply-fg`. By default, with `fg: none` the text color is left as-is and dimming has no effect.

`bg:` Initial background color (default `none`). How this color is used can be changed with `apply-bg`. By default, with `bg: none` the background color is left as-is .

`bold-is-bright:` If `true`, bold text in standard normal color (one of the first 8 colors in the palette) is also rendered "bright" by using the corresponding bright color from the palette. Default: `false`.

`apply-fg:` The function to apply the foreground color. The default uses `text` if `fg != none`.

`apply-bg:` The function to apply the background color. The default uses `highlight` if `bg != none`.

`bold:` The function to apply for bold text. The default uses `text(weight: "bold")`.

`italic:` The function to apply for italic text. The default uses `text(style: "italic")`.

`overline:` The function to apply for overlined text. The default uses `overline`.

`underline:` The function to apply for underlined text. The default uses `underline`.

`strike:` The function to apply for strikethrough text. The default uses `strike`.

`dim:` The function to apply for dimmed text. The default makes the text 50% transparent.

conceal: The function to apply for concealed text. The default uses `hide` to prevent secrets from leaking into compiled documents. To instead make the text invisible but still present and selectable, use for example `conceal: (it, ..args) => text(it, fill: rgb(0, 0, 0, 0))`.

strip

Strips escape sequences from the given string.

```
strip( str ) → str
```

console-block

Renders the given string as a "console block", processing ANSI escape sequences to render colors, etc. correctly.

```
console-block(
  template: function,
  str,
  ..args,
) → content
```

template: A template function to apply when rendering a console block.

The additional arguments are forwarded to the `render` function.

The default template adjusts the paragraph leading, text edges and `highlight` defaults to avoid gaps in background color between successive rows of text. This also ensures that box-drawing characters in adjacent rows connect properly.

During processing, the given string is wrapped in a raw block, but the raw block is eventually discarded by the show rule that does the rendering. This temporary raw block is used to apply the raw font as well as any user show-set rules that might be defined for `raw` elements. The final result itself is not a raw element, as the text is processed by `ansi.render` which returns styled content.

console-block-template

The default template used by `console-block`.

```
console-block-template(
  target: selector,
  text-edges: dictionary,
  inset: dictionary,
  content
) → content
```

target: The selector to be used as target for the show-set rules. Defaults to `raw.where(lang: "ansi")`.

text-edges: A dictionary with fields `top-edge` and `bottom-edge`.

inset: A dictionary with fields `top`, `bottom`, and `x`.

7. Cell Preprocessing and Header

The lower-level `cells` function (and its `cell` variant) can be used to retrieve literal cell dicts as found in the notebook JSON, with the following processing applied:

- A cell ID is generated if missing (this field is mandatory since nbformat 4.5).
- An `index` field is added with the cell index in the notebook, starting at 0.
- The cell source is normalized to be a simple string (nbformat also allows an array of strings).
- For code cells, a metadata header is processed and removed if present: By default, if the first source lines are of the form `#| key: value` (optionally with whitespace between `#` and `|`), they are treated as metadata. The key-values pairs are added to the `cell.metadata.callisto.header` dictionary, and the header lines are removed from the cell source (unless `keep-cell-header` is set to `true`).

  For example, a code cell containing the following source (spacing inconsistency intentional):

  ```
  #| label: plot1
  # | echo: false
  scatter(x)
  ```

  will have the first two lines replaced with the following two entries in the cell metadata (using dot notation for nested dictionaries):

  - `cell.metadata.callisto.header.label = "plot1"`
  - `cell.metadata.callisto.header.echo = "false"`

  The format of header lines can be changed using the `cell-header-pattern` setting.

Cell dicts returned by `cells` can be used as a form of cell specification when calling functions such as `sources`, `outputs` or `render`.

8. Handlers

A handler is a function called to process a value such as a cell source, a cell output such as a PNG image, or even a whole cell. Each handler is associated with a "MIME type", which is really an arbitrary string used to identify the kind of value being processed. In the case of rich outputs (of type `display` or `result`) which can be available in multiple formats, the item is rendered by calling the handler for the selected format. In this case the "MIME type" is a real MIME type, for example `image/png`. Other handlers use pseudo MIME types without slash character, for example `code-cell`.

Handlers offer a powerful mechanism for customization. A particular value is typically processed in several steps by chaining calls to different handlers from more abstract to more concrete. This allows the theme or the user to plug in their code at the right step in the chain.

8.1. Arguments

Handlers are always called with a positional argument for the data to render, and a `ctx` keyword argument for contextual data (see Context section below). Some handlers also take additional arguments, such as `alt`, `width` and `height` for image handlers.

When defining a handler, it is good practice to add an `..args` sink for possible extra arguments (especially as additional arguments might be introduced in a future version):

```
#let handler(data, ctx: none, ..args) = ...
```

8.1.1. Context

A `ctx` dict is passed to all handler calls and holds resolved settings (replacing most `auto` values with resolved values) as well as contextual data including at least the following fields:

- `cell-spec`: the cell specification.
- `cfg`: a dict with all the settings supported by callisto.config, using default values for settings not set by the user (this holds the non-resolved settings values). This dict can be used to call functions such as `outputs` from a handler while applying the user settings. Example:

```
#let my-handler(data, ctx: none, ..args) = {
  let outs = callisto.outputs(ctx.cell, ..ctx.cfg)
  ...
}
```

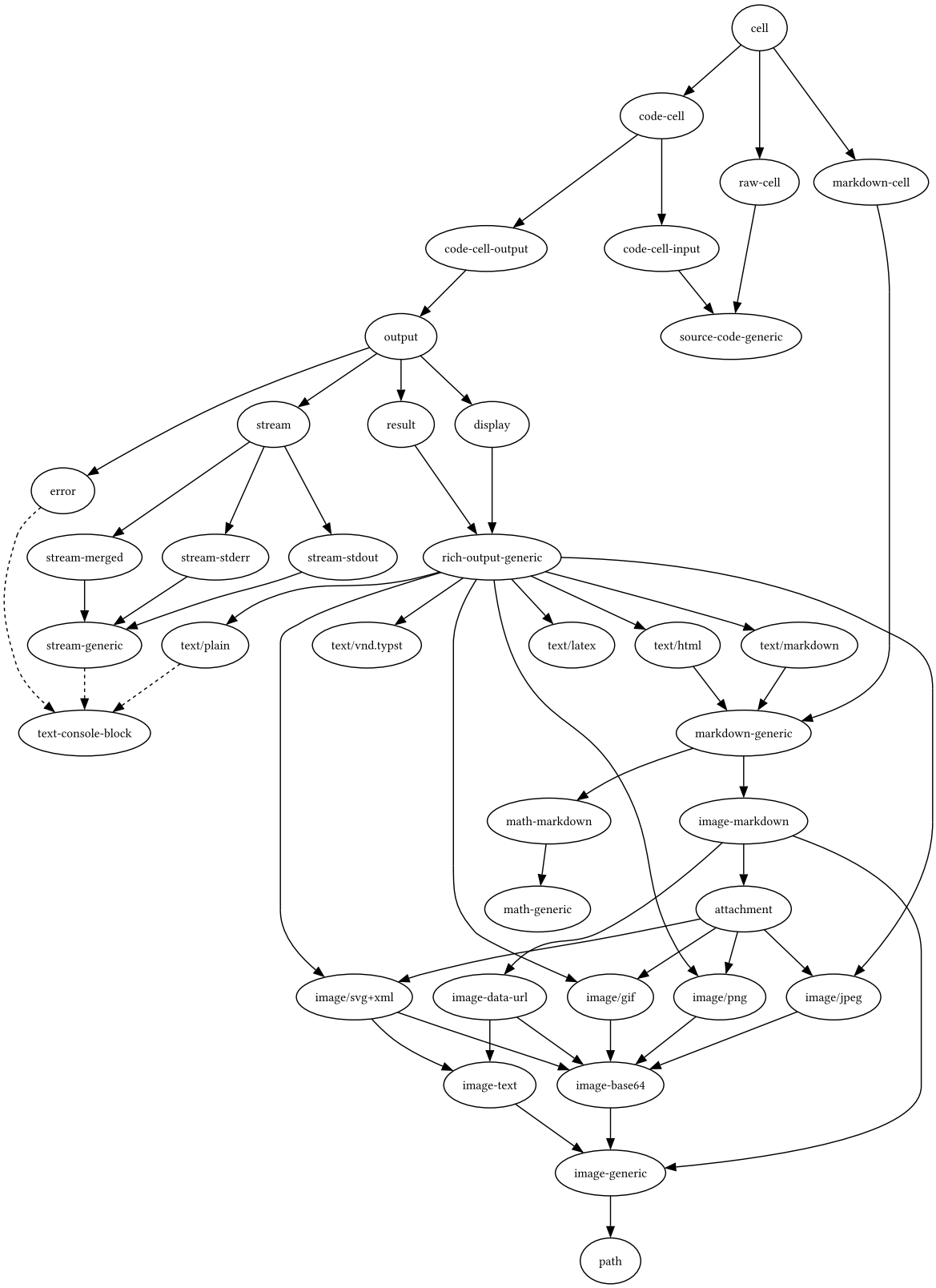
- `cell`: the dict of the cell being processed.
- `item-desc`: a dict with information on the cell item (output item or attachment) being processed if any, or `none` otherwise. When not `none`, the dict contains at least the following fields:
  - `index`: the item index in the cell output list (`none` for attachments),
  - `type`: the output type, or `"attachment"` for attachments.

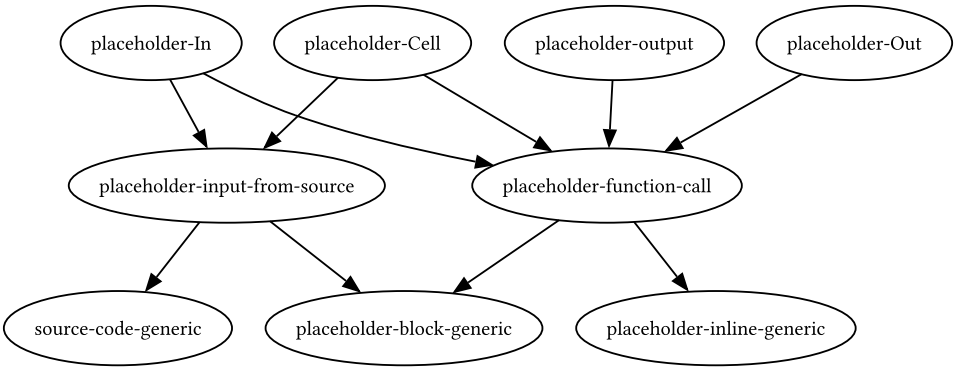
  For rich items, this dict contains also
  - `format`: the format selected for this rich item.
  - `metadata`: the format-specific metadata if present, or the whole metadata dict associated with this item otherwise.
- `latex-preamble`: a string with all the LaTeX command definitions (of the `\newcommmand` form) found in the notebook, or `none` if `gather-latex-defs` is `false`.

For example suppose the `input` setting is `auto`. When Callisto processes a cell with header line `#| echo: false`, the `auto` value resolves to `false`, so handlers are called with `ctx.cfg.input` set to `auto` and `ctx.input` set to `false`.

8.2. Default Handlers

The following diagrams shows which handlers can call which other handlers in the default configuration. Handlers with names ending in `-generic` are close to the bottom of the chain: they deal with fairly concrete value types that need to be processed by several higher-level handlers. Dashed lines indicate handlers called only during rendering.


cell
code-cell
raw-cell
markdown-cell
code-cell-output
code-cell-input
source-code-generic
output
stream
result
display
error
stream-merged
stream-stderr
stream-stdout
rich-output-generic
stream-generic
text/plain
text/vnd.typst
text/latex
text/html
text/markdown
text-console-block
markdown-generic
math-markdown
image-markdown
math-generic
attachment
image/svg+xml
image-data-url
image/gif
image/png
image/jpeg
image-text
image-base64
image-generic
path



The next sections describe these handlers in some details.

Note: The descriptions below include information on the default implementation of the handlers. These implementations are subject to change so this information is not covered by the semver stability guarantees.

8.2.1. For Cell Rendering

The following handlers take a cell dict and return a rendered cell (or part thereof).

`cell`: For any type of cell. The default delegates to the type-specific handler.

`markdown-cell`: For Markdown cells.

`code-cell`: For a whole code cell. The default delegates to `code-cell-input` and/or `code-cell-output`.

`code-cell-input`: For the source of code cells.

`code-cell-output`: For the output of code cells.

`raw-cell`: For raw cells.

8.2.2. For Output Items

The following handlers take an output item as dict and return the processed value. The dict fields depend on the output type (see below).

`output`: For any output item. The default delegates to the handler specific to the output type, such as `display`.

`display`: For display items. The data fields are:

- `data`: the item data encoded in the selected format.
- `metadata`: the metadata dictionary associated with the selected format (or with the whole output item if no format-specific metadata is found).
- `format`: the selected format, as string.

The default delegates to `rich-output-specific`.

`result`: For result items. The data fields are as for `display`. The default delegates to `rich-output-specific`.

`rich-output-generic`: For processing display and result items. The data fields are as for `display`.

`error`: For error items. The data fields are:

- `name`: the error name, as string.
- `message`: the full error message, as string.
- `traceback`: the function call stack, as an array of strings.

The default returns the message unmodified. The default themes (active during rendering) override this by processing the message through the `text-console-block` handler.

`stream`: For any stream item. The data fields are:

- `name`: the stream name: `"stdout"`, `"stderr"`, or `"all"` (when merging both `stdout` and `stderr` in a `full-streams` call with setting `stream: "all"`).
- `text`: the content as string.

The default delegates to the stream-specific handler.

`stream-stdout`: For an "stdout" item. See `stream` for the data fields.

`stream-stderr`: For an "stderr" item. See `stream` for the data fields.

`stream-merged`: For an item obtained by merging the `stdout` and `stderr` streams produced by `full-streams` with setting `stream: "all"`. See `stream` for the data fields.

`stream-generic`: Base handler used by stream-specific handlers for processing the stream content. See `stream` for the data fields. The default returns the text unmodified. The default themes (active during rendering) override this by delegating to `text-console-block`.

8.2.3. For Output Item Data

The following handlers process raw data.

The image handlers turn encoded data into an `image` element. They all accept `alt`, `width` and `height` keyword arguments. The default image handlers delegate to one of the image processing handlers in the next section.

`txt/vnd.typst`: For Typst source code. The data is the source code as string. The default renders the source code using `eval` with `mode: "markup"`.

`image/svg+xml`: For SVG images.

`image/png`: For PNG images.

`image/jpeg`: For JPEG images.

`image/gif`: For GIF images.

`text/markdown`: For Markdown text. The data is a string. The default delegates to the `markdown-generic` handler and wraps the result in a block.

`text/html`: For HTML. The data is a string. The default delegates to the `markdown-generic` handler and wraps the result in a block. Therefore, by default HTML rendering is limited to the tags supported by cmarker. Support for additional tags can be implemented using the `html` field of the `cmarker` setting.

`text/latex`: For LaTeX text. The data is a string. The default renders the string using MiTeX.

`text/plain`: For plain text. The data is a string. The default returns the string unmodified. The default themes (active during rendering) override this by delegating to `text-console-block`.

8.2.4. For Image Processing

The following handlers process image data and return an `image` element. They all accept `alt`, `width` and `height` keyword arguments.

`image-markdown`: For images in Markdown, which can refer to an attachment (an image stored in the notebook itself) or to an external file. The data is a string that holds the path to an external file, or a string of the form `"attachment:<name>"` where `<name>` is the name of an attachment in the cell dictionary. The default dispatches to the `attachment`, `image-data-url` or `image-generic` handler.

`image-data-url`: For images encoded as data URLs, for example in `<img>` HTML tags. The data is a string that includes the `data:` prefix.

`image-base64`: For base64 encoded image. The data is a base64-encoded string. The default decodes the string and delegates the result to `image-generic`.

`image-text`: For text-encoded images such as some SVGs. The data is a string. The default passes the string as `bytes` to `image-generic`.

`image-generic`: Base handler used by others. The data is of a type that can be passed to the `image` function. The default calls the `image` function with the data and extra handler arguments (such as `alt`). If the data is a string however, it is first converted to a `path` by calling the `path` handler (triggering a panic if the `path` handler was not defined by the user).

8.2.5. For LaTeX Math

The following handlers are used to process LaTeX math found in Markdown or in cell outputs.

`math-markdown`: For processing math formulas in Markdown. This handler is responsible for inserting the "preamble" of all LaTeX command definitions found in the notebook (and removing existing definitions from the equation to avoid duplicate definitions). See the `gather-latex-defs` setting.

The data is a string. The default deals with the "preamble" and passes the result to the `math-generic` handler.

`math-generic`: Base handler for math formulas. The data is a string. The default uses MiTeX to convert LaTeX formulas to Typst.

8.2.6. For Placeholders

The following handlers generate placeholders for functions that render a single cell or return a single output item when the cell or item is missing. Unless otherwise specified, the data is always `none`. These handlers can nonetheless produce informative placeholders using the function name (`Cell`, `output`, etc.) and the cell specification available in `ctx.cell-spec` (which can be the cell source itself, for example for `Cell` calls made by the `execute` function).

The default handlers all eventually delegate to `placeholder-inline-generic` and `placeholder-inline-block`.

`placeholder-Cell`: Produces a placeholder for the `Cell` function.

`placeholder-In`: Produces a placeholder for the `In` function.

`placeholder-Out`: Produces a placeholder for the `Out` function.

`placeholder-output`: Produces a placeholder for the `output` function and its variants.

`placeholder-input-from-source`: Produces a placeholder for a code cell input using the cell source. This is used in cases where the cell cannot be found in the notebook, but the source is known because it is given as a raw element in the cell specification. The data is the cell source as string.

`placeholder-function-call`: Produces a placeholder that shows the function name and cell specification of the call that requires a placeholder. The data is the function name as a string. This handler accepts an extra argument `block` which can be `true` to render as block, `false` to render inline, or `auto` to use some heuristic (the default is to render as block except for `output` placeholders when the cell specification is an inline raw element).

`placeholder-inline-generic`: Implements the generic layout of an inline placeholder. The data is the placeholder value. The default wraps the value in a box with dashed stroke.

`placeholder-block-generic`: Implements the generic layout of block placeholder. The data is the placeholder value. The default wraps the value in a block with dashed stroke.

8.2.7. Other

`attachment`: For items stored as attachment in the notebook. The data is a dictionary keyed by MIME types. This handler accepts extra arguments `metadata` and `subhandler-args`. The `metadata` value can be a simple dict of metadata, or a dict keyed by MIME types where every value is a metadata dict.

The handler should pick a format among those available in the data dict, and process the corresponding data with the appropriate handler, passing it `subhandler-args` as extra arguments.

For example this handler is used to render Markdown that refers to images stored as cell attachment. In this case `subhandler-args` might include an `alt` argument that should be forwarded to the image handler.

`source-code-generic`: For rendering source code. The data is the source as string. This handler accepts an extra `lang` argument for the source language. The default wraps the source in a raw element. This handler is called by the default handlers for code cell inputs and raw cells.

`markdown-generic`: For rendering Markdown. The data is the Markdown string. This handler should return inline content. The default uses cmarker to convert Markdown to Typst content.

`text-console-block`: For rendering text as console output, correctly handling ANSI escape sequences and adjusting text edges to have the background color and box-drawing characters connect nicely from one line to the next. The data is the string to render. The default renders the text according to the `console-text` setting.

`path`: For reading the content of files specified by path. By setting this handler the user can give Callisto permission to read any file under the project root. This is necessary to render Markdown that uses external image files. The data is a string. The default panics with a message asking the user to define the handler.

8.3. Delegation

To call a particular handler from inside your own, use `handle`. For example, the default handler for raw cells is

```
#let raw-cell(cell, ctx: none, ..args) = handle(
 cell.source,
 mime: "source-code-generic",
 ctx: ctx,
 lang: ctx.raw-lang,
 ..args,
)
```

This simply delegates the processing of the cell source to the `source-code-generic` handler, with the handler's `lang` parameter set to the notebook's `raw-lang`. The default `source-code-generic` handler is defined as

```
#let source-code-generic(txt, ctx: none, lang: none, ..args) = {
  // Ensure the source has at least one (possibly empty) line
  // (without this the raw block looks weird for empty cells)
  if txt == "" {
    txt = "\n"
  }
  raw(txt, lang: lang, block: true)
}
```

which renders the source as a raw block.

The "notebook" theme redefines the `raw-cell` handler to the following:

```
#let _raw-cell(cell, ctx: none) = block(
  spacing: 1.5em,
  width: 100%,
  inset: 0.5em,
  fill: luma(240),
  handle(cell.source, mime: "source-code-generic", ctx: ctx, lang: ctx.raw-lang),
)
```

This also calls the `source-code-generic` handler to process the cell source, but wraps the result in a block with light-gray background color.

8.4. Configuration

When Callisto processes a particular value, which handler gets called is determined as follows:

- During rendering (when the user calls `render`, `Cell`, `In`, `Out` or `execute`), the handlers defined by the theme replace the default handlers. For example the "plain" theme replaces the `text/plain`, `stream-generic` and `error` handlers with functions that delegates to the `text-console-block` handler (to convert into text styles the ANSI escape sequences that might appear in these text messages).
- During other function calls (like `source` or `outputs`), the theme handlers are *not* used unless the user set `apply-theme` to `true`.
- The handlers defined by the user through the `handlers` setting always take precedence.

9. Themes

A theme is basically a dictionary of handlers to be used in place of the default handlers during rendering (i.e. during calls to `render`, `Cell`, `In`, `Out` or `execute`).

The theme dictionary can also include an additional field `template` to define a document template. This can be useful to maintain a cohesive style in documents that mix Typst-native content with notebook cells, or to separate global styles from element-specific styles.

For example the "neat" theme defines a template function that changes the text font, increases some spacings and styles raw elements. The template can be used as follows:

```
#let (template, render) = callisto.config(nb: path("file.ipynb"), theme: "neat")

#show: template

#render()
```

9.1. Writing Themes

A theme only needs to define the handlers it wants to modify. The default handlers will be used as fallback. An existing theme can be customized by adding to its dictionary. The dictionaries for the standard themes are available in the `callisto.themes` variable.

It is good practice for theme handlers to delegate as much work as possible to other handlers, to pick up behavior configured by the user using these handlers. For example:

- call `outputs` rather than manually processing the `outputs` field in the `cell` dict (the `outputs` function will call the `output` handler on each output),
- or if manually processing `cell.output`: call the `output` handler on each output,
- or if manually processing e.g. a stream output: call the `stream` handler, etc.

9.1.1. Example: A theme that wraps cell outputs in figures

In the following, we extend the "neat" theme to wrap the cell output in a figure whenever the cell header defines a figure caption (`fig-cap` key):

```
#let caption-handler(cell, ctx: none, ..args) = {
  let value = callisto.outputs(cell, ..ctx.cfg).join()
  let header = cell.metadata.callisto.header
  if "fig-cap" in header {
    return figure(
      value,
      caption: header.fig-cap,
      // Also set alt text if fig-alt key is defined
      alt: header.at("fig-alt", default: none),
    )
  }
  return value
}

#let (render,) = callisto.config(
  nb: path("julia.ipynb"),
  theme: callisto.themes.neat + (code-cell-output: caption-handler),
)
```

Note: The `code-cell-output` handler receives the cell as positional argument. If this wasn't the case, we could still access the cell as `ctx.cell.metadata`.

9.1.2. Example: Complete code for the “neat” theme

```
#import "@preview/callisto:0.3.0"

#let _fill = rgb(233, 236, 239)
#let _inset = 8pt
#let _radius = 5pt
#let _extent = 3pt

#let _raw-block-cfg = (
  width: 100%,
  inset: _inset,
  radius: _radius,
  fill: _fill,
)

// Document template
#let _template(doc, set-fonts: true) = {
  set text(font: "Noto Sans") if set-fonts
  show raw: set text(font: "Noto Sans Mono") if set-fonts
  show heading: set text(weight: "semibold") if set-fonts

  show heading: set block(below: 1em)
  show heading.where(level: 1): set text(1.4em)
  show heading.where(level: 2): set text(1.2em)

  show raw.where(block: false): it => {
    let cfg = (fill: _fill, top-edge: 1em, bottom-edge: -0.4em)
    highlight(..cfg, radius: (left: _radius))[~#sym.wj]
    highlight(..cfg, extent: _extent, it)
    highlight(..cfg, radius: (right: _radius))[#sym.wj~]
  }

  show raw.where(block: true): set block(.._raw-block-cfg)

  show math.equation: set text(1.1em)

  doc
}

#let _code-cell-input(cell, ctx: none, block-args: none, ..args) = {
  let has-output = ctx.output and cell.outputs.len() > 0
  set text(rgb("#005979"))
  show raw: set block(.._raw-block-cfg, ..block-args, above: 1em)
  show raw: set block(below: 1em) if not has-output
  handle(
    cell.source,
    mime: "source-code-generic",
    ctx: ctx,
    lang: ctx.lang,
  )
}

#let _code-cell-output(cell, ctx: none, ..args) = {
  let outs = callisto.outputs(cell, ..ctx.cfg)
  if outs.len() == 0 { return }
```

```
  // Override template show rule for raw blocks
  // (we don't want simple text outputs to be shown in rounded gray rects)
  show raw: set block(width: auto, inset: 0pt, radius: 0pt, fill: none)
  block(
    .._raw-block-cfg,
    fill: none,
    above: if ctx.input { 0pt } else { 1em },
    below: 1em,
    outs.join(),
  )
}

#let _placeholder-input-from-source(source, ctx: none, ..args) = {
  let cell = (source: source)
  let block-args = (stroke: (dash: "dashed"))
  ctx.output = false
  return _code-cell-input(cell, ctx: ctx, block-args: block-args)
}

#let theme = callisto.themes.plain + (
  template: _template,
  code-cell-input: _code-cell-input,
  code-cell-output: _code-cell-output,
  placeholder-input-from-source: _placeholder-input-from-source,
)
```