marimo Examples

# Recipes

This page includes code snippets or "**recipes**" for a variety of common tasks. Use them as building blocks or examples when making your own notebooks.

In these recipes, **each code block represents a cell**.

## Control Flow

### Show an output conditionally

**Use cases.** Hide an output until a condition is met (*e.g.*, until algorithm parameters are valid), or show different outputs depending on the value of a UI element or some other Python object

**Recipe.**

1. Use an `if` expression to choose which output to show.

```
# condition is a boolean, True of False
condition = True
"condition is True" if condition else None
```

## Run a cell on a timer

**Use cases.**

- Load new data periodically, and show updated plots or other outputs. For example, in a dashboard monitoring a training run, experiment trial, real-time weather data, …
- Run a job periodically

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create a `mo.ui.refresh` timer that fires once a second:

```
refresh = mo.ui.refresh(default_interval="1s")
# This outputs a timer that fires once a second
refresh
```

3. Reference the timer by name to make this cell run once a second

```
import random

# This cell will run once a second!
refresh

mo.md("#" + "🌿" * random.randint(1, 10))
```

**Requires 'on cell change' autorun**

For this to work, the runtime configuration's `on cell change` should be set to `autorun`

## Require form submission before sending UI value

**Use cases.** UI elements automatically send their values to the Python when they are interacted with, and run all cells referencing the elements. This makes marimo notebooks responsive, but it can be an issue when the downstream cells are expensive, or when the input (such as a text box) needs to be filled out completely before it is considered valid. Forms let you gate submission of UI element values on manual confirmation, via a button press.

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create a submittable form.

```
form = mo.ui.text(label="Your name").form()
form
```

3. Get the value of the form.

```
form.value
```

### Stop execution of a cell and its descendants

**Use cases.** For example, don't run a cell or its descendants if a form is unsubmitted.

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create a submittable form.

```
form = mo.ui.text(label="Your name").form()
form
```

3. Use `mo.stop` to stop execution when the form is unsubmitted.

```
mo.stop(form.value is None, mo.md("Submit the form to continue"))

mo.md(f"Hello, {form.value}!")
```

## Grouping UI elements together

### Create an array of UI elements

**Use cases.** In order to synchronize UI elements between the frontend and backend (Python), marimo requires you to assign UI elements to global variables. But sometimes you don't know the number of elements to make until runtime: for example, maybe

you want to make a list of sliders, and the number of sliders to make depends on the value of some other UI element.

You might be tempted to create a Python list of UI elements, such as `l = [mo.ui.slider(1, 10) for i in range(number.value)]`: *however, this won't work, because the sliders are not bound to global variables*.

For such cases, marimo provides the "higher-order" UI element `mo.ui.array`, which lets you make a new UI element out of a list of UI elements: `l = mo.ui.array([mo.ui.slider(1, 10) for i in range(number.value)])`. The value of an `array` element is a list of the values of the elements it wraps (in this case, a list of the slider values). Any time you interact with any of the UI elements in the array, all cells referencing the array by name (in this case, "`l`") will run automatically.

**Recipe.**

1. Import packages.

```
import marimo as mo
```

2. Use `mo.ui.array` to group together many UI elements into a list.

```
import random

# instead of random.randint, in your notebook you'd use the value of
# an upstream UI element or other Python object
array = mo.ui.array([mo.ui.text() for i in range(random.randint(1, 10))])
array
```

3. Get the value of the UI elements using `array.value`

```
array.value
```

## Create a dictionary of UI elements

**Use cases.** Same as for creating an array of UI elements, but lets you name each of the wrapped elements with a string key.

**Recipe.**

1. Import packages.

```
import marimo as mo
```

2. Use `mo.ui.dictionary` to group together many UI elements into a list.

```
import random

# instead of random.randint, in your notebook you'd use the value of
# an upstream UI element or other Python object
dictionary = mo.ui.dictionary({str(i): mo.ui.text() for i in range(random.randint(1, 10))})
dictionary
```

3. Get the value of the UI elements using `dictionary.value`

```
dictionary.value
```

## Embed a dynamic number of UI elements in another output

**Use cases.** When you want to embed a dynamic number of UI elements in other outputs (like tables or markdown).

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Group the elements with

`mo.ui.dictionary` or `mo.ui.array`, then retrieve them from the container and display them elsewhere.

```
import random

n_items = random.randint(2, 5)

# Create a dynamic number of elements using `mo.ui.dictionary` and
# `mo.ui.array`
elements = mo.ui.dictionary(
    {
        "checkboxes": mo.ui.array([mo.ui.checkbox() for _ in range(n_items)]),
        "texts": mo.ui.array(
            [mo.ui.text(placeholder="task ...") for _ in range(n_items)]
        ),
    }
)

mo.md(
    f"""
    Here's a TODO list of {n_items} items\n\n
    """
    + "\n\n".join(
        # Iterate over the elements and embed them in markdown
        [
            f"{checkbox} {text}"
            for checkbox, text in zip(
                elements["checkboxes"], elements["texts"]
```

```
            )
        ]
    )
)
```

3. Get the value of the elements

```
elements.value
```

## Create a hstack (or vstack) of UI elements with `on_change` handlers

**Use cases.** Arrange a dynamic number of UI elements in a hstack or vstack, for example some number of buttons, and execute some side-effect when an element is interacted with, e.g. when a button is clicked.

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create buttons in `mo.ui.array` and pass them to hstack -- a regular Python list won't work. Make sure to assign the array to a global variable.

```
import random


# Create a state object that will store the index of the
# clicked button
get_state, set_state = mo.state(None)
```

```
# Create an mo.ui.array of buttons - a regular Python list won't work.
buttons = mo.ui.array(
    [
        mo.ui.button(
            label="button " + str(i), on_change=lambda v, i=i: set_state(i)
        )
        for i in range(random.randint(2, 5))
    ]
)

mo.hstack(buttons)
```

3. Get the state value

```
get_state()
```

## Create a table column of buttons with `on_change` handlers

**Use cases.** Arrange a dynamic number of UI elements in a column of a table, and execute some side-effect when an element is interacted with, e.g. when a button is clicked.

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create buttons in `mo.ui.array` and pass them to `mo.ui.table`. Make sure to assign the table and array to global variables

```
import random


# Create a state object that will store the index of the
# clicked button
get_state, set_state = mo.state(None)

# Create an mo.ui.array of buttons - a regular Python list won't work.
buttons = mo.ui.array(
    [
        mo.ui.button(
            label="button " + str(i), on_change=lambda v, i=i: set_state(i)
        )
        for i in range(random.randint(2, 5))
    ]
)

# Put the buttons array into the table
table = mo.ui.table(
    {
        "Action": ["Action Name"] * len(buttons),
        "Trigger": list(buttons),
    }
)
table
```

3. Get the state value

```
get_state()
```

## Create a form with multiple UI elements

**Use cases.** Combine multiple UI elements into a form so that submission of the form sends all its elements to Python.

**Recipe.**

1. Import packages.

```
import marimo as mo
```

2. Use `mo.ui.form` and `Html.batch` to create a form with multiple elements.

```
form = mo.md(
   r"""
   Choose your algorithm parameters:

   - $\epsilon$: {epsilon}
   - $\delta$: {delta}
   """
).batch(epsilon=mo.ui.slider(0.1, 1, step=0.1), delta=mo.ui.number(1, 10)).form()
form
```

3. Get the submitted form value.

```
form.value
```

## Populating form with pre-defined examples

**Use cases.** To give examples of how a filled form looks like. Useful for illustrating complex API requests or database queries. The form can also be populated from URL query parameters (notebook example).

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create dropdown of examples

```
examples = mo.ui.dropdown(
    options={
        "ex 1": {"t1": "hello", "t2": "world"},
        "ex 2": {"t1": "marimo", "t2": "notebook"},
    },
    value="ex 1",
    label="examples",
)
```

3. Create form from examples.

```
form = (
    mo.md(
        """
    ### Your form

    {t1}
    {t2}
"""
    )
    .batch(
        t1=mo.ui.text(label="enter text", value=examples.value.get("t1", "")),
        t2=mo.ui.text(label="more text", value=examples.value.get("t2", "")),
    )
```

```
    .form(
        submit_button_label="go"
    )
)
```

4. Run pre-populated from or recompute with new input.

```
output = (
    " ".join(form.value.values()).upper()
    if form.value is not None
    else " ".join(examples.value.values()).upper()
)
examples, form, output
```

# Working with buttons

## Create a button that triggers computation when clicked

**Use cases.** To trigger a computation on button click and only on button click, use `mo.ui.run_button()`.

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create a run button

```
button = mo.ui.run_button()
```

```
button
```

3. Run something only if the button has been clicked.

```
mo.stop(not button.value, "Click 'run' to generate a random number")

import random
random.randint(0, 1000)
```

## Create a counter button

**Use cases.** A counter button, i.e. a button that counts the number of times it has been clicked, is a helpful building block for reacting to button clicks (see other recipes in this section).

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Use `mo.ui.button` and its `on_click` argument to create a counter button.

```
# Initialize the button value to 0, increment it on every click
button = mo.ui.button(value=0, on_click=lambda count: count + 1)
button
```

3. Get the button value

```

```

```
button.value
```

## Create a toggle button

**Use cases.** Toggle between two states using a button with a button that toggles between `True` and `False`. (Tip: you can also just use `mo.ui.switch`.)

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Use `mo.ui.button` and its `on_click` argument to create a toggle button.

```
# Initialize the button value to False, flip its value on every click.
button = mo.ui.button(value=False, on_click=lambda value: not value)
button
```

3. Toggle between two outputs using the button value.

```
mo.md("True!") if button.value else mo.md("False!")
```

## Re-run a cell when a button is pressed

**Use cases.** For example, you have a cell showing a random sample of data, and you want to resample on button press.

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create a button without a value, to function as a *trigger*.

```
button = mo.ui.button()
button
```

3. Reference the button in another cell.

```
# the button acts as a trigger: every time it is clicked, this cell is run
button

# Replace with your custom logic
import random
random.randint(0, 100)
```

## Run a cell when a button is pressed, but not before

**Use cases.** Wait for confirmation before executing downstream cells (similar to a form).

**Recipe.**

1. Import packages

```

```

```
import marimo as mo
```

2. Create a counter button.

```
button = mo.ui.button(value=0, on_click=lambda count: count + 1)
button
```

3. Only execute when the count is greater than 0.

```
# Don't run this cell if the button hasn't been clicked, using mo.stop.
# Alternatively, use an if expression.
mo.stop(button.value == 0)

mo.md(f"The button was clicked {button.value} times")
```

## Reveal an output when a button is pressed

**Use cases.** Incrementally reveal a user interface.

**Recipe.**

1. Import packages

```
import marimo as mo
```

2. Create a counter button.

```
button = mo.ui.button(value=0, on_click=lambda count: count + 1)
```

```
button
```

3. Show an output after the button is clicked.

```
mo.md("#" + "🌿" * button.value) if button.value > 0 else None
```

# Caching

## Cache function outputs in memory

**Use case.** Because marimo runs cells automatically as code and UI elements change, it can be helpful to cache expensive intermediate computations. For example, perhaps your notebook computes t-SNE, UMAP, or PyMDE embeddings, and exposes their parameters as UI elements. Caching the embeddings for different configurations of the elements would greatly speed up your notebook.

**Recipe.**

1. Use `mo.cache` to cache function outputs given inputs.

```
import marimo as mo

@mo.cache
def compute_predictions(problem_parameters):
    # replace with your own function/parameters
    ...
```

Whenever `compute_predictions` is called with a value of `problem_parameters` it has not seen, it will compute the predictions and store them in a cache. The next time it is called with the same parameters, instead of recomputing the predictions, it will

return the previously computed value from the cache.

## Persistent caching for very expensive computations

**Use case.** If you are using marimo to capture very compute intensive results, you may want to save the state of your computations to disk. Ideally, if you update your code, then this save should be invalidated. It may also be advantageous to add UI elements to explore your results, without having to recompute expensive computations. You can achieve this with `mo.persistent_cache`.

**Recipe.**

1. Use `mo.persistent_cache` to cache blocks of code to disk.

```
import marimo as mo

with mo.persistent_cache("my_cache"):
    # This block of code, and results will be cached to disk
    ...
```

If the execution conditions are the same, then cache will load results from disk, and populate variable definitions.