Getting started with the marimo notebook

For educators

Akshay Agrawal
akshayka@cs.stanford.edu

Contents

1 Python notebooks

Python notebooks. A notebook is a type of program in which code is organized in a sequence of code blocks, called *cells*. A Python notebook is a notebook in which each cell is Python code.

Cells are typically displayed in a graphical user interface, or *editor*, that lets you edit their code and execute them interactively. Unlike traditional Python scripts, which run from start to finish without human intervention, notebooks allow you to execute a section of code and inspect the execution results — at which point you may choose to execute another cell, modify code, or otherwise author entirely new cells. In this way notebooks allow you to author programs or explore data incrementally, using intermediate execution results as feedback for what to author next, which can be of great help when working with data.

Outputs. The last expression of a cell is its *output*, which is visualized by the editor below the cell. Outputs can be arbitrary Python objects. When working through this primer, you will frequently output plots, matrices, and other data structures.

For example, this cell visualizes a matplotlib plot:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 plt.plot(np.array([0, 1, 2]))
5 plt.gca()
```

In this code block, the cell visualizes a plot by outputting `plt.gca()` (“get current axes”) in line 5.

Markdown. In addition to Python, notebooks allow you to document your work with *Markdown*. Markdown is a simple markup language for formatted text. It is common to intersperse cells with expository Markdown that explains work or demarcates sections. Markdown is very widely used, and there are many free tutorials available online that teach its syntax. Nearly all Markdown implementations in notebooks also allow you to write mathematical notation with LaTeX.

Documents. A notebook, consisting of cells, outputs, and Markdown, can be interpreted as a *document*, in addition to a program. Notebook editors allow you to export your notebook as a standalone HTML or PDF file that can be shared with others.

Traditional notebooks. Traditional notebooks are similar to read-eval-print loops. In a traditional notebook, running a cell imperatively mutates memory, without running dependent cells. As a result, notebooks accumulate *hidden state*. This means that code and outputs can easily become out of sync; for example, the code on the page may say `x = 0`, but depending on what cells were run previously, an output may show a different value for `x`, such as `1`. The accumulation of hidden state is not only confusing (and different from other interactive programming models, such as spreadsheets), it has also contributed to a reproducibility crisis in research, documented in various studies.

The most well-known traditional notebook is the Jupyter notebook, paired with the imperative IPython kernel. While Jupyter notebooks have been widely used in research and industry, they suffer not only from reproducibility issues but also from a lack of reusability, incompatibility with version control, and difficult-to-use interactive widgets.

2 The marimo notebook

This primer is about the *marimo* notebook. marimo (stylized in lowercase) is an open-source notebook designed specifically for the Python language. Unlike traditional notebooks, marimo keeps code and outputs in sync: run a cell and marimo automatically runs dependent cells (or lazily marks them as stale, your choice). This *reactive execution* eliminates hidden state and hidden bugs, while also giving users immediate feedback as they edit and run cells. marimo solves widely-documented problems with traditional notebooks, including reproducibility (through reactive execution and package management), reusability (as Python modules and scripts), and interactivity (through easy-to-use UI elements like sliders).

Designed for educators and researchers. marimo was designed for educators and researchers by someone who identifies as both, with input from Stanford scientists who sought a more interactive medium for computer science education and a more reproducible environment for computational research. In its original design, marimo drew significant inspiration from `Pluto.jl`, a reactive Julia notebook from MIT designed for education.

Documentation. marimo and its documentation are freely available at

`https://docs.marimo.io`

The online documentation contains a quickstart, comprehensive user guides, and several examples.

Code repository. marimo is free and open-source, available under the Apache 2.0 License:

`https://github.com/marimo-team/marimo`.

We welcome feedback, which you may provide by opening an issue in our GitHub repository.

Examples. A collection of example notebooks is available at our online gallery, at

https://marimo.io/gallery

2.1 Setup

marimo can be used locally, through our open-source software package, or online, through our free cloud-hosted marimo notebook service called molab. In this subsection we provide setup guides for both, though molab is usually far easier for students to use than the local setup.

2.1.1 Local setup

marimo is open source and available on the Python Package Index and Conda Forge. You can install marimo with any Python package manager, such as `uv`, `pip`, or `conda`. For example:

```
$ pip install marimo
```

In the remainder of this section, we provide simple instructions on how to get started with marimo on your local machine. While we focus on using marimo's browser-based editor, marimo also has a VS Code extension, which you can find online in the VS Code and Cursor marketplaces.

Installing Python. First, install the `uv` package manager following the instructions available on its website, https://docs.astral.sh/uv/. Then, install Python with

```
$ uv python install 3.13 --default
```

This adds the installed version of Python to your system `PATH`, letting you run Python from the command line with the `python` command.

Installing the marimo command-line interface. The main entry point to marimo is the command-line interface (CLI). For simplicity, we recommend installing and invoking the marimo CLI using the `uvx` command:

```
$ uvx marimo
```

Here, `uvx` provides access to the marimo CLI, downloading it in an isolated environment if necessary before invoking it. If you and your students are familiar with virtual environments and Python projects, you can install and invoke marimo however you like, with or without `uv`/`uvx`.

Running notebooks. To create or edit an existing notebook, use the following command:

```
$ uvx marimo edit --sandbox my_notebook.py
```

This opens marimo's browser-based editor, which provides a fully featured programming environment for working with marimo notebooks. To run a notebook server capable of opening and editing multiple notebooks, run the `edit` command without a filename:

```
$ uvx marimo edit --sandbox
```

To learn more about marimo's editor, visit

https://docs.marimo.io/guides/editor_features.

Installing third-party packages. marimo comes with a built-in user interface for package management. When a user imports a library such as `numpy`, they are prompted by the marimo editor to install it. If the notebook was started with the `-sandbox` flag, marimo will do something more: it will record the installed package (and its version) as metadata in the notebook file. The next time the notebook is run, marimo will automatically run it in temporary isolated virtual environment (a "sandbox") containing just those packages. This eliminates the error-prone task of managing a pyproject.toml or requirements.txt file, while also guaranteeing that your notebook won't fail because of missing packages.

To learn more about marimo's package management features, visit

https://docs.marimo.io/guides/package_management/

Convert Jupyter notebooks to marimo notebooks. Convert a Jupyter notebook to a marimo notebook with:

```
$ uvx marimo convert my_notebook.ipynb -o my_notebook.py
```

To learn more about migrating from Jupyter, visit

https://docs.marimo.io/guides/coming_from/jupyter/

2.1.2 Online setup

molab, marimo's free cloud-hosted notebook service, is the easiest way for students to get up and running with marimo:

https://molab.marimo.io/notebooks

It is especially useful for students who are uncomfortable using their machine's command line or installing Python, since it makes marimo accessible as a self-contained web application. If you are familiar with Google Colab, you will find molab's product experience similar, but with all the benefits that marimo has over traditional notebooks.

Sharing notebooks. Notebooks created on molab are public but not discoverable by default, and can be shared with others by URL. Students can also download their notebooks from molab, in marimo's own format or as ipynb or PDFs, making it easy for them to submit their work to grading systems such as Gradescope.

Preview notebooks hosted on GitHub. Visit

https://molab.marimo.io/github

to preview notebooks hosted on GitHub as molab notebooks. This service provides a unique URL for your notebook that remains up-to-date even as your notebook changes on GitHub. Here is a sample preview created by the service:

https://molab.marimo.io/github/marimo-team/marimo/blob/main/marimo/_tutorials/intro.py

If you store homework notebooks on GitHub, this provides a convenient way to share the most up-to-date versions of your notebooks with your students. From the preview page, students can fork the notebook into their own workspace.

2.2 Concepts

In this subsection, we explain the basic concepts of marimo, with an emphasis on how it is different from traditional notebooks and on features relevant for education.

Built-in tutorials. marimo comes packaged with interactive tutorials that are themselves marimo notebooks. As an alternative to reading this subsection, students may prefer stepping through the built-in tutorials. To open the introductory tutorial, run

```
$ uvx marimo tutorial intro
```

from the command line. To see a list of all tutorials, run

```
$ uvx marimo tutorial --help
```

2.2.1 Reactive execution

When a cell is run, marimo reacts by automatically running all other cells that reference any of the variables it defines; delete a cell and marimo invalidates other cells that depended on its variables. This reactive execution keeps code and outputs in sync, eliminating hidden state and along with it a large class of silent bugs associated with traditional notebooks. Reactive execution also gives students immediate feedback, which can greatly increase learning outcomes.

For notebooks with long-running cells, you may choose to configure marimo to run cells lazily, marking them as stale instead of automatically running them. Learn more about reactive execution at our documentation:

https://docs.marimo.io/guides/reactivity/

2.2.2 Variables

In order to implement reactive execution, marimo imposes a constraint on your code: each variable must be defined in just one cell. If you accidentally define a variable in two (or more) cells, marimo will raise an error. For this reason, when using marimo you should define global variables sparingly, wrapping intermediate variable definitions in functions when possible. This not only enables reactivity, but it also results in better code.

Local variables. marimo's rule on variable naming has one exception. Variables that begin with an underscore are made *local* to a cell, and as such their names can be bound in multiple cells. This is helpful for loop variables, which for example may be written as

```
for _i in range(10):
  print(_i)
```

2.2.3 User interface elements

marimo comes packaged with user interface (UI) elements like numerical sliders, dropdowns, and selectable scatterplots, and more. To create a UI element, you must first import the `marimo` library into your notebook.

```
import marimo as mo
```

UI elements are exposed through `marimo`'s `ui` module. Create a UI element assign it to a variable, and output it in the same cell.

```
x = mo.ui.slider(start=1, stop=10, step=1, label="$x$")
x
```

Access its associated value through the `value` attribute in any cell other than the one defining the UI element.

```
x.value
```

When you interact with a UI element (for example, sliding the knob of a slider), all other cells that reference the UI element will run automatically, thanks to reactive execution. This makes it easy for students to rapidly answer what-if questions about data and algorithms: If I change this one input, how does that affect the output? For a rich interactive educational notebook, see this one from Stanford on signal decomposition:

https://molab.marimo.io/notebooks/nb_3gk1j4rzKeFpz8rNwrVU5h/app

Learn more about interactive elements at our documentation:

https://docs.marimo.io/guides/interactivity/

2.2.4 Testing

marimo has built-in support for pytest, making it easy to provide students with sanity-checks for their code. Simply import `pytest` at the beginning of your notebook; then, any cells that contain only functions starting with `test_` will be evaluated by pytest. This can be used to provide students with descriptive and immediate feedback on the correctness of their code as they implement their function, guiding them to a correct solution without outright giving them the answer.

A common pattern is the following. In one cell, provide a partial implementation of a function that a student needs to implement, such as:

```
def add(x, y):
  """A function that returns the sum of x and y."""
  # BEGIN YOUR CODE
  pass
  # END YOUR CODE
```

Then, in another cell, write tests, such as:

```
@pytest.mark.parametrize(
    ("x", "y", "z"),
    [
        (5, 6, 11),
        (2.1, 2.3, 4.4),
    ],
)
def test_add(x, y, z):
    assert add(x, y) == z

def test_add_type():
    assert isinstance(add(4, 2.1), float)
```

Whenever the student runs the cell defining `add`, the test cell will run and provide them with feedback.

Learn more about testing at our documentation:

https://docs.marimo.io/guides/testing/pytest/

2.2.5 Reusable functions and classes

marimo is stored as a pure Python program; not only does it make it easy to version notebooks with Git and edit in text editors, but it also enables code reuse, such as

```
from my_notebook import my_function, MyClass
```

Such code reuse is impossible in traditional notebooks, which are either stored as JSON or (unlike marimo) execute the whole notebook on import. This is an advanced feature of marimo, and requires the use of a special cell called the *setup cell*. Learn more about this feature at our documentation:

https://docs.marimo.io/guides/reusing_functions/