Rituals and Patterns

Saturday, May 4, 2019 5:22 PM

There are three ways to obtain a usable computer program.

1. Copy someone else's, or
2. Copy pieces of your program from others, or
3. Try to understand the pieces, and put them together in new ways.

We will start with 1. and work up to 3.

In (1), you are placing the responsibility for correctness on someone else.

- If that person is particularly studious, the program might be documented sufficiently that you can use it.
- If not, it can be more difficult to use it than to write another.

Important documentation:

1. **Preconditions:** statements that must be true before the program will work.
2. **Postconditions:** statements that will be true after the program runs, given that the preconditions are met beforehand.

Preconditions and postconditions form parts of a **contract** about the program.

If preconditions are met beforehand, then

postconditions will be met afterward.

A program is correct if it obeys that contract, and flawed if not.

Several important facts about preconditions and postconditions:

1. If preconditions are not met, then postconditions are not guaranteed.
2. Nothing is claimed about what happens if preconditions are not met. What happens might still be reasonable – or not!

Thus, to use a program someone else has written, you need to

1. Assure the preconditions the program needs.
2. Match the postconditions to your needs.

I sometimes call a program with incompletely documented preconditions a **ritual**.

- Sometimes it works.
- Sometimes it doesn't.
- One usually cannot determine why.

We call a program with completely documented preconditions and postconditions a **pattern**:

- It always works.
- Provided you give it something reasonable as input.

Example: a pattern for drawing a histogram.

Preconditions: x is set to a list of values.
Postconditions: a histogram is drawn of those values.

The pattern is in one cell:

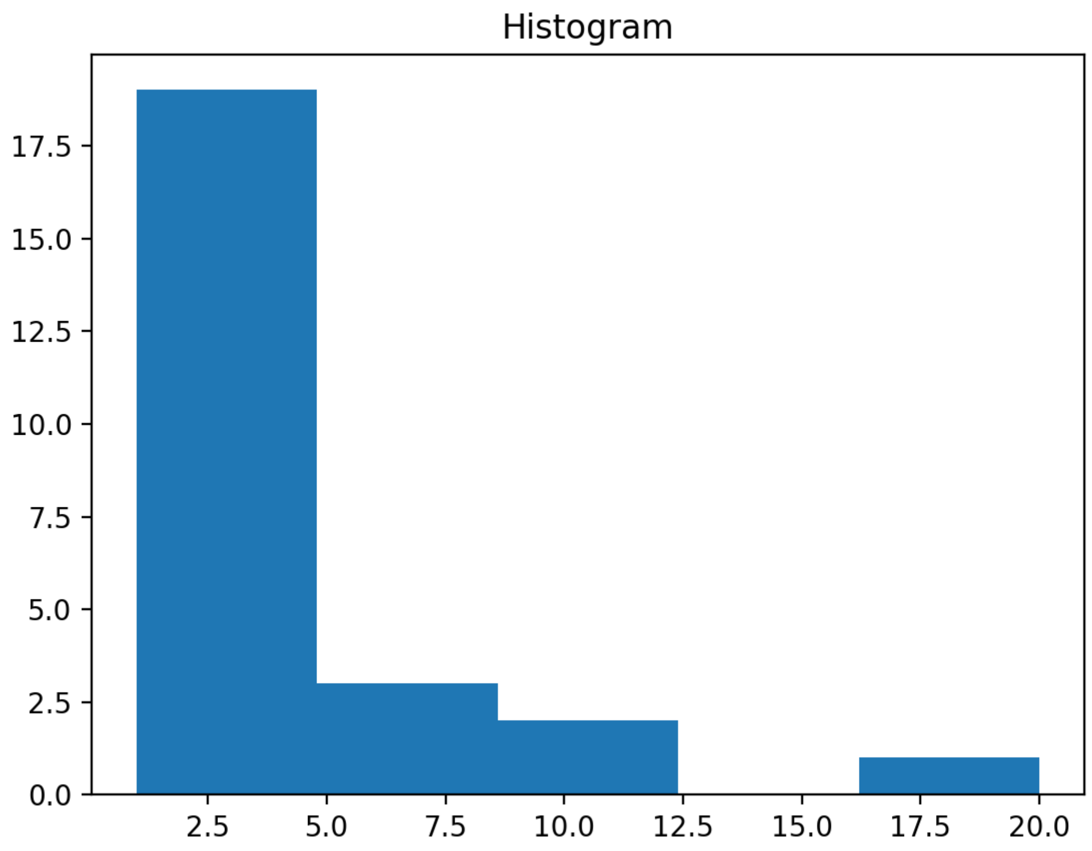
```
%matplotlib notebook
import matplotlib
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
n, bins, patches = ax.hist(data, num_bins)
ax.set_title(r'Histogram')
plt.show()
```

The data is in a prior cell:

```
data = [1, 4, 5, 2, 3,
        4, 4, 7, 8, 2,
        2, 2, 3, 1, 1,
        1, 2, 20, 2, 2,
        9, 10, 1, 2, 3]
num_bins = 5
```

And if you run both cells in order, you get this:

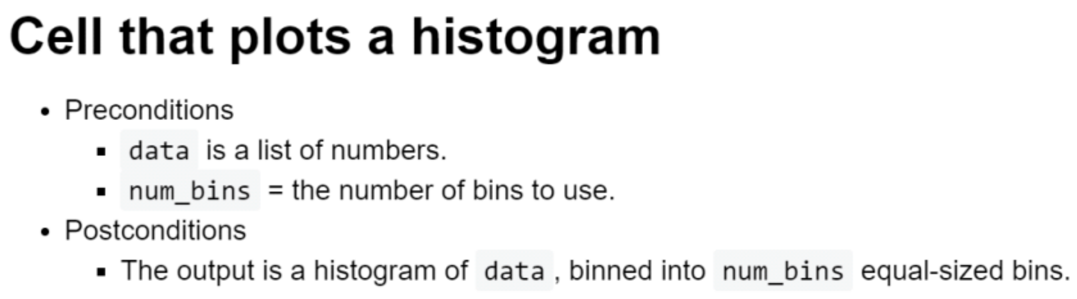


It is usually a good idea to document a cell in the previous cell. For this you might utilize "markdown" format, as follows:

```
# Cell that plots a histogram

- Preconditions
    - `data` is a list of numbers.
    - `num_bins` = the number of bins
      to use.
- Postconditions
    - The output is a histogram of `data`,
      binned into `num_bins` equal-sized
      bins.
```

This appears in Jupyter as:



Cell that plots a histogram

- Preconditions
    - `data` is a list of numbers.
    - `num_bins` = the number of bins to use.
- Postconditions
    - The output is a histogram of `data`, binned into `num_bins` equal-sized bins.

Thus, there are several components to a good pattern:

1. The code that does the work.
2. Documentation of what is required.
3. A prior cell in which those requirements are met.