SD5913 · WEEK 04

Interfaces

One control, one clear response.

Today

01 Your first plot, and the final vote for our mark

02 Turn a picture into something a person can use

03 One action travels through a Streamlit app

04 Waiting, polling and callbacks

05 APIs: tide records and a generated image

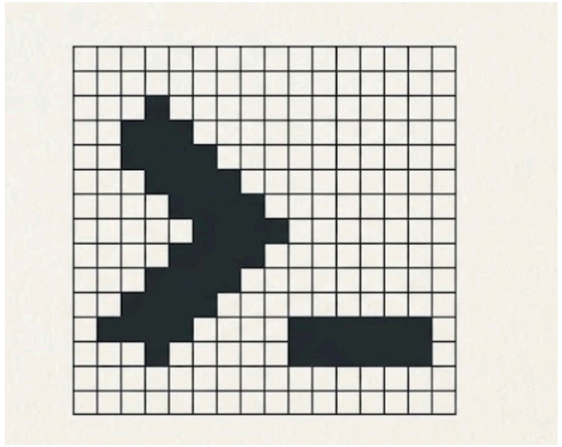
06 One test, the workshop and Assignment 2

QUESTION · IMAGE UPLOAD

Upload your first plot.

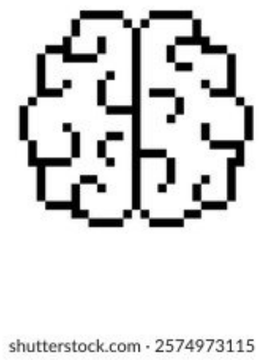
Caption, 50 characters or fewer: phenomenon · source

The class vote: three finalists



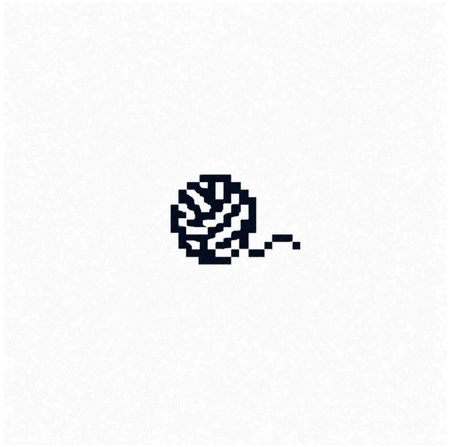
Mark 38

Score 4.04
28 wins · 5 losses · 33 comparisons



Mark 04

Score 3.40
27 wins · 6 losses · 33 comparisons

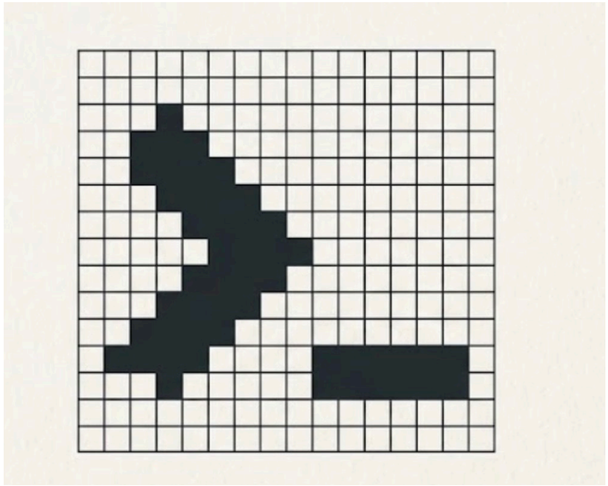


Mark 18

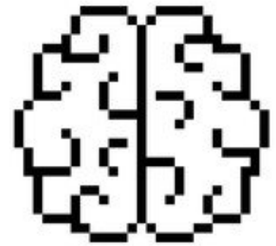
Score 3.01
26 wins · 7 losses · 33 comparisons

THE FINAL THREE

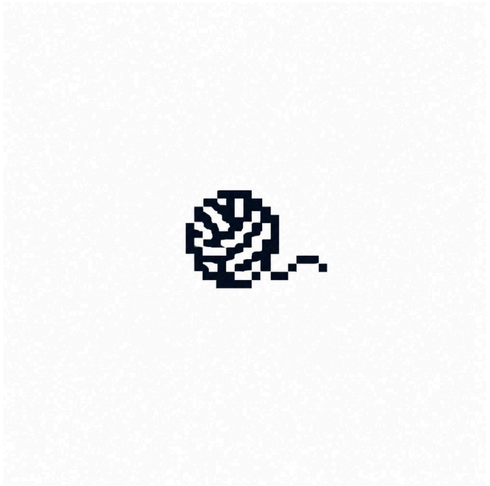
Which mark should represent SD5913?



A — Mark 38



B — Mark 04



C — Mark 18

Four words for one interaction

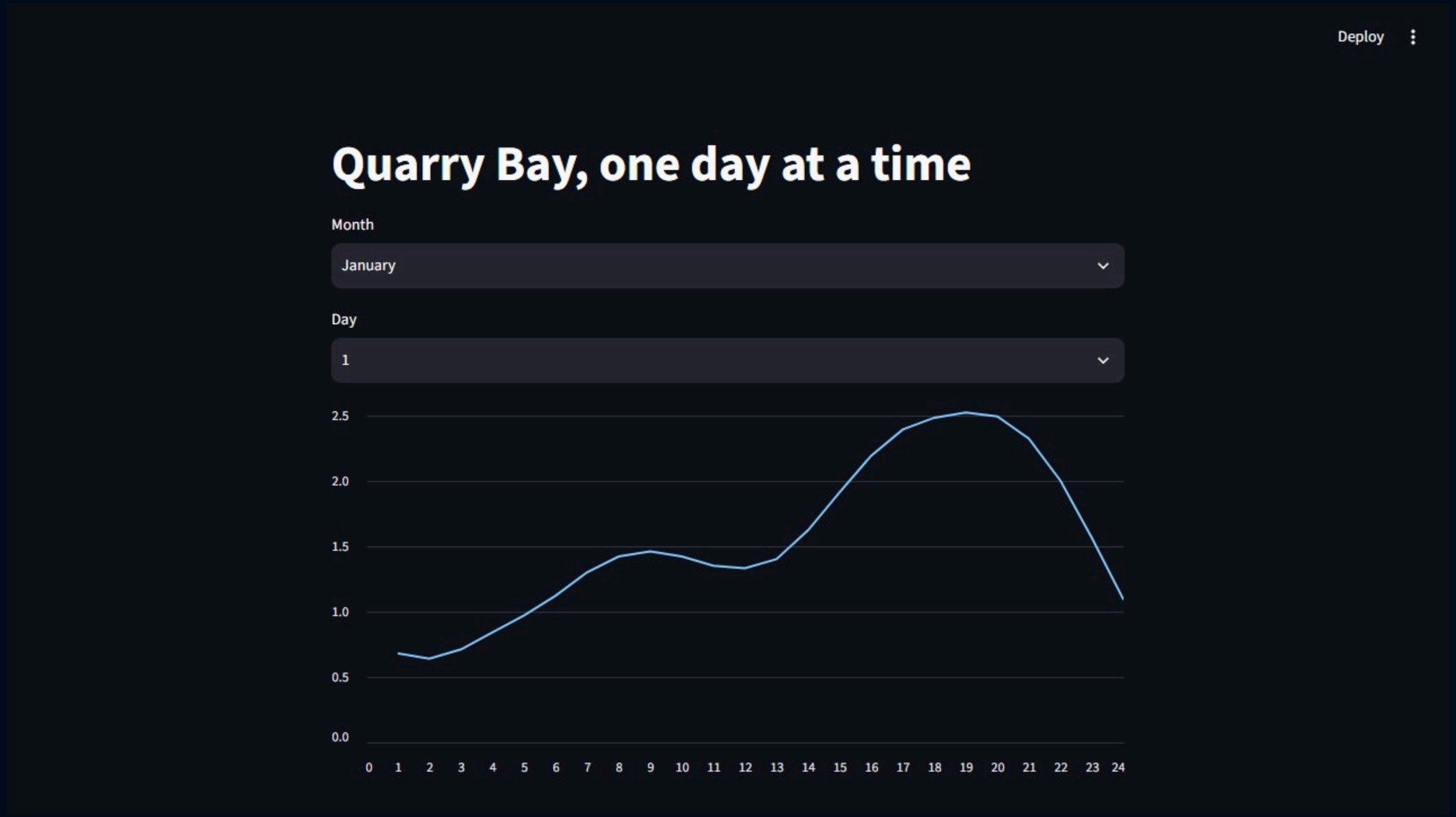
- **interface** — the part of a program a person can see and use.
- **input** — information or an action a person gives the program.
- **state** — what the program remembers between actions.
- **response** — the visible result of an action.

More words appear when they have a job: sd5913.github.io/teaching/glossary.html

01

The surface

SEE THE COMPLETE INTERACTION BEFORE OPENING THE CODE



01 · THE WORKING APP

Choose a day. See its 24 hourly tide heights.

One control, one clear response.

“When I choose a day, the chart shows that day’s 24 hourly tide heights.”

After someone chooses day 17, what is the state?

- **A** The mouse click
- **B** 17
- **C** The 24 plotted heights
- **D** The JSON file

02

One action through the app

THE SELECTOR SUPPLIES A VALUE; THE CHART USES ONE RECORD

Streamlit: a surface in Python

An **open-source framework** for turning Python scripts into interactive data apps.

Add controls, use their values, and show charts or tables.

Today: a month selector, a day selector, and a tide chart.

PROJECT **streamlit.io**
What it makes, with examples to explore.

SOURCE **github.com/streamlit/streamlit**
The code, issues, and release history.

Basic concepts
Understand widgets and script reruns.

API reference
Find selectors, charts, and worked examples.

Three parts of the interaction

INPUT

A person chooses

The day selector supplies 17.

STATE

The app keeps the value

The current run sees day = 17.

RESPONSE

The surface changes

The chart draws that record.

The value reaches the chart

17 — the selector returns a day.

One record — select_day finds it.

24 heights — the chart draws them.

```
day = st.selectbox(
    "Day", [r["day"] for r in rows]
)
record = select_day(rows, day)
chart = pd.DataFrame(
    {"Height (m)":
record["heights"]},
    index=range(1, 25),
)
st.line_chart(chart)
```

A changed widget reruns the script

When someone changes a selector, Streamlit runs the script again from top to bottom.

The widget keeps its selected value. During the new run, `day` contains that value, `select_day` returns a different record, and the chart is drawn again.

This rerun model belongs to Streamlit. Other interfaces handle events differently.

03

Waiting and events

BLOCKING INPUT, A REPEATED CHECK, OR A RESPONSE FUNCTION

One line waits

A command-line prompt stops at `input()` until a person answers.

That works for a short conversation. It does not work for a screen that must keep drawing or respond to other actions while it waits.

```
name = input("Name? ")
print("Hello", name)
```

Check, update, draw. Repeat.

A loop checks input during each frame.

At **60 fps**, the whole cycle has about **16.7 ms**.

A slow update delays the next check and the next frame.

```
while running:
    events = check_for_events()
    update_state(events)
    draw_frame()
```

An event becomes a visible change

01 · EVENT

A person clicks

The browser queues the event.

02 · CALLBACK

Your function runs

Read the selected day. Update the chart data.

03 · RENDER

The page updates

The browser gets a chance to draw.

Ready for the next event

Connect an event to a function

Event: the click.

Callback: update_chart.

The toolkit calls your function when the event arrives.

```
def update_chart(event):
    day = day_picker.value
    record = select_day(rows, day)
    chart.show(record)

button.on_click(update_chart)
```

Match the response to the program

COMMAND LINE

Wait

Ask once, then continue.

CONTINUOUS PICTURE

Poll

Check input during every frame.

DISCRETE ACTION

Callback

Run a function when an event arrives.

The semester loop

In week 1, this was a picture of a semester: keep working while the semester is on, then respond to the current problem and state.

Now the loop has a technical meaning. It checks a condition, updates values and decides what happens next.

Week 1 source

```
while semester() == 1:
    problem = current_problem()
    try:
        problem.solve()
    except:
        simplify(problem)
```

Which pattern best fits a browser button?

- **A** Blocking input
- **B** A callback
- **C** A 60 fps loop
- **D** A file refresh

04

Frontend and backend

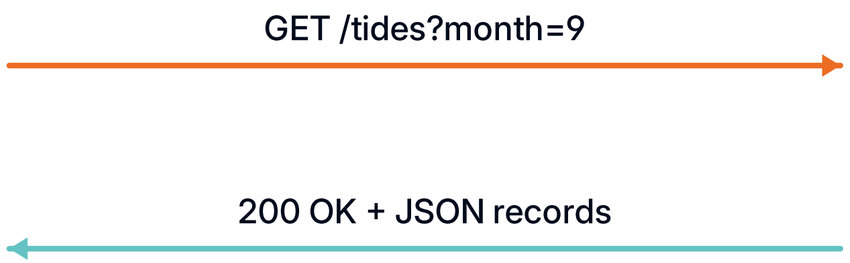
TWO EXAMPLES, TWO DATA PATHS

Two places, one conversation

FRONTEND · IN THE BROWSER

Choose and draw

The person chooses a month.
JavaScript asks for records.
The chart uses the reply.



BACKEND · ON THE SERVER

Read and return

FastAPI checks the request.
Python selects the records.
The saved JSON supplies data.

GitHub Pages delivers the frontend files.

Cloudflare runs this Python backend.

Streamlit connects the two sides for you

BROWSER

Choose day 17

A widget sends a value.
The page shows the chart.

widget value →

← page update

PYTHON PROCESS · YOUR LAPTOP OR A SERVER

Streamlit reruns app.py

Read the local JSON file.
Select the day's 24 heights.
Build the updated chart.

There is still a browser and a server. You write one Python app; no separate /tides API.

Ask, read, then draw

Ask for September records.

Read the JSON reply.

Draw one day in the browser.

```
const response = await fetch(
  API + "/tides?month=9"
);
const rows = await response.json();
const record = rows[0];

drawChart(record.heights);
```

FastAPI: a service in Python

An **open-source framework** for building HTTP APIs in Python.

Connect a URL to a function, validate inputs, and return structured data.

Today: ask for a month and receive its tide records as JSON.

PROJECT **fastapi.tiangolo.com**
The framework and its documentation.

SOURCE **github.com/fastapi/fastapi**
The code, issues, and release history.

First steps
Create a route and explore the /docs page.

Query parameters and validation
Read values from a URL and check them.

A Python function, available at a URL

GET /hello calls hello().

The function returns a dictionary. FastAPI sends it as JSON.

It also builds an interactive **/docs** page. Try our tide API.

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello")
def hello():
    return {"message": "Hello"}
```

Return the matching records

GET /tides?month=9

FastAPI reads month as an integer from 1 to 12.

Our function selects the records. FastAPI sends them as JSON.

```python
@app.get("/tides")
def tides(
    month: int = Query(ge=1, le=12)
):
    return select_month(
        load_rows(), month
    )
```

Three things to check

ADDRESS

The right endpoint

The frontend asks the Worker URL for /tides?month=9.

BROWSER PERMISSION

The allowed origin

CORS lets the teaching site read the API response.

RESPONSE

The expected shape

Each record has a month, a day and 24 heights.

Ask the deployed API for September

Run the request. Find **how many days**, **which first date**, and **four heights** in the reply.

YOUR TURN

```
import json
from pyodide.http import open_url

url = (
    "https://sd5913-week04-tides.venetanji.workers.dev"
    "/tides?month=9"
)
response = open_url(url)
rows = json.loads(response.read())

first = rows[0]
print(len(rows), "daily records")
print("first:", first["month"], first["day"])
print(first["heights"][:4])
```

This image came back from an API

A prompt went out. Image data came back.

Prompt: “Editorial paper sculpture of a tidal wave becoming a flowing ribbon…”

Qwen Image 2.1 · ComfyUI

Generated illustration · not measured tide data

A prompt goes in. An image comes back.

CLIENT · OUR PYTHON SCRIPT

Describe an image

Send a prompt + model.
Keep the key on the server.

POST /v1/images/generations

JSON containing image data

SERVICE · IMAGE API

Generate an image

Qwen Image 2.1
ComfyUI runs the workflow.

Decode the reply → save a PNG → place it in these slides.

Describe the job in JSON

POST sends a job to the service.

model chooses the generator. **prompt** describes the image.

The script reads the **API key** from its environment.

```
{
  "model": "qwen-image-2.1",
  "prompt": "Editorial paper ...",
  "size": "1024x1024",
  "n": 1,
  "response_format": "b64_json"
}
```

In the tide app, what does the service send back?

A The finished chart

B JSON records

C The mouse click

D The PowerPoint

05

Test the API

ONE FILE. ONE FEATURE. RED, THEN GREEN.

One file checks the API

```
def test_september_tides():
    with urlopen(API) as response:
        assert response.status == 200
        rows = load(response)
    first = rows[0]
    assert (first["month"], first["day"]) == (9, 1)
    assert len(first["heights"]) == 24
    assert isinstance(first["heights"][0], float)
```

week04/tdd/test_api.py · run first: watch it fail.

Return the data in that format

```python
@app.get("/tides")
def tides(month: int):
    rows = load_rows()
    return [row for row in rows if row["month"] == month]
```

Source: week04/api.py · rerun test_api.py.

One promise, four steps

01

Write the test

Say what response would convince you.

02

See it fail

Read the failure: does it point to the missing feature?

03

Add the route

Implement the smallest change that meets the promise.

04

Run it again

The same test now passes.

06

Workshop

TRY THE DEMOS, ADAPT AN IDEA, THEN WORK ON YOUR ASSIGNMENT

The tutorial is one continuous path

Use the first hour to try all the demo code: Streamlit with the local file, the browser/API request, the event-loop examples and the red-to-green API test.

Then adapt one idea to your own project. Use the final half-hour to work on Assignment 2.

github.com/sd5913/pfad/tree/2026/week04

Two hours

0:00–1:00

Try every demo

Run the local-file Streamlit app, browser/API request, event-loop examples and red-to-green test. Change a value and inspect what happens.

1:00–1:30

Adapt what you learned

Bring one idea into your own project: add a useful control, respond to an event, show data or write a small check.

1:30–2:00

Work on Assignment 2

Last chance to ask tutors about your submission. Use the rest for your question, data, plot, README and PROCESS.md.

06 · ASSIGNMENT 2

Keep the data picture moving

Assignment 2 is due **Sunday 4 October, 23:59**.

This interface can help you explore your data. The submitted work still needs its own question, source, picture, README and PROCESS.md.

Every control should help someone see or ask something.

Read the check in three groups

EXPLAIN

README + process

150+ words, the picture shown, and a meaningful PROCESS.md.

REPRODUCE

Code + data

Python parses, dependencies declared, raw data and picture committed.

SHOW PROGRESS

Commit history

At least three commits across two or more days.

The same check, on your laptop

```
uv run https://raw.githubusercontent.com/sd5913/pfad/2026/assignments/check.py --
assignment 2
```

Run from your assignment repo. Read the result, fix one item, then push again.

One control, one clear response

No class on 1 October · Assignment 2 due 4 October.

SD5913.GITHUB.IO/TEACHING