Problem Set 1

Selina Yu

2025-09-10

Problem 1

Memory and disk storage serve different purposes in a computer system. Memory is fast but limited and temporary, holding the data and instructions programs need while running; its contents are lost when power is turned off. Disk storage, by contrast, is much slower but provides far greater capacity and permanent persistence for files and applications. In data analysis, disk space tends to run out when saving or generating very large datasets, intermediate files, or model outputs, while memory is exhausted when attempting to load or process data that exceeds the available RAM, such as trying to read a dataset much larger than the computer's memory.

Problem 2: File Sizes and Formats

2a.

```python
import numpy as np

# Each float64 = 8 bytes
# For 20 columns, each row = 20 * 8 = 160 bytes
# To get ~16 MB: rows   16e6 / 160   100,000
rows = 100000
x = np.random.randn(rows, 20)
x.shape
```

```
(100000, 20)
```

Explanation: We calculated the number of rows needed by dividing 16 MB   16 million bytes by the size of each row. This gives around 100,000 rows.

2b.

```python
import pandas as pd
import os

# Round numbers to 12 decimal places
x = x.round(decimals=12)

# Save to CSV
pd.DataFrame(x).to_csv("x.csv", header=False, index=False)
print("CSV size (MB):", os.path.getsize("x.csv")/1e6)

# Save to Pickle
pd.DataFrame(x).to_pickle("x.pkl", compression=None)
print("Pickle size (MB):", os.path.getsize("x.pkl")/1e6)
```

```
CSV size (MB): 30.776975
Pickle size (MB): 16.000573
```

Explanation: CSV is a text format so every number is written as characters. Each value takes up around fifteen to twenty characters once you count digits, decimal points, signs, commas, and line breaks. Pickle is different because it stores binary floats directly, with each one taking eight bytes. That is why the CSV file ends up roughly twice the size of the pickle file.

2c.

If we round to 4 decimals, each number becomes shorter (7–8 characters). This reduces the CSV size, but it is still larger than pickle because CSV has to store digits, commas, and line breaks as text. Pickle remains fixed at ~16 MB since it always stores binary values. So CSV will not become smaller than pickle.

2d.

Changing to "one number per row" removes commas but adds newlines. Since both commas and newline characters take one byte, the total size stays roughly unchanged

2e.

```
# Compare read_csv vs read_pickle
%time df_csv = pd.read_csv("x.csv", header=None)
%time df_pkl = pd.read_pickle("x.pkl")
```

```
CPU times: user 217 ms, sys: 19.2 ms, total: 236 ms
Wall time: 248 ms
CPU times: user 349  s, sys: 2.66 ms, total: 3.01 ms
Wall time: 2.98 ms
```

Explanation: Pickle loads data faster since it works with binary numbers right away. CSV needs to parse text into numbers first, which slows it down. Sometimes the very first read can feel slower because of the operating system cache, but later reads are quicker.

2f.

```
# Read first 10,000 rows
%time chunk1 = pd.read_csv("x.csv", nrows=10000, header=None)

# Read 10,000 rows from the middle
%time chunk_mid = pd.read_csv("x.csv", skiprows=50000, nrows=10000, header=None)
```

```
CPU times: user 20.9 ms, sys: 2.09 ms, total: 23 ms
Wall time: 22.9 ms
CPU times: user 55.4 ms, sys: 3.79 ms, total: 59.2 ms
Wall time: 59.8 ms
```

Explanation: Reading with skiprows does not save time because pandas still needs to process earlier rows before it can move forward. It cannot jump straight to the middle of the file. This means that reading a block from the middle is not faster than reading all the rows up to that point.

Problem 3

In writing the code for Problem 4, I tried to follow the good practices recommended in Unit 4. I added docstrings to each function to explain the inputs, outputs, and purpose. I broke the work into small functions that each do one clear task, which makes the code easier to understand and reuse. I used descriptive names for variables and functions, with snake_case style. I kept the formatting clean with consistent spaces and indentation. I also ran ruff to check the style and fixed small issues. For the API requests, I added error handling so the program will not fail silently. The only thing I did not do was write unit tests, since that will come later. I also think the line length limit is a bit strict, and sometimes I use slightly longer lines if it improves clarity.

Problem 4

4a.

```python
import inspect
import ps1_Q4_Module as p4

print(inspect.getsource(p4.get_commits))

df_commits = p4.get_commits("numpy", "numpy")
df_commits.head()
```

```
def get_commits(owner: str, repo: str) -> pd.DataFrame:
    """
    Get up to 100 recent commits from a GitHub repo.

    owner(str): the account that owns the repo, for example "numpy"
    repo(str): the name of the repo, for example "numpy"

    Returns a DataFrame with commit details.
    """
    url = f"https://api.github.com/repos/{owner}/{repo}/commits"
    params = {"per_page": 100}
    response = requests.get(url, params=params)
    response.raise_for_status()
    data = response.json()

    # Extract relevant fields
    commits = []
    for c in data:
        commits.append({
            "sha": c.get("sha"),
            "author_name": c.get("commit", {}).get("author", {}).get("name"),
            "author_email": c.get("commit", {}).get("author", {}).get("email"),
            "date": c.get("commit", {}).get("author", {}).get("date"),
            "message": c.get("commit", {}).get("message"),
            "committer_login": (c.get("committer") or {}).get("login"),
        })

    return pd.DataFrame(commits)
```

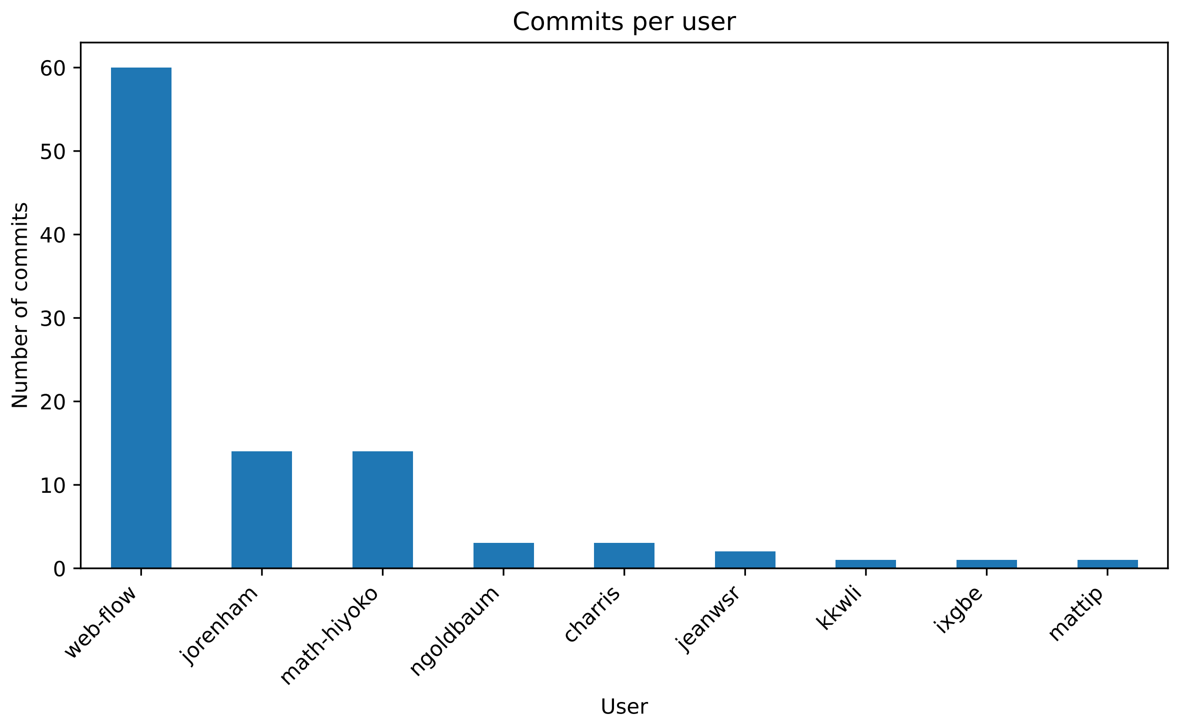
	sha	author_name	author_email
0	83402f7a43363e8041b8f297fdd4f76a7fa9fea1	Sandeep Gupta	sandeep.gupta12@ibm.com
1	9b9fe6fc775541135ac287fbba6f43076e4b84b7	Charles Harris	charlesr.harris@gmail.com
2	4d7e310dc796e5529fba033a6a18918fdf191b6e	Kelvin Li	kli@ca.ibm.com
3	c9cae4d6d5f7d550146cbd850d776f830cb9d088	dependabot[bot]	49699333+dependabot[bot]@users.norep
4	f579bee88c73db688f6c1b6452d7e5abb8a2ccc9	Charles Harris	charlesr.harris@gmail.com

Explanation: The get_commits function talks to the GitHub API and asks for up to one hundred commits from a repository. It picks out fields like the SHA, author name and email, commit date, commit message, and committer login. These details are then returned in a data frame.

4b

```
print(inspect.getsource(p4.plot_commits_hist))

# plot the histogram
p4.plot_commits_hist(df_commits)
```

```
def plot_commits_hist(df: pd.DataFrame) -> None:
    """
    Make a bar chart showing how many commits each user made.

    df: a DataFrame from get_commits

    Shows the plot directly, does not return anything.
    """
    counts = df["committer_login"].value_counts()

    plt.figure(figsize=(8, 5))
    counts.plot(kind="bar")
    plt.title("Commits per user")
    plt.xlabel("User")
    plt.ylabel("Number of commits")
    plt.xticks(rotation=45, ha="right")
    plt.tight_layout()
    plt.show()
```



Explanation: The histogram shows how many commits each user has made. It gives a quick picture of which contributors are the most active in the repository.

4c

```
print(inspect.getsource(p4.get_top_committer))

top_user = p4.get_top_committer("numpy", "numpy", verbose=True)
top_user
```

```
def get_top_committer(owner: str, repo: str, verbose: bool = True) -> dict:
    """
    Find the user with the most commits and get their GitHub info.

    owner(str): the account that owns the repo
    repo(str): the name of the repo
    verbose(bool, default=True): if True, also print a short summary

    Returns a dictionary with the user’s profile info.
    """
    df = get_commits(owner, repo)
    top_user = df["committer_login"].value_counts().idxmax()

    # Get user info
    url = f"https://api.github.com/users/{top_user}"
    response = requests.get(url)
    response.raise_for_status()
    user_info = response.json()

    if verbose:
        print(f"Top committer: {top_user}")
        print(f"Name: {user_info.get('name')}")
        print(f"Public repos: {user_info.get('public_repos')}")
        print(f"Followers: {user_info.get('followers')}")

    return user_info
```

```
Top committer: web-flow
Name: GitHub Web Flow
Public repos: 0
Followers: 904
```

```
{'login': 'web-flow',
 'id': 19864447,
 'node_id': 'MDQ6VXNlcjE5ODY0NDQ3',
 'avatar_url': 'https://avatars.githubusercontent.com/u/19864447?v=4',
 'gravatar_id': '',
 'url': 'https://api.github.com/users/web-flow',
 'html_url': 'https://github.com/web-flow',
 'followers_url': 'https://api.github.com/users/web-flow/followers',
 'following_url': 'https://api.github.com/users/web-flow/following{/other_user}',
 'gists_url': 'https://api.github.com/users/web-flow/gists{/gist_id}',
 'starred_url': 'https://api.github.com/users/web-flow/starred{/owner}{/repo}',
 'subscriptions_url': 'https://api.github.com/users/web-flow/subscriptions',
 'organizations_url': 'https://api.github.com/users/web-flow/orgs',
 'repos_url': 'https://api.github.com/users/web-flow/repos',
 'events_url': 'https://api.github.com/users/web-flow/events{/privacy}',
 'received_events_url': 'https://api.github.com/users/web-flow/received_events',
 'type': 'User',
 'user_view_type': 'public',
 'site_admin': False,
 'name': 'GitHub Web Flow',
 'company': None,
 'blog': '',
 'location': None,
 'email': None,
 'hireable': None,
 'bio': 'This account is the Git committer for all web commits (merge/revert/edit/etc...) mad
 'twitter_username': None,
 'public_repos': 0,
 'public_gists': 0,
 'followers': 904,
 'following': 0,
 'created_at': '2016-06-10T22:16:20Z',
 'updated_at': '2020-11-06T19:35:33Z'}
```

Explanation: The get_top_committer function finds the user with the largest number of commits. It then pulls that person's GitHub profile, such as their display name, the number of public repositories, and how many followers they have. If verbose is set to true it prints a short message, and if it is set to false it only returns the data object.

4d

All the GitHub API–related code is organized inside the ps1_Q4_Module.py file. In this .qmd file, we only display the functions using inspect.getsource() and show how to call them. This ensures the code is modular, reusable, and follows good practices.

4e

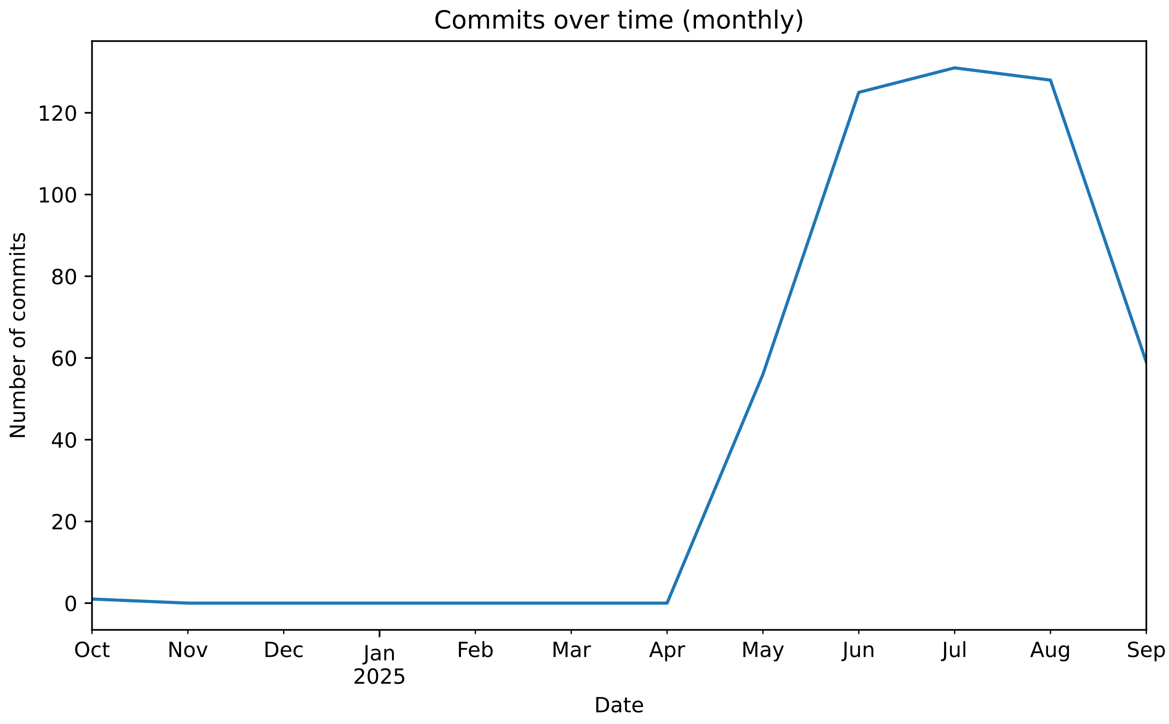
A environment.yml file is included in the repository. It specifies the packages used in this assignment: - python=3.12 - jupyter - numpy - pandas - matplotlib

4f

```
print(inspect.getsource(p4.get_all_commits))
df_all = p4.get_all_commits("numpy", "numpy", max_pages=5)
df_all.shape
# transfer the date as datetime and plot
df_all["date"] = pd.to_datetime(df_all["date"])
df_all.set_index("date", inplace=True)

# gets the monthly counts
monthly_counts = df_all.resample("M").size()

import matplotlib.pyplot as plt
plt.figure(figsize=(8,5))
monthly_counts.plot()
plt.title("Commits over time (monthly)")
plt.xlabel("Date")
plt.ylabel("Number of commits")
plt.tight_layout()
plt.show()
```

```
def get_all_commits(owner: str, repo: str, max_pages: int = 10) -> pd.DataFrame:
    """
    Get commits from many pages of a GitHub repo.

    owner(str): the account that owns the repo
    repo(str): the name of the repo
    max_pages(int, default=10): number of pages to fetch, each page has up to 100 commits

    Returns a DataFrame with all the commits collected.
    """
    all_commits = []
    for page in range(1, max_pages + 1):
        url = f"https://api.github.com/repos/{owner}/{repo}/commits"
        params = {"per_page": 100, "page": page}
        response = requests.get(url, params=params)
        if response.status_code != 200:
            break
        data = response.json()
        if not data:
            break
        # Save the basic details from each commit
        for c in data:
            all_commits.append({
                "sha": c.get("sha"),
                "author_name": c.get("commit", {}).get("author", {}).get("name"),
                "date": c.get("commit", {}).get("author", {}).get("date"),
                "committer_login": (c.get("committer") or {}).get("login"),
            })

    return pd.DataFrame(all_commits)
```

```
/var/folders/dz/jc8t_y152n157pf8n56463v40000gn/T/ipykernel_13019/351024062.py:9: FutureWarni
  monthly_counts = df_all.resample("M").size()
```



Explanation: For 4f, pagination lets us request many pages of commit data instead of just one. In this case we fetched up to five hundred commits. We then resampled the commit dates by month and drew a time series plot. The chart shows how active the project has been at different points in time and gives a sense of its development history.