STAT 1010 Lecture Notes

Yi Wang

2023-08-11

Table of contents

Preface

This is a book for STAT 1010: Introduction to Data Science at Auburn University at Montgomery. The book is written using Quarto.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

This is a book for STAT 1010: Introduction to Data Science offered at Auburn University at Montgomery.

This an ongoing project and updates are perpetually added.

2 Setting-up Python Computing Environment

2.1 Use Google Colab

All you need is a Google account. Sign in your Google account in a browser, and navigate to Google Colab. Google Colab supports both `Python` and `R`. `Python` is the default engine. Change the engine to `R` in `Connect->change runtime type`. Then you are all set. Your file will be saved to your Google Drive or you can choose to send it to your `GitHub` account (recommended).

2.1.1 How to run a project file from your Google Drive?

Many times, when you run a python file in Colab, it needs to access other files, such as data files in a subdirectory. In this case, it would be convenient to have the same file structure in the Google Colab user home directory. To do this, you can use Google Drive to store your project folder, and then mount the Google Drive in Colab.

Let's assume the project folder name, `pydata-book/`.Here are the steps:

1. `git clone` the project folder (example: `git clone https://github.com/wesm/pydata-book.git`) to your local folder. This step is only needed when you want to clone some remote repo from GitHub.
2. **Upload** the folder (ex: `pydata-book`) to Google Drive.
3. **Open the file using Colab**. In Google Drive, double click on the `ipynb` file, example, `ch06.ipynb` (or click on the three dots on the right end, and choose `open with`, then `Google Colaborotary`), the file will be opened by Google Colab.
4. **Mount the Google Drive**. In Google Colab, with the specific file (example, `ch06.ipynb`) being opened, move your cursor to the first code cell, and then click on the folder icon (this should be the fourth icon) on the upper left border in the Colab browser. This will open the file explorer pane. Typically you would see a folder named `sample_data` shown. On the top of the pane, click on the Google Drive icon to mount the Google Drive. Google Colab will insert the following code below the cursor in your opened `ipynb` file:

```
from google.colab import drive
drive.mount('/content/drive')
```

Run this code cell by pressing `SHIFT+ENTER`, and follow the prompts to complete the authentication. Wait for ~10 seconds, your Google Drive will be mounted in Colab, and it will be displayed as a folder named `drive` in the file explorer pane. You might need to click on the `Refresh` folder icon to see the folder `drive`.

5. Open a new code cell below the above code cell, and type the code

```
%cd /content/drive/MyDrive/pydata-book/
```

This is to change the directory to the project directory on the Google Drive. Run this code cell, and you are ready to run the file `ch06.ipynb` from the folder `pydata-book` on your personal Google Drive, just like it's on your local computer.

2.2 On your own computer

1. **Anaconda**: Download anaconda and install using default installation options
2. **VSC**: Download VSC and install
3. start VSC and install VSC extensions in VSC: Python, Jupyter, intellicode
4. (optional) **Quarto** for authoring: Download Quarto and install
5. Start an anaconda terminal. Navigate to the file directory.
6. Setup a conda **virtual environment**: stat1010 and install python and ipykernel engines

   `conda create -n stat1010 python ipykernel`
7. Activate the venv: `conda activate stat1010`
8. start VSC by typing `code .` in the anaconda terminal.
9. open/create a `.ipynb` or `.py` file.
10. Select the kernel `stat1010`
11. Run a code cell by pressing `Shift+Enter` or click the triangular play button.
12. Continue to run other cells.
13. After finishing using VSC, close the VSC, and deactivate the virtual environment in a conda terminal: `conda deactivate`

2.3 Best workflow with GitHub from Colab (or a local device)

Pre-req: Local repo is a clone of the GitHub repo with aligned HEAD

1. Keep sync with the upstream original owner repo. On GitHub, in the forked repo (under your account), Click on "Sinc fork".
2. Open (or create) a notebook from G-drive to work with in Colab.
3. Then, mount the G-drive. If on a local device, use the same workflow (open or create a notebook from the project directory).
4. In a termnal of Colab (or a terminal in VSC in a local device)

- `git pull` or `git pull --ff` or (safer method: `git pull --ff-only`)

If permission denied on G-drive, run this first then repeat git pull.

```
chmod +x .git/hooks/*
```

3. After editing, and before finish

```
git status
git add  files-to-commit
git commit -m "commit message"
git push  # this will push the files-to-commit to your fork/main
```

2.4 Team Github workflow

2.5 Initial setup

1. Fork and Clone

- **Fork**: You click "Fork" on GitHub → it creates a **copy** of the repo under *your GitHub account*. Navigate to `https://github.com/ywanglab/STAT4160`, then click on "Fork".
- **Clone**: You download a local copy of *your fork* to your computer. (only do this for the first time)

So after forking, you typically do (only for the first time)

```
git clone https://github.com/YOUR-USERNAME/REPO-NAME.git #REPO-NAME should be STAT4160
cd REPO-NAME   # the REPO-NAME should be STAT4160, cd to the current working directory
```

2. Add the original repo as “upstream”

Your fork is linked to your GitHub account (the “origin”). To stay in sync with the original project, add a remote for the source repository:

```
git remote add upstream https://github.com/ywanglab/STAT4160
```

Check remotes:

```
git remote -v
# origin    https://github.com/YOUR-USERNAME/REPO-NAME.git (push/pull)
# upstream  https://github.com/ORIGINAL-OWNER/REPO-NAME.git (pull only)
```

3. Create a feature branch in your fork

Never work directly on `main`. Instead create a new branch:

```
git checkout -b feature/my-contribution  # or homework/your_initial
# edit files...

# after you done your edit, push changes to origin/main

git add files-to-commit  # git add filename (or directoryname) use `.` rarely as it will add
git commit -m "Fix bug in utils"
git push origin feature/my-contribution  #git push by default push changes to origin/main
```

4. Open a Pull Request (PR) (only do this for the contribution you want to make, such as homework)
5. Go to your fork on GitHub (`https://github.com/YOUR-USERNAME/REPO-NAME`).
6. GitHub will show a banner: “You recently pushed to `feature/my-contribution`. Do you want to open a Pull Request?”
7. Click it → select **base repository = (upstream) original owner repo**, **compare = your branch**.

Note: `head repository` → your fork (e.g. YOUR-USERNAME/REPO-NAME) and branch (feature/…) that contains your changes.

4. Write a good description and submit the PR.

Now the maintainers of the original repo will review it. If approved, they’ll merge it.

2.6 Keep your fork in sync

Before making new contributions, update your `fork/main` with the latest `main` from upstream:

Option A) Do it on GitHub: If GitHub shows something like “This branch is 1 commit behind”, “Sync Fork”.

Option B): do it via terminal:

```
git checkout main   # checkout main
git pull upstream main        # pull from the upstream original repo
git push origin main          # update your fork on GitHub
```

Then branch off `main` again for your next feature.

3 Setting-up R Studio Computing Environment

3.1 Setting up your own computing environment on a personal computer

This is the recommended way and the advantage is that it's easy to handle files.

- Go to the website <https://posit.co/download/rstudio-desktop/>.
- Follow the two steps:

    1. download and install R: Choose the appropriate operating system, and then choose "base" to "install R for the first time". You can simply accept all default options.
    2. download Rstudio Desktop and Install it.

After installation, start R-Studio, and you are ready to use it.

3.2 Use R-Studio Cloud (No setting-up needed)

Alternatively, one can save the hassle of setting up on a personal computer and use the R-Studio Cloud for **free**. Here are the steps:

- Go to the website https://login.rstudio.cloud.
- Either create a new account using an email address such as your AUM email or simply "Log in using Google" or click on other log-in alternative.

After log-in to your account, you are ready to use R Studio.

4 Use Git and GitHub

I assume you already have an account on https://github.com. If not, you need to create an account there.

4.1 Download Git

1. Go to the website https://git-scm.com/downloads, select an appropriate operating system, select "Click here to download"
2. Run the downloaded setup file with a name such as `Git-2.42.0.2-64-bit.exe`, and accept all default options.

4.2 Establish a connection between a local repo and a remote GitHub repo

4.2.1 Clone an existing repo on GitHub

This is an easier way to establish a connection between a local repo and a remote repo if the remote repo is created ahead. We will make a connection between a remote repo in your GitHub account and a local directory. If the remote repo is not under your account, then skip steps 1 and 2.

1. Sign in to your GitHub account, and create a GitHub repo (such as named `homework`) on `GitHub` (https://github.com), you can add a README.md file or just choose not to add a README.md file.
2. On your local computer, open a `Git Bash` terminal.
3. Skip this step if you simply want the cloned repo to be in the current directory. Otherwise, In the terminal, type `mkdir myfolder` (create a folder named `myfolder` within the current directory) and then `cd myfolder` (change to the directory `myfolder`). The directory name `myfolder` can be any name you want.

4. `git clone https://github.com/Your_Git_UserName/homework.git` (change the remote repo path to match your actual remote repo).

   > ℹ Note
   >
   > To specify a specific folder to clone to, add the name of the folder after the repository URL, like this: `git clone github-repo-URL mylocalfolder`

5. Now you have established a connection between your local directory `homework` and the remote repo `homework` on GitHub.

6. Create a new file in the current local directory `homewor` on your local computer, such as using your favorite editor to create a file named `myfirstlocalfile.txt` with any content in it. Or for the sake of demonstration, you can use the following Linux command to create this file containing the line `#My first local file`.

   ```
   echo "#My first local file" >> myfirstlocalfile.txt
   ```

7. In the terminal, `git add .` This will add all changes to the **staging area**. This lets Git start to track the changes to files in your local directory.

8. Now you are ready to **commit** the changes, which versions (takes a snapshot of) the current files in the directory. A commit is a checkpoint where you can go back to.

   ```
   git commit -m "my first commit from local"
   ```

9. Now you are ready to sync the local repo with the remote repo.

   ```
   git push
   ```

   The GitHub might ask you to sign in for the first time. Choose `Sign in with your brower` to sign in to complete the push.

4.2.2 Initializing a Git Directory Locally First

The previous approach initializes a local Git repo by cloning a remote repo. You can also initialize a local Git repo by using `git init`. Follow the following steps:

3. Sign in to your GitHub account.

4. Create a GitHub **empty** repo (such as named `homework`) on `GitHub` (https://github.com) but make sure it is empty (do not add Readme.md file)

5. Start a Git Bash Terminal window on your local computer (You could also use the Terminal Window in RStudio or VSC). Navigate to the project directory; if you haven't yet created a project directory such as `homework`, do

   `mkdir project_dir` Example: `mkdir homework`

   Use `cd project_directory_name` to enter your local project directory;

   Use `ls` to list all files and directories or use `ls -al` to include all hidden files and directories. In your local Git Terminal, (note at this moment your local project directory is empty)

   ```
   echo "# homework0" >> README.md  #create a file README.md
   git init
   git branch -M main #rename the branch name to main
   git add .  # may use git add --all
   git commit -m "first commit"
   git remote add origin https://github.com/ywanglab/homework.git #(change the remote repo
   git push -u origin main
   ```

   > **i Note**
   >
   > 1. the general command format: `git push [remote-name] [branch-name]`
   > 2. difference between `git add .` and `git add --all`:
   >
   >   `git add .`: stages changes in the current directory and its subdirectories but does not include file deletions
   >
   >   `git add --all`: stages changes in the entire working tree, including deletions and untracked files. It is a more aggressive option and can be useful when you want to ensure that every change, including file deletions, is included in the next commit.
   >
   >   `git add --all` is equivalent to `git add -A`

6. if your local project directory already 1) contains files and 2) had performed `init git` before, then

   ```
   git remote add origin https://github.com/ywanglab/homework.git` #(change the remote repo
   git branch -M main
   git push -u origin main
   ```

7. in the pop-out `GitHub Sign-in` window, click on `Sign in with your browser`.

8. Note an empty folder would not be pushed to the remote repo until it has a file (even an empty file) in it. In this case, you can create an empty file such as `.gitignore`

4.3 Some other common commands

1. check git status: `git status` and `git status --short` for a compact way.
2. `git commit -a -m "message"` will stage and commit every changed, already tracked file without using `git add changed_file`
3. `git add file_changed`

   # add `file_changed` to the staging environment, i.e., git repo to start track those changes.
4. use `git log` to check all commits. Use `git log --pretty=oneline` or just `git log --oneline` for shorter display.

   `git log origin/main` #check the remote repo `origin/main` commits
5. use `git diff origin/main` to show the differences between the local `main` and `origin/main`.
6. use `git checkout .` to revert back to the previous commit. Any changes after the previous commit will be abandoned.
7. to get to a previous commit, use `git checkout seven_character_commit_hash`. To get back to main, use `git checkout main`.
8. Git commit --amend

```
`commit --amend` is used to modify the most recent `commit`. It combines changes in the `stag

One of the simplest things you can do with `--amend` is to change a `commit` message with spe
```

9. Git Revert HEAD:

   `revert` is the command we use when we want to take a previous commit and add it as a new commit, keeping the log intact. Revert the latest `commit` using git `revert HEAD` (`revert` the latest change, and then `commit`), adding the option `--no-edit` to skip the commit message editor (getting the default `revert` message):

   `git revert HEAD --no-edit`

   > Note
   >
   > To revert to earlier commits, use `git revert HEAD~x` (x being a number. 1 going back one more, 2 going back two more, etc.)

10. Git Reset

    `reset` is the command used when we want to move the repository back to a previous `commit`, discarding any changes made after that `commit`. Let's try and do that with reset.

    ```
    git reset seven-char-commit-hash
    ```

11. Git Undo Reset

    Even though the commits are no longer showing up in the log, it is not removed from Git. If you know the commit hash you can reset to it:

    ```
    git reset seven-char-commit-hash
    ```

12. To permanently go back to a previous commit, use

    ```
    git reset --hard seven_char_commit_hash
    ```

13. to go back to a previous commit, but not changing the files in the working directory use the `--soft`‘ option.

    ```
    git reset --soft seven_char_commit_hash
    ```

14. `git remote -v` Get the reminder of the remote repo. To rename the remote origin: `git remote rename origin upsteam` rename remote repo `origin` to `upstream`

    > **Note**
    >
    > According to Git naming conventions, it is recommended to name your own repository `origin` which you have read and write access; and the one you forked for `upstream` (which you only have read-only access.)

15. if you want to remove the file only from the remote GitHub repository and not remove it from your local filesystem, use:

```
git rm -rf --cached file1.txt #This will only remote files; If intending to remove local file
git commit -m "remove file1.txt"
```

And then push changes to remote repo

```
git push origin main
```

14. For some operating system, such as Mac or Linux, you might be asked to tell GitHub who you are. When you are prompted, type the following two commands in your terminal window:

```
git config --global user.name "Your Name"
git config --global user.mail "your@email.com"
```

This will change the Git configuration in a way that anytime you use Git, it will know this information. Note that **you need to use the email account that you used to open your GitHub account**. `global` sets the username and e-mail for **every repo** on your computer. If you want to set the username/e-mail just for the current repo, remove `global`.

4.4 Use Git help

1. `git command -help` See all the available options for the specific command. Use '`--help` instead of `-help` to open the relevant Git manual page.
2. `git help --all` See all possible commands

4.5 When the upstream repo changes

When Git tells you the upstream repo is ahead,

15. Do `git pull` or `git pull origin`

    This is equivalent to `git fetch origin`, and then `git merge origin/main.`Then you can commit and push a new version to the remote repo.

16. `git pull` will not pull a new branches on the remote repo to local, but it will inform you if there is a new branch on the remote repo. In this case, just `git checkout the_remote_new_branch_name` will pull the remote branch to local. Note there is **no need** to create locally the branch by `git branch the_remote_new_branch_name`

4.6 Create branch

16. To add a branch to the main branch `git branch branchname`

    Switch the branch `git checkout branchname`

    To combine the above two actions, `git checkout -b branchname`, create a new branch named `branchname` if it does not exist and move to it.

Adding a file in branch `echo "#content" >> filename.txt`

Then `add` the file and commit the file. To push the branch to the remote repo we **have to use**

`git push --set-upstream origin branchname` The option `--set-upstream` can be replaced by `-u`

to see all branches in both local and remote: `git branch -a` Or `git branch -r` for remote only.

4.7 Merge branch to main branch

1. Switch from a branch (with name such as `branchname` to the `main` using
   `git checkout main`
2. on the `main` branch, Merge command to merge the branches
   `git merge branchname`

To delete a branch:

```
git branch -d branchname
```

4.8 Handle large files (>= 150Mb) on GitHub

GitHub does not allow to upload a file of size greater than 150Mb. However, one can use `git lfs` to handle large files exceeding this size up to several Giga bytes. The first thing is to install `git lfs`. Head to https://git-lfs.com, once dowlonad and install the Git command line extension, set up Git LFS for your user account by running

```
git lfs install  #(only need to do this the first time)
```

Then In each Git repository where you want to use Git LFS, select the file types you'd like Git LFS to manage (or directly edit your .gitattributes). You can configure additional file extensions at any time.

```
git lfs track "path/to/file"
```

Then do the regular `git add .` and `git -m "message"` and `git push`. Note one must use `git lfs track` a file first before doing `git add` and `git commit`.

ℹ Note

Note you need to track the large-size file first before you add it to the staging area. But often you will find this error after you try to push your changes to the GitHub. In this case, you will have to remove the commit history of this file first. One way to do this is to `reset -soft` the HEAD to the previous working HEAD, and then do `git lfs track` followed by `git add` and `git commit`, `git push`. Specifically,

```
git reset --soft HEAD ~1  # or the_7-char_commit_hash
git lfs track "path/to/large_file"
git add .
git commit -m "commit message"
git push
```

Note the `--soft` option allows the changes in the working directory not affected, otherwise any change after the previous commit will be removed.

4.9 Contribute by forking a GitHub repo and commit to the forked repo and create a pull request

1. after forking a (foreign) GitHub repo to your own GitHub account, `git clone` that repo under your account to your local repo.
2. make changes in your local directory.
3. Submitting your changes for review

    1. **Commit your changes locally.** Once you are ready to submit your changes, run these commands in your terminal:

    ```
    git add -A                            # Stages all changes, short for --all
    git commit -m '[your commit message]' # Makes a git commit
    ```

    2. **Make a pull request.** (A pull request is a proposal to change) A GitHub pull request allows the owner of the forked upstream repo to review and make comments on your changes you proposed. Once approved, the upstream owner can merge your changes. Run:

    ```
    git push origin # Push current branch to the same branch on GitHub
    ```

4. Then go to your remote forked repo in your account on the GitHub site and click **Contribute**,and then **Open pull request**, this will take you to the upstream repo. In the form, leave a message explaining the change, and **Create pull request**. **Do not** select `Close pull request` unless you want to cancel the pull request.

4.10 Project

1. First make sure you have forked the course repo `https://github.com/ywanglab/stat1010.git` to your own GitHub account.
2. Now go to your GitHub account, git clone the forked course repo

```
git clone https://github.com/your_git_user_name/stat1010.git
```

to your local computer

4. add your resume file in the folder `./resume`

`git add`, `commit` and `push` your changes to the upstream repo using

```
git add .

git commit -m "added YourFirstName's resume"

git push origin
```

5. Then go to your remote forked repo in your account on the GitHub site and click **Contribute**,and then **Open pull request**, this will take you to the upstream repo. In the form, leave a message explaining the change, and **Create pull request**. **Do not** select `Close pull request` unless you want to cancel the pull request.

4.11 More on git

`git pull` = `git fetch` + `git merge`

- **`git fetch`** → downloads commits from the remote into your local refs (e.g. `origin/main`).
- **`git merge`** → merges those new commits into your current branch.

4.12 2. What does `--ff` mean?

- `--ff` = **fast-forward if possible**.
- That means: if your branch has **no local commits** since it last matched the remote, Git will simply **move the branch pointer forward** to match the remote — no merge commit is created.

Example (before pull):

```
A---B---C  (origin/main)

A---B   (main)
```

If you run `git pull --ff` and your branch is strictly behind `origin/main`, Git just slides `main` forward:

```
A---B---C   (origin/main, main)
```

4.13 3. How is this different from the defaults?

- `git pull` without flags:
  - May create a **merge commit** if histories diverged.
- `git pull --ff`:
  - Does a fast-forward if possible.
  - If not possible (you made local commits), Git falls back to a **merge commit**.
- `git pull --ff-only`:
  - Does a fast-forward **only**.
  - If not possible, it **aborts** with an error (no merge commit allowed).

4.14 4. When to use it

- `--ff` is safe if you don't mind merge commits being created when necessary.
- `--ff-only` is stricter (no merge bubbles, linear history).
- Teams often configure one of these globally so `git pull` always behaves consistently.

4.15 1) when there is a diverge

Recall the setup:

```
A B C     (origin/main)
     \
      D     (main)
```

- `C` and `D` share a common ancestor `B`. So Git sees:
- One branch has new work (`C`).
- Another branch has different new work (`D`).
- **D** is a **local commit** you made on `main` after you last pulled.
- If you run `git pull --ff` (or `--ff-only`), **fast-forward is NOT possible** because your branch has extra work (D).
  - `--ff-only` → **aborts** with an error.
  - `--ff` → falls back to a **merge**, creating a merge commit (see next section).

4.15.1 `git pull` or `git pull --ff` (merge fallback)

- Git fetches `origin/main` at `C`
- Git merges `C` into your local `main` with `D`, producing **M**:

```
After pull (local):
A B C
     \ \
      D M     (main)
        ^
     merge commit
```

When you run `git merge origin/main` (or `git pull` with merge strategy):

- Git identifies the **common ancestor** of the two branches → here, commit **B**.
- Then it looks at:
  - The changes between `B` → `C` (remote's changes).
  - The changes between `B` → `D` (your changes).
- Git tries to combine both sets of changes into a new snapshot.

That new snapshot becomes a new commit **M**. The merge commit **M** exists only **locally** until you `git push`. * When you **push**, `origin/main` is updated to point to **M**, and the remote history now includes that merge commit.

- Pros: Preserves exact history as it happened (no rewrite).
- Cons: Adds **merge commits**; history can get "braided".

4.15.2 Option 2: `git pull --rebase` (replay your work on top of remote)

- Git rewrites your local commits onto the fetched remote tip:
  - Rewrites `D` into `D'` applied after `C`.

```
A  B  C  D'       (main)
          ^
      rebased (new) commit
```

- Pros: **Linear history**, no merge commit.
- Cons: **Rewrites** your local commits (new SHAs). If you had already pushed `D`, you'll need `git push --force-with-lease`.

4.16 How to set opitons gloabally

- **Team prefers linear history** → `git pull --rebase` (and set it as default)

  ```
  git config --global pull.rebase true
  git config --global rebase.autoStash true
  ```

- **Keep exact history / avoid rewrite** → `git pull --ff` (merge when needed)

  ```
  git config --global pull.rebase false
  ```

- **Never auto-merge; be explicit** → `git pull --ff-only`

  ```
  git config --global pull.ff only
  ```

5 1) Concrete example: what does "merge C with D to produce M" look like?

Assume the repo has one file, `README.md`.

5.0.1 Commits and changes

- **B (common ancestor)** `README.md`:

  ```
  Hello project
  ```

- **C (remote, on origin/main)** — someone else added a line:

  ```
  Hello project
  Remote line
  ```

- **D (your local commit, on main)** — you added a different line:

  ```
  Hello project
  Local line
  ```

So history diverged:

```
A B C        (origin/main)
     \
      D        (main)
```

5.0.2 You run: `git pull` (merge strategy) or `git merge origin/main`

Git computes the diff **B→C** ("add `Remote line`") and **B→D** ("add `Local line`"), applies both, and creates **merge commit M**:

- **M (merge result)**:

  ```
  Hello project
  Local line
  Remote line
  ```

(order may vary if both append—Git picks a consistent merge; if both edit *the same* line, you'll get a conflict to resolve.)

New history:

```
A B C
     \ \
      D M      (main)
```

M has **two parents**: D and C. That's a "merge commit".

5.1 Option B: `git pull --rebase` (replay your work on top of remote)

Git takes your local commit(s) and **replays** them atop `C`, producing a *new* commit `D'`:

```
A B C D'    (main)
```

- Pros: **Linear history** (no merge commits), cleaner `git log`, bisect/blame often simpler.
- Cons: **Rewrites** your local commit(s) (new SHAs).
  - If you had already pushed `D`, you must **force-push** the rewritten branch (see below).

5.1.1 Which should you use?

- Many teams prefer `--rebase` for a clean linear history (especially on feature branches).
- If your team forbids history rewrites on shared branches, use **merge** (`git pull` or `git pull --ff`), or make sure you only rebase commits that haven't been pushed/shared yet.

  Tip to make rebase the default:

  ```
  git config --global pull.rebase true
  git config --global rebase.autoStash true
  ```

6 3) What is `git push --force-with-lease` (and why it's safer than `--force`)?

When you **rebase** local commits that were already pushed, your local branch history no longer matches the remote's. A normal `git push` will be rejected. You need to overwrite the remote branch tip—i.e., a force push.

- **`git push --force`** overwrites the remote branch **unconditionally** (dangerous—you could clobber someone else's new commits if they pushed while you were rebasing).
- **`git push --force-with-lease`** is the **safe** version:
  - It says: "Force-push **only if** the remote branch still points to the commit I think it does."
  - If someone else has pushed new commits, **the push is rejected** instead of overwriting their work.

6.0.1 Typical rebase + push flow

```
# Update local view of remote
git fetch

# Rebase your local work onto the remote tip
git rebase origin/main   # resolve conflicts if any; git rebase --continue

# Safely update the remote branch
git push --force-with-lease
```

6.0.2 Example workflow with `git stash`

6.0.2.1 1. Check repo status

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    modified:   app.py
    modified:   utils.py
```

You've made some edits but don't want to commit yet.

6.0.2.2 2. Stash your changes

```
$ git stash push -m "WIP: refactor utils"
Saved working directory and index state WIP on main: 1a2b3c4 Add logging
```

Now the working directory is clean.

6.0.2.3 3. Verify with `status`

```
$ git status
On branch main
nothing to commit, working tree clean
```

The changes are hidden away.

6.0.2.4 4. List stashes

```
$ git stash list
stash@{0}: On main: WIP: refactor utils
```

Your stash is safely stored.

6.0.2.5 5. Switch branch, pull, or do other work

```
$ git switch feature-branch
Switched to branch 'feature-branch'
```

6.0.2.6 6. Apply the stash back

```
$ git stash apply stash@{0}
On branch feature-branch
Changes not staged for commit:
    modified:   app.py
    modified:   utils.py
```

Changes are back, but the stash still exists in the list.

6.0.2.7 7. Or use `pop` to apply *and remove*

```
$ git stash pop
On branch feature-branch
Changes not staged for commit:
    modified:   app.py
    modified:   utils.py
Dropped refs/stash@{0} (abc123def456...)
```

6.0.2.8 8. Confirm stash list is empty

```
$ git stash list
# (no output - list is empty)
```

*Summary of this session**

- You edited files.
- `git stash push` cleaned your working directory but saved changes.
- Later, `git stash apply` or `git stash pop` restored those changes.

6.1 Rebuild the index respecting `.gitignore`

If you have modified `.gitignore` and you already pushed some files that you did not want to push, to remove those files already pushed to Github, you need to remove them from the git index to untrach them.

```
git rm -r --cached .
git add .
git commit -m "Reindex: drop ignored files from repo"
git push origin <your-branch>
```

to remove specific folder or files:

```
git rm -r --cached .Rhistory .Rproj.user # `-r` is needed for a directory
```

6.2 Unstage and untrack

- **unstage** = remove from the staging area (index), but keep the file under Git's tracking.
- **untrack** = stop Git from tracking the file altogether.

6.2.1 To unstage (but keep tracking):

If you already ran `git add file.txt` and want to undo that:

```
git reset HEAD file.txt
```

Now `file.txt` is back in "modified" state but not staged. To unstage everything:

```
git reset HEAD
```

6.2.2 To untrack (stop Git tracking the file):

If a file is already committed to the repo but you want Git to forget it:

```
git rm --cached file.txt
```

- `--cached` removes it from the index (tracking) but leaves it in your working directory.
- Next commit will record the removal.

- If you want to untrack entire directories:

```
git rm -r --cached my_folder/
```

6.2.3 Prevent tracking in the future

Add the file or folder to `.gitignore` so Git won't pick it up again:

```
# .gitignore
file.txt
my_folder/
```

6.3 Team Git workflow

Avoid committing directly to `main`?

6.3.1 1. Clone the repo & stay off `main`

Everyone starts from:

```
git clone https://github.com/org/project.git
cd project
git checkout main
```

But nobody codes directly on `main`.

6.3.2 2. Create feature branches

When starting new work:

```
git checkout -b feature/login-page
# do edits, commits...
git push origin feature/login-page
```

- Each branch is **short-lived**, focused on a feature, fix, or experiment.
- Branch names like `feature/`, `bugfix/`, `hotfix/` make it clear what's happening.

6.3.3 3. Push branch to GitHub and open a Pull Request (PR)

On GitHub:

- Open a **PR from `feature/login-page` → `main`**.
- Teammates can review code, request changes, and run tests before merging.

6.3.4 4. Keep your branch up-to-date

If `main` has moved on while you're coding:

```
git checkout main
git pull origin main
git checkout feature/login-page
git merge main   # or git rebase main
```

This keeps your branch compatible with the latest `main`.

6.3.5 Summary

- **Don't commit directly to `main`.**
- Each collaborator works in their own **branch**, pushes it to GitHub, and opens a **Pull Request**.
- Merges into `main` only after **review** + **tests pass**.

This way, `main` always works, and the team avoids chaos.

7 My Jupyter Notebook

Yi Wang (boldfaced using ** **)

Educator AUM

The following line is italicized using * *

I am interest in data science because it is a discipline that I feel love with.

7.0.1 Perform addtion

```
# code block
1+1
```

```
2
```

7.0.2 Horizontal Rule

Three or more

first rule using ***

---

using dashes —

Using (underscores) _____

7.0.3 Bulet list

using *

- Bird
- Frog
- Cat
- Dog

7.0.4 Numbered list

using 1. item (there is a space between 1. and item)

1. Apple
2. Pear
3. Peach

7.0.5 Tables

left-aligned	centered	right-aligned
1/2/2020	Mary	Apple
1/3	Johnason	Tomato

7.0.6 Hyperlinks

Click here to access my github account.

7.0.7 Images

Figure 7.1: A computer monitor

7.0.8 Code/Syntax highlighting

```
s = "Python syntax highlighting"
print s
```

7.0.9 Blocked quotes

using >

> Blockquotes are very handy in email to emulate reply text.
>
> This line is part of the same quote.

7.0.10 Strikethrough

using ~~ before and after a phrase

~~strikethrough this~~

8 Homework Assignments

I will use some assignments from https://cognitiveclass.ai.

1. Browser `Course & Projects`. Search for `Python for Data Science`. `Enroll Now` the class, and `Go to the Course`, and `Start the Course`.

2. Complete the following assignments from `Modules 1-4` and Part of `Module 5.` Excluding the `API` section in `Module 5`.

Module	Contents	Suggested Deadlines
Module 1	Python Basics	10/09/2023
Module 2	Python Data Structures	10/09/2023
Module 3	Python Prrogramming Fundamentals	10/09/2023
Module 4	Working with Data in Python	10/16/2023
Module 5	Working with Numpy Arrays (Excluding Simple APIs)	10/16/2023
Final Exam	Optional	

   Complete all `Practice Questions`, `Review Questions` and `Labs`. After your completing all the assignments, click on `Progress`, print the page (in PDF or hard copy), and send it to me. The page should show your `username` on the top right corner.

3. Enroll in the course `Data Anlaysis with Python`.

   Complete the following assignments.

Module	Contents	Suggested Deadlines
Module 1	Introduction	10/23/2023
Module 2	Data Wrangling	10/30/2023
Module 3	Exploratory Data Analysis	11/06/2023

4. Enroll in the course `Data Visualization with Python`.

   Complete the following assignments.

Module	Contents	Suggested Deadlines
Module 1	Introduction to Visualization	11/13/2023
Module 2	Basic Visualization Tools	11/20/2023
Module 3	Specialized Visualization Tools	11/27/2023
Module 4	Advanced Visualizaiton Tools (Optional)	

References