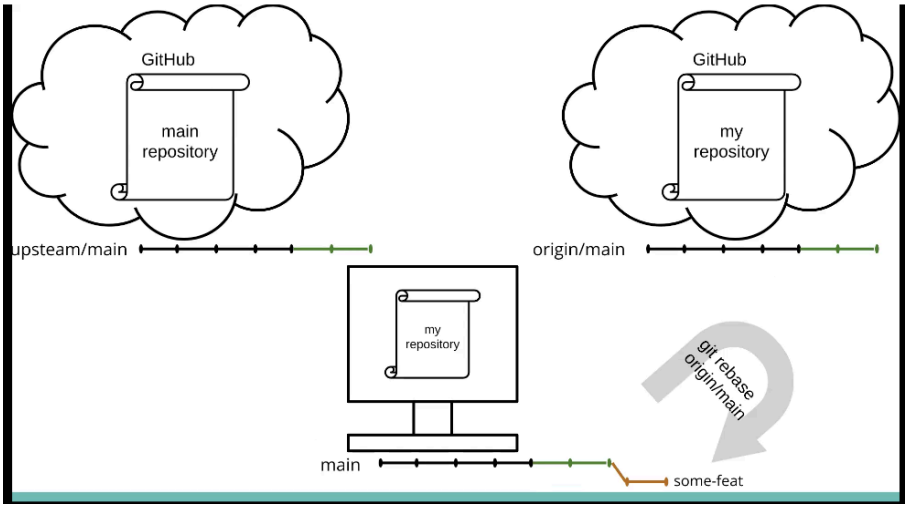
CAS CS 506
Lec 02

Git Review & Clean Code

1. Git Review
    a. git add myfile.txt
       git commit -m “first commit”
       git status
       git diff
       git add myfile.txt → after made changes on the file
       git commit -m “second commit”
    b. git log
       git checkout “commit id” → the file has went back to the first commit version/ might not have all the files
    c. git checkout -b → create a new branch that has not been defined
       ex)
       git checkout -b new-branch → new-branch is the new branch
       git branch → shows all the branches
    d. git add anotherfile.txt
       git commit -m “fourth commit” → happens on the “new-branch”
    e. git checkout master → go back to the “master” branch
       git checkout new-branch → go back to the “new-branch” branch
    f. Rebase
        i. To keep fork up to date,
           git pull upstream main
           git push origin main
        ii. Do not commit to your main branch but create a new feature/branch
        iii. 

2. The Problem
    a. Software Systems get replaced not when they wear out but when they crumble under their own weight because they have become too complex
3. Set Yourself Up for Success
    a. Stay organized - have a plan and prioritize
    b. Structure
        i. Do one thing
        ii. Compartmentalize
        iii. Many functions with small bodies > one function with a large body
        iv. Before writing code ask “How will someone use this (or part of this) code?”. Minimize side effects
    c. Method/Process
        i. Top Down Approach
            1. Start off and build pseudo code in detail with all the helper functions
            2. Anything that is too complicated, write down first and code later (declare but implement later)
            3. Be clear and pretend that I have access to all the functions
            4. Build some sort of a tree
            5. Drawback: with a program language like Python, it does not work well since we do not have runnable code until the end → cannot run the code more often
            6. Therefore, it is nice with type languages where we can run the type checker on the functions that we declare and at least verify that the inputs and outputs align with one another
        ii. Bottom Up Approach
            1. Requires much more planning (reverse the top down approach)
            2. Combine all the building blocks to make higher order functions and combine those to make higher order or complex functions
            3. Drawback: Requires a good planning but no one is good at planning (could be a bit of waste of time)
            4. Easier to debug
        iii. Solve the 90% problem first - then improve / refine to get to 100%
        iv. Boring code is good code: keep it simple
        v. Check soundness by reading your code before testing
4. But if You Must Debug
    a. Don’t Panic
    b. Read the error
        i. What is the error telling you?
        ii. Where did the error occur?

iii. Is this a cause or just a symptom (one bug can hide another)?

c. Re-read your code - take your time!
    i. Can you mentally trace through your code to reproduce the error in your head?
        1. If not, the code may need some refactoring because it's too complex

d. Sanity check where you can
    i. Is everything set up properly? Are the things that are supposed to communicate actually communicating?

e. Now Look Online for Some Help
    i. Hopefully with the above out of the way you have a good idea what to search for in order to actually fix the issue?

f. Take Break

5. The 8-queen puzzle
    a. DFS
    b. class Board:

```
def __init__(self):
    self.board = [["-" for _in range(8)] for _ in range(8)]
def __repr__(self):
    res = ""
    for row in range(8):
        for col in range(8):
            res += self.board[row][col]
            res += " "
        res += "\n"
    return res
def set_queen_at(self, row, col):
    self.board[row][col] = "Q"
def unset_queen_on(self, row):
    self.board[row] = [ "-" for _ in range(8)]
def is_valid_row(self, row, col):
    for j in range(8):
        if j != col and self.board[row][j] == "Q":
            return False
    return True
def is_valid_col(self, row, col):
    for i in range(8):
        for i != row and self.board[i][col] == "Q":
            return False
    return True
```

```
def is_valid_move(self, row, col):
    if not self.is_valid_row(self, row, col):
        return False
    if not self.is_valid_col(self, row, col):
        return False
    if not self.is_valid_diag(self, row, col):
        return False
    return True
def get_queen_on_row(self, row):
    for i in range(8):
        if self.board[row][i] == "Q":
            return i
    raise ValueError("no queen on row")
    # we should never hit this case
def find_solution(self):
    row = 0
    col = 0
    while row < 8:
        # we are searching for a solution
        if self.is_valid_move(row, col):
            self.set_queen_at(row, col)
            row += 1
            col = 0
        else:
            col += 1
            if col >= 8:
                # we weren't able to place a queen on this
                row
                # we need to backtrack and adjust the
                position of the queen on the previous row
                col = self.get_queen_on_row(row - 1)
                col += 1
                row -= 1
    # we have found a solution
    print("Found a solution: ")
    print(self)
```

c.
```
test = Board()
test.set_queen_at(1, 1)
print(test)
```

```
test.unset_queen_on(1)
print(test)
```

6. Coding steps
    a. Create a board first
    b. Get and Set Queens on the board
    c. Is a location valid
    d. Search for one solution first
    e. Find all solutions
    f. Refine