Managing Large Code Projects

15-110 – Friday 10/31

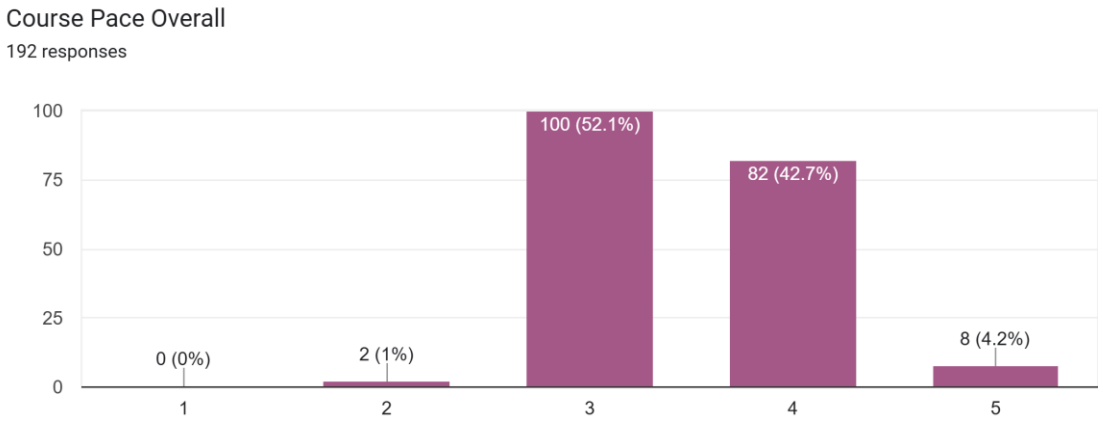
Announcements

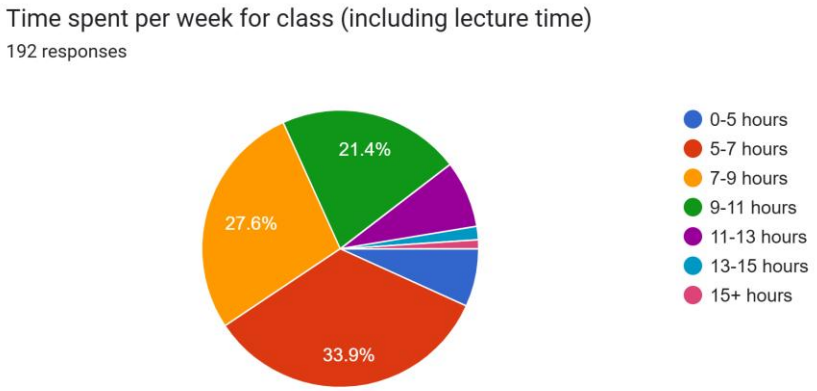
- **Check5** due on Monday
  - Can do almost all of Hw5's written component after today's lecture too!

Midsemester Feedback Report

- Thanks to the 192 students who filled out the midsemester survey!

- Major takeaways:
  - Average pace: **3.5/5**. In line with most semesters.
  - Average hours per week: **8**. Expected hours/week is 10, so this is great!

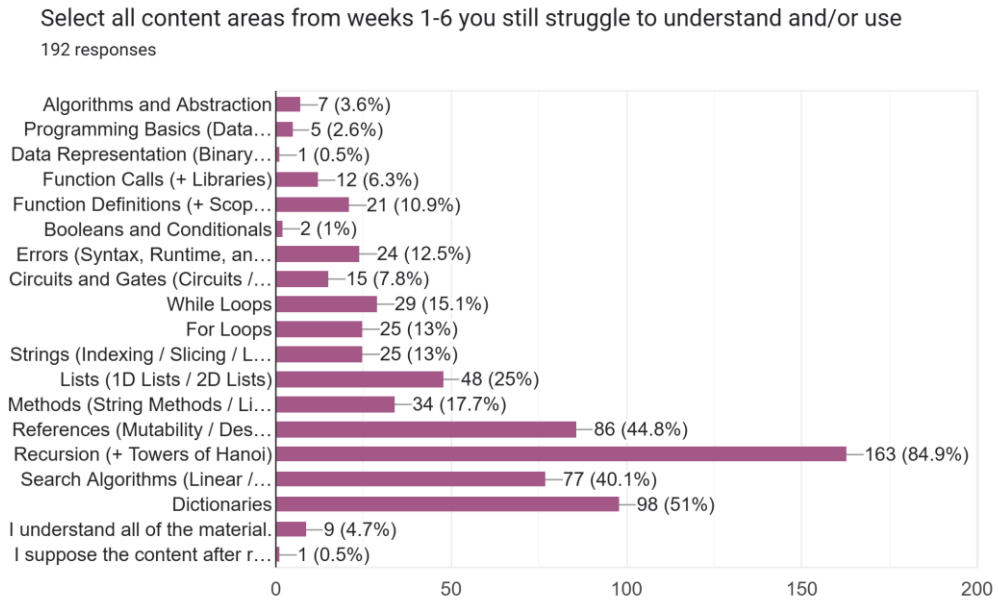


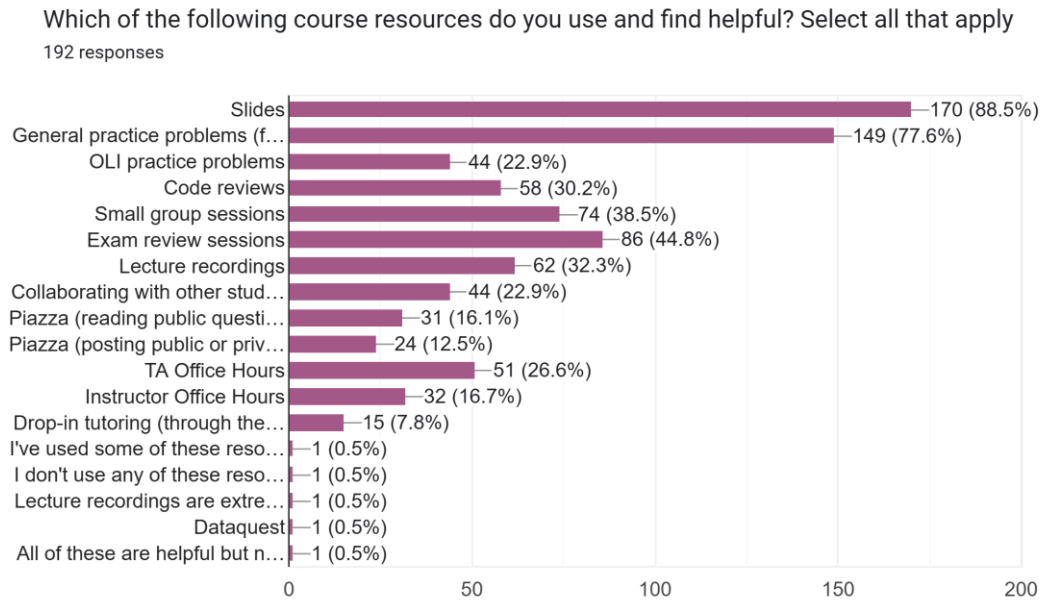


Midsemester Feedback Report

- Hardest topics: Recursion (84.9%), Dictionaries (51%)

- Most useful course resources: Slides (89%), Practice problems (78%), Exam review sessions (45%), Small group sessions (39%)

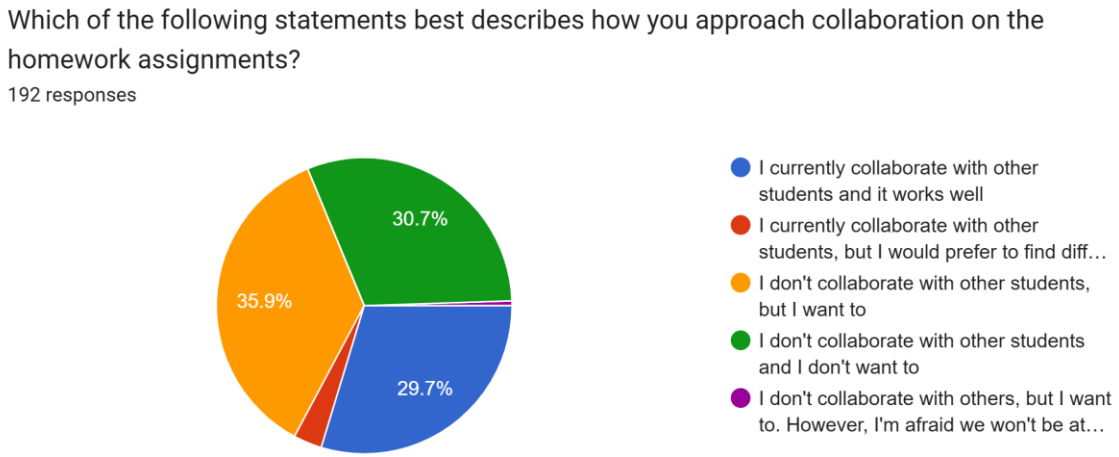




Midsemester Feedback Report

- 29.7% students collaborate and it works great

- 35.9% of students don't yet collaborate but want to. If you're in this group, consider reaching out to a friend. If you don't know anyone, sign up for the Hw5 collaboration form to find collaborators: https://forms.gle/CjjjFAEdgz78v7y19



Learning Goals

- Read and write data from **files**

- Implement and use **helper functions** in code to break up large problems into solvable subtasks

Reading Data from Files

Reading Data From Files

As we start building more complex programs, we'll often need to refer to data stored elsewhere on the computer. That means we need to **read data from a file**.

Recall that all the files on your computer are organized in **directories**, or **folders**. The file structure in your computer is a **tree** – directories are the inner nodes (recursively nested) and files are the leaves.

When you're working with files, always make sure you know which sequence of folders your file is located in. A sequence of folders from the top-level of the computer to a specific file is called a **filepath**.

For example, **Users > krivers > Documents > sample.txt** refers to the file **sample.txt** in the **Documents** folder, which is in the **krivers** folder, which is in the **Users** folder, which is at the top level of the computer.

Opening Files in Python

To interact with a file in Python we'll need to access its contents. We can do this by using the built-in function `open(filepath)`. This will create a **File object** which we can read from or write to.

```
f = open("/Users/krivers/Documents/sample.txt")
```

`open` can either take a full filepath or a **relative path** (relative from the location of the python file). It's often easiest to put the file you want to read/write in the same directory as the python file so you can simply refer to the filename directly.

```
f = open("sample.txt")
# if .py file is in Documents, will search for this file there
```

Reading and Writing from Files

When we open a file we can specify whether we plan to **read from** or **write to** the file. This will change the **mode** we use to open the file.

```
filename = "sample.txt"
f = open(filename, "r") # read mode
text = f.read() # reads the whole file as a single string
# or
lines = f.readlines() # reads the lines of a file as a list of strings

f = open("sample2.txt", "w") # write mode
f.write(text) # writes a string to the file
```

Only one instance of a file can be kept open at a time, so you should always **close** a file once you're done with it.

```
f.close()
```

Be Careful When Programming With Files

WARNING: when you write to files in Python, backups are not preserved. If you overwrite a file, the previous contents are gone forever.

Be careful when writing to files! Make sure you're using the correct filename before you run the program. Avoid overwriting original data whenever possible; you can always wait and delete it after you're done.

Activity: Read a File

You do: Download the file `chat.txt` from the schedule page and move it to the same folder as a python script.

Try using `open` and `read` to open the file and read the contents, then `print` the contents.

Common file reading issues:

- make sure the file is actually in the same directory as your python script (check directory in the `%cd` line when you run Thonny)
- make sure the filename you've entered is actually the right filename (including the filetype at the end!)

Helper Functions

Helper Functions

In Hw5 and Hw6, and in projects you might work on outside of 15-110, you'll often need to write many functions that work together to solve a larger problem.

We briefly talked about how to call functions from other functions when we first learned about function definitions and calls. Let's revisit the idea now.

We refer to a function that solves a subpart of a larger problem as a **helper function.** By breaking up a large problem into multiple smaller problems and solving those problems with helper functions, we can make complicated tasks more approachable.

Designing Helper Functions

In Hw5 and Hw6 we've broken a problem down into helper functions for you. But if you work on a separate project, you'll need to do this process on your own.

Try to identify **subtasks** that are repeated or are separate from the main goal; break down the problem into smaller parts. Have **one subtask per function** to keep things simple.

Example: Tic-Tac-Toe

Consider the game tic-tac-toe. It seems simple, but it involves multiple parts to play through a whole game.

Discuss: what are the subtasks of tic-tac-toe?

Breaking down Tic-Tac-Toe

Let's organize our tic-tac-toe game based on four core subtasks:

`makeNewBoard()`: construct and return the starter board (a 2D list of strings)

`showBoard(board)`: display a given board

`takeTurn(board, player)`: let the given player (`"X"` or `"O"`) make a move on the board, returning the updated board

`isGameOver(board)`: return `True` or `False` based on whether or not the game is over

We'll only go over how to implement each helper function briefly. The most important thing right now is how we **use the helper functions** in the main code.

Start With Assumptions

We'll start by **assuming the helper functions already work**. Write a function that calls each helper function in the appropriate place.

Start by calling `makeNewBoard` to generate the board. Display the starting state by calling `showBoard`.

Use a loop to iterate over every turn in the game. Alternate a Boolean variable to decide whether it's X's or O's turn, and call `takeTurn` on the board *and the appropriate player* to decide which move to make. Call `showBoard` again each time to show the updated board.

Keep looping until the game is over by checking `isGameOver` in the loop condition.

```
def playGame():
    print("Let's play tic-tac-toe!")
    board = makeNewBoard()
    showBoard(board)
    player1Turn = True
    while not isGameOver(board):
        if player1Turn:
            board = takeTurn(board, "X")
        else:
            board = takeTurn(board, "O")
        showBoard(board)
        player1Turn = not player1Turn
    print("Goodbye!")
```

makeNewBoard and showBoard

makeNewBoard and showBoard are simple; we can program these just using concepts we've already learned.

The board will be a 3x3 2D list with "." for empty spaces, "X" for player X, and "O" for player O.

Note that makeNewBoard takes no parameters and returns a board, whereas showBoard takes the board and returns None. They match how we used them before!

```
# Construct the tic-tac-toe board
def makeNewBoard():
    board = []
    for row in range(3):
        # Add a new row to board
        board.append([".", ".", "."])
    return board

# Print the board as a 3x3 grid
def showBoard(board):
    for row in range(3):
        line = ""
        for col in range(3):
            line += board[row][col]
        print(line)
```

takeTurn

takeTurn has the user input the row and col they want to fill in using our old friend input. This is also similar to programs we've written before!

Check to make sure the row and col are numbers with isdigit and ensure that they select a valid and unfilled space with if statements.

Keep looping until a valid location is chosen. Update the board at that spot, then return the updated board.

```
# Ask the user to input where they want
# to go next with row,col position
def takeTurn(board, player):
  while True:
    row = input("Enter a row for " + player + ":")
    col = input("Enter a col for " + player + ":")
    # Make sure it's a number!
    if row.isdigit() and col.isdigit():
      row = int(row)
      col = int(col)
      # Make sure it's in the grid!
      if 0 <= row < 3 and 0 <= col < 3:
        if board[row][col] == ".":
          board[row][col] = player
          # stop looping when move is made
          return board
        else:
          print("That space isn't open!")
      else:
        print("Not a valid space!")
    else:
      print("That's not a number!")
```

`isGameOver` needs more helper functions

`isGameOver` is a bit more complicated. There are multiple scenarios where the game can end- if a player gets three in a row horizontally, or vertically, or diagonally. The game can also end if the board is filled.

Use more helper functions to break up the work into parts!

`horizLines`, `vertLines`, `diagLines`: take a board, return a list of strings (all the lines of that type in the board).

`isFull`: take a board, return `True` if the board is full, `False` otherwise.

Now we can write the function assuming these helpers already work.

```
# True if game is over, False if not
def isGameOver(board):
    if isFull(board):
        return True
    allLines = horizLines(board) + \
               vertLines(board) + \
               diagLines(board)
    for line in allLines:
        if line == "XXX" or \
           line == "OOO":
            return True
    return False
```

isGameOver Helpers

Again, we can create the helper functions for `isGameOver` using familiar logic.

```
# Generate all horizontal lines
def horizLines(board):
  lines = []
  for row in range(3):
    lines.append(board[row][0] + \
                 board[row][1] + \
                 board[row][2])
  return lines

# Generate all vertical lines
def vertLines(board):
  lines = []
  for col in range(3):
    lines.append(board[0][col] + \
                 board[1][col] + \
                 board[2][col])
  return lines
```

```
# Generate both diagonal lines
def diagLines(board):
  leftDown = board[0][0] + \
             board[1][1] + \
             board[2][2]
  rightDown = board[0][2] + \
              board[1][1] + \
              board[2][0]
  return [ leftDown, rightDown ]

# Check if the board has no empty spots
def isFull(board):
    for row in range(3):
        for col in range(3):
            if board[row][col] == ".":
                return False
    return True
```

Functions Work Together

Put it all together and you've got a fully working Tic-Tac-Toe game!

The most important takeaways are:

- Use **helper functions** to separate out complicated subtasks and make the overall task easier to represent
- Thoughtfully consider **which data** will need to be passed into each helper function call so it can find the correct answer
- Keep track of **which data** will be returned by each function call

Learning Goals

- Read and write data from **files**

- Implement and use **helper functions** in code to break up large problems into solvable subtasks