#4-1: Top-Down Design

CS SCHOLARS – PROGRAMMING

Hw3 Takeaways

Lots of people haven't finished yet – that's okay! You can keep working on the assignment this week, up until **Thursday at noon**.

Range and list indexing: when you have a loop that iterates over indexes for a list, **pay close attention to the range!** If it isn't the default `range(len(lst))` it may visit different values than what you expect. Also, remember: to get the last value of a list you use `lst[len(lst)-1]`, **not** `lst[len(lst)]`.

Testing code: don't forget to test your code as you go so you can catch small errors, like rounding incorrectly or forgetting to sort the result! You can always call the test function in the interpreter, then call the function shown in the `assert` to see what it's actually returning.

Final Evaluation on Thursday

We'll do a **final evaluation** on Thursday during the afternoon session.

You'll complete homework-like problems during the class session. You can use the slides and online documentation but will need to complete the problems independently (no collaboration).

This is an opportunity for you to evaluate how much you've learned over the past four weeks and what you still need to work on.

Learning Goals

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

Helper Functions

In real life the code you write will be bigger than a single function. 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 call a function that solves a subpart of a larger problem 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

How can you break a project into helper functions?

Try to identify **subtasks** that are repeated or are separate from the main goal. Have **one subtask per function** to keep things simple. Use these functions to **break down the main function** into individual steps.

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()`, which constructs and returns the starter board (a 2D list of strings)

`showBoard(board)`, which displays a given board

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

`isGameOver(board)`, which returns `True` or `False` based on whether or not the game is over

We'll build the whole game from scratch, but the most important thing to focus on 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 using only the 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 isnumeric 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.isnumeric() and col.isnumeric():
      row = int(row)
      col = int(col)
      # Make sure its 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! Generate strings holding all rows/columns/diagonals with horizLines, vertLines, and diagLines. Check if the board is already full with isFull.

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

```python
# True if game is over, False is 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
```

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

isGameOver Helpers

```
# 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

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