#5-2: Large Projects

CS SCHOLARS – PROGRAMMING

Learning Goals

Use **try/except** structures to handle direct user input in code

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

Try/Except Structures

Reminder: Getting Input from the User

Recall from earlier in the program that the built-in function `input(msg)` displays a message in the interpreter, lets the user type a response in the interpreter, then **returns the response** as a string when the user presses enter.

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

`input` will **always** return a string. If we want to use a user's response as a number, we need to use type-casting to change it.

```
age = int(input("Enter your age: "))
print("You'll be", age + 1, "next year")
```

Handle User Errors with Try-Except Statements

What happens if we ask the user to enter a number and try to convert their text to a number, but they enter a non-number instead?

The code will throw a `ValueError` when it tries to convert the text to an int. This is not great, because users get frustrated if the program crashes each time they make a mistake.

In order to make a program robust against human errors, we can use a **try-except** control structure to recover from such errors.

Try-Except Statements

A **Try-Except** statement looks like this:

```
try:
    <try-block>
except:
    <what to do if the try code throws an error>
```

This works a bit like an if-else statement. Go to the `try` block first. If the code in the `try` block runs correctly, the `except` block is skipped. Alternatively, if Python encounters a runtime error in the `try` block, it immediately exits that block and jumps to the beginning of the `except` block.

Example: Inputting a Number

Let's try our age-entering program again, this time with error handling.

```
try:
    age = int(input("Enter your age:"))
    print("You'll be", age + 1, "next year")
except:
    print("That's not a real age!")
```

Note that the first print statement does not run if the user enters a non-number into the input.

Example: Opening a File

Try-except statements are also useful when you write code that needs to read from files! They provide an easy way to deal with files that aren't where they're supposed to be.

```
try:
    f = open("data.txt", "r")
    text = f.read()
    f.close()
except:
    print("Could not find data.txt")
    text = ""
```

Activity: Write error-catching code

You do: write a short snippet of code that asks the user to enter two numbers (with two separate `input()` calls), then prints the result of multiplying those two numbers. If at least one of the inputs isn't a number, print an error message using an `except` block.

Test your code by trying good inputs (two inputs) and bad inputs of different kinds.

User Input Loops Ensure Correct Inputs

Sometimes you might need to try several times to get the user to input a valid option into the program.

When you need to get a real input from a user, use a **loop** to continue asking them for input until they get it right.

What's the loop control variable? You could use the variable you're setting based on the user's entry. You could also use `while True`, then `break` when you get the right input.

Example: Entering y/n

For example, let's write a simple program that requires the user to respond with either y (yes) or n (no).

```
answer = ""
while True:
    answer = input("Do you like ice cream? [y/n]:")
    if answer == "y" or answer == "n":
        break
    else:
        print("Seriously, answer the question.")
if answer == "y":
    print("Me too!")
else:
    print("Lactose intolerance sucks :(")
```

Activity: Update the Code

You do: modify your code from before to force the user to keep entering answers with `input` until they actually give you two integers. Use a `while` loop to do this!

Helper Functions

Helper Functions

In Hw5 (and in projects you might work on outside of this program), 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 call a function that solves part of a larger problem this way 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.

We used helper functions in the simulation framework to break up a simulation into different parts!

Designing Helper Functions

How can you determine which helper functions are needed to solve a problem?

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()`, which constructs and returns the starter board

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

`takeTurn(board, player)`, which lets the given player make a move on the board

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

We'll program the whole thing as a class, but the most important thing to focus on is how we **use the helper functions** in the main code.

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.

We'll **call** these functions in a main function that will actually run the game.

```
# 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` uses the concepts we just went over in the User Input section!

Have the user input the row and col they want to fill in. Check to make sure the row and col are numbers with `try`/`except` and ensure that they show 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:
    try:
      row = int(input("Enter a row for " + \
                      player + ":"))
      col = int(input("Enter a col for " + \
                      player + ":"))
      # 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
          break
        else:
          print("That space isn't open!")
      else:
        print("Not a valid space!")
    except:
      print("That's not a number!")
  return board
```

`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`.

```
# 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 ]
```

`isGameOver` and `isFull`

We can also make a separate function to check whether the board is full.

Now all we need to do in `isGameOver` is call our functions. First, check whether the board is full. If it isn't, generate all the lines and check whether any hold `"XXX"` or `"OOO"`. Much easier!

Note that when we call the helper functions, we have to **pass in the needed data** as arguments to the call. For now, that's just the board.

```
# 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

# 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
```

Put it All Together

Now we can finally write the main function!

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()
    player1Turn = True
    while not isGameOver(board):
        if player1Turn:
            board = takeTurn(board, "X")
        else:
            board = takeTurn(board, "O")
        showBoard()
        player1Turn = not player1Turn
    print("Goodbye!")
```

Learning Goals

Use **try/except** structures to handle direct user input in code

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