Managing Large Code Projects

15-110 – Monday 11/01

Announcements

- **Check5** was due today
- **Check4/Hw4** revisions due **tomorrow**
- **Quiz4** on **Wednesday**
- **Reminder – election day tomorrow!**
  - www.vote411.org to find your ballot

Learning Goals

- Read and write data from **files**
- Use **try/except** structures to manage code that might raise errors outside of your control
- 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**.

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("sample.txt")
```

`open()` can either take a full filepath or a **relative path** (relative from the location of the python file). It's usually 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.

Reading and Writing from Files

When we open a file we need to 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
lines = f.readlines() # reads the lines of a file as a list of strings
# or
text = f.read() # reads the whole file as a single string

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**.

WARNING: if you have multiple Python files open in Pyzo and you try to open a file from a relative path, Pyzo might get confused. To be safe, when working with files, only have one file open in Pyzo at a time. And make sure to '**Run File as Script**' when working with files.

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.

If Python says a filename doesn't exist when you're sure that it does, raise your hand to get help; there's a few common problems that can occur.

Common file reading issues:

- make sure the file is actually in the same directory as your python script
- make sure the filename you've entered is actually the filename (including the filetype at the end!)
- make sure you're using **Run File as Script** (execute usually won't work)
- make sure only one file is open in Pyzo

Try-Except Structures

Handling User Errors

As we start building programs that draw data from outside the program, we may run into errors caused by the **user** instead of the code. For example, a file might not be located in the correct place, or may contain data in the wrong format.

```
f = open("data.txt", "r") # crashes if data.txt not in place
```

We don't want our code to throw runtime errors if the user makes a mistake, because the error message **isn't helpful** – it doesn't tell the user what actually went wrong. 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: Loading a File

Let's try loading a file again, this time with error handling.

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

Note that if the file isn't located, the variable `text` is still assigned an initial value. That means the rest of the program can still run properly!

Example: getting data from input

Try-except structures are also great when you need to get data directly from the user using the input built-in function! In the following example, `int` will crash if the input is not a number, but that's okay- the try-except handles it.

```
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.

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.

Helper Functions

Helper Functions

In Hw5 and Hw6 (and in projects you might work on outside of 15-110), 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 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.

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()`, 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 only go over how each function works briefly. The most important thing right now 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 one of the concepts we just went over in the Managing Errors 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(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!")
```

Learning Goals

- Read and write data from **files**
- Use **try/except** structures to manage code that might raise errors outside of your control
- Implement and use **helper functions** in code to break up large problems into solvable subtasks

Feedback: https://bit.ly/110-f21-feedback