Managing Large Code Projects

15-110 – Friday 11/11

Announcements

- **Hw5** is due **Monday**
  - If you haven't started yet, do so now!

- Exam2 grades will be released by Monday morning

Learning Goals

- Read and write data from **files**
- Install **external modules** and **import** them into 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**.

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
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()
```

Sidebar: Special Characters

As we start working with file text, we'll need to account for characters that commonly show up in files but are hard to represent in string values. These include the enter character (**newline**) and the tab character (**tab**). We can't type these directly into a string, so we'll use a shorthand instead:

```
"ABC\nDEF" # '\n' = newline, or pressing enter/return
"ABC\tDEF" # '\t' = tab
```

The \ character is a special character that indicates an **escape sequence**. It is modified by the letter that follows it. These two symbols are treated as a single character by the interpreter.

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.

Python Modules

Recap: Python Modules

The Python programming language comes with a large set of built-in functions that cover a range of different purposes. However, it would take too long to load all these functions every time we want to run a program.

Python organizes its different functions into **modules**. When you run Python, it loads only a small set of functions from the built-in module. To use any other functions, you must **import** them.

Pre-Installed Modules

We've already used a few core modules for homework assignments - mainly `math` and `tkinter`.

For a full list of python libraries, look here:
https://docs.python.org/3/library/index.html

External Modules

There are many other libraries that have been built by developers outside of the core Python team to add additional functionality to the language. These modules don't come as part of the Python language, but can be added in. We call these **external modules**.

In order to use an external module, you must first **install** it on your machine. To install, you'll need to download the files from the internet to your computer, then integrate them with the main Python library so that the language knows where the module is located.

Finding Useful Modules

One of the main strengths of Python as a language is that there are thousands of external modules available, which means that you can start many projects based on work others have done instead of starting from scratch.

You can find a list of popular modules here: wiki.python.org/moin/UsefulModules

And a more complete list of pip-installable modules here: pypi.org (we'll define what pip is in a moment)

Install External Modules with `pip`

In order to use an external module, you must first **install** it on your machine. To install, you'll need to download the files from the internet to your computer, then integrate them with the main Python library so that the language knows where the module is located.

It is usually possible to install modules manually, but this process can be a major pain. Luckily, Python also gives us a streamlined approach for installing modules – the **`pip` module**! This feature can locate modules that are indexed in the Python Package Index (a list of commonly-used modules), download them, and attempt to install them.

Running `pip`

Traditionally, programmers run `pip` from the **terminal**. This is a command interface that lets you make changes directly to your computer. You can access this in the application **Terminal** (Mac/Linux) or **Command Prompt** (Windows). To run `pip` in the terminal, use the command:

```
pip install module-name
```

This will identify the module and start the download and installation process. It may run into a **dependency error** if the module needs a second module to already be installed – in general, installing that module and then running `pip` again will fix the problem.

Thonny Has its Own Installer

If you're using Thonny, you can install modules without using pip!

Go to **Tools > Manage Packages**. Search for the module you want to install and click on it in the results. Then click 'Install' and Thonny will handle the pip installation for you.

You do: try installing the popular mathematics module **numpy**. Go to Tools > Manage Packages, search for numpy, and click **Install**.

Using an Installed Module

Once you've successfully installed a module, you should be able to put

`import module-name`

at the top of a Python file, and it will load the module the same way it would load a regular library. For example:

```
import numpy
```

Note: this may fail if you have multiple versions of Python installed on your machine and you install in the terminal. Make sure to use the `pip` associated with the version of Python you're using in your editor. You can check your editor's version in Thonny with Help > About Thonny, then call `pip` using

`python`*`version-number`* `-m pip install` *`module-name`*

Lots of Libraries

There are thousands of useful libraries out there that you can install and use!

As a starting place, check out the bonus slides on the course website for some popular libraries that CMU students tend to like to use.

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 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 **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()`, 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 only go over how to implement each 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 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.

```
# 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
```

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**
- Install **external modules** and **import** them into files
- Implement and use **helper functions** in code to break up large problems into solvable subtasks