Assignment 3

COMP 208 Fall 2019

posted:	Monday, Nov. 4, 2019
due:	Monday, Nov. 18, 2019 at 11:59

Primary Learning Objectives

By the end of this assignment, students should be able to...

- perform operations on lists of lists, including creating, indexing, and searching.
- understand how to raise exceptions when invalid input is given.
- plot simple graphs in Matplotlib based on mathematical functions.
- load files from disk and translate them into an appropriate Pythonic format.
- understand the basics of object-oriented programming, including how to define classes, constructors and methods.

Submission Instructions

- Please store all your files in a folder called **Assignment3**, zip the folder and then **submit the file Assignment3.zip** to myCourses. (See the instructions on myCourses for more details on how to zip files.) Inside your zipped folder, there must be the following files (**please use these exact file names**).
  - `convergence.py`
  - `taylorsum.png`
  - `banner.py`
  - `dungeon.py`

  Any deviation from these requirements will result in deduction of points.
- Don't worry if you realize you made a mistake after submitting your zip file: you can submit multiple times on myCourses. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and myCourses may be overloaded during rush hours).

Penalties

- Late assignments will be accepted up to 2 days (48 hours) after the due date and will be penalized by 10 points per day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct 10 points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it wrong file submitted, wrong file format submitted or any other reason. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
- **Note that the output obtained by running your code should be exactly as specified in the instructions for each questions. Failure to do so will result in deduction of points.**
- If your program does not work at all, e.g. it gives an error and does not produce any output, you will automatically get zero points for that question. If your program executes without errors but produces incorrect output, partial grades may be awarded based on the correctness of the code.
- You are expected to put **comments in each file**, on average 1 comment for every 2 or 3 lines, explaining what the lines are doing. Failure to comment your code in this manner will result in deduction of points.
- You are expected to use **descriptive names for your variables** whenever possible (depending on the question). Do not use variable names like x, y or z. Instead, use names like user_input, sum_of_numbers, or average_value. Failure to give your variables descriptive names will result in deduction of points.
- If anything is unclear, it is up to you to clarify it by either directly asking a TA during office hours, or making a post on the myCourses discussion board.
- Note: Some students may know more advanced ways in Python to do these questions. Although it may be more efficient or optimal to use more advanced concepts, it defeats the purpose of the assignment. We would like to have a level playing field for everyone; therefore, **please only use the topics/concepts discussed in the class up to and including November 4 (the day assignment-3 was posted)**. Marks may be taken off for assignment solutions that use Python concepts taught after November 4.
- You must not use any third-party libraries for this assignment, except Matplotlib for Question 2.

Question 1 [25 points]

In this question, we will implement the Taylor series in Python and, using Matplotlib, check the accuracy of the approximation versus the actual value of $ln(x+1)$, which is what the series computes.

There are two parts to this question. The first part is to write a Python function called **`taylor_sum`**, which computes the Taylor series expansion for $ln(x+1)$, as given by the following alternating series:

$$\ln(x+1) = \lim_{n\to\infty} x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots \pm \frac{x^n}{n}$$

where $-1 < x \leq 1$.

Your function **`taylor_sum`** should take two arguments: a real number k and an integer $n >= 1$. It should then compute n terms of the above series to find $ln(k+1)$. (If k is not between -1 and 1, a ValueError should be raised instead.)

Next, you will use this function to create a graph using Matplotlib. In one figure, we will plot several lines. First, for x in the range $-0.9, -0.8, ..., 0.8, 0.9, 1$, calculate the Taylor sum for x and plot the line given by the coordinates `(x, taylor_sum(x, n))`. Plot one line in this manner for each $n \in \{1, 3, 5, 9\}$. Finally, plot $ln(x+1)$ for x in the same range as above. $ln(x+1)$ can be calculated in Python using the `math.log(x+1)` function (make sure to import math first). This final line should be of black color and have line width of 2.

When plotting each of the five lines, please label each line with the value of n used to generate the line; for the final line showing $ln(x+1)$, use the label `"ln(x+1)"`. Make sure that the plot contains a legend with each label.

Finally, give your plot an appropriate title, and save the figure to disk as `taylorsum.png`. You must include both your code file and this .png file in your .zip submission. You can save a figure using the function `savefig` as: `plt.savefig("taylorsum.png")`.

See below for what your figure should look like.

Filename

You must write this program in a file called **convergence.py**.

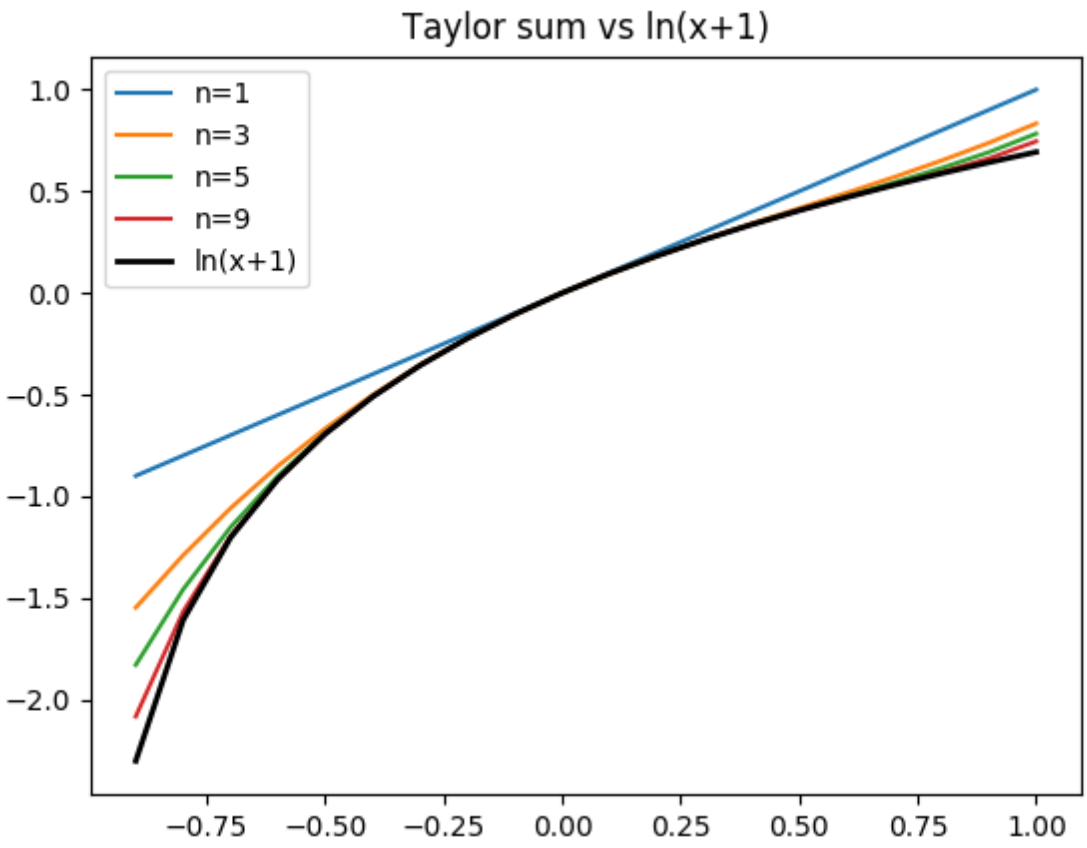

Figure 1: Sample image for Question 1.

Question 2 [35 points]

For this question, you are given a text file `"banner_letters.txt"` which contains data used to display letters in a large size akin to a banner. The data consists of the number sign (`'#'`) (also called hash or pound sign) and space character (`' '`) used to represent 26 letters from `A` to `Z`.

As you will see when you open the `"banner_letters.txt"` file, it contains the data for each letter in the following way:

- Each banner letter from `A` to `Z` is represented by a sequence of **seven lines**. For example, lines 1-7 make up the letter `A`, lines 8-14 make up the letter `B`, and so on.
- The file consists of 182 lines; 7 lines for each of the 26 letters from `A` to `Z`.
- For a given banner letter, each of its seven lines in the file will have same length. For example, lines 1-7 for the letter `A` are all nine characters long, while lines 176-182 for the letter `Z` are all eight characters long. Since the letter `I` is more narrow than other letters, each of its seven lines (57-63) has only four characters each. (But, you do not actually need to worry about the fact that different letters, such as `A` and `I`, have such a difference in their line lengths.)

Shown below are the first two letters `A`, `B`, and the last letter `Z`. Note that only the number sign (`'#'`) and spaces (`' '`) will be present in the file. The dots (`'·'`) are only shown below to indicate the presence of empty spaces in the file. To get a better idea, take a look inside `"banner_letters.txt"` to see how the banner letters appear in it.

```
1  ···###···
2  ··##·##··
3  ·##···##·
4  ##·····##
5  #########
6  ##·····##
7  ##·····##
8  ########·
9  ##·····##
10 ##·····##
11 ########·
12 ##·····##
13 ##·····##
14 ########·
```

```
########
·····##·
····##··
···##···
··##····
·##·····
########
```

The Task

You must write a function `make_banner` that takes one argument `text`, and returns a string containing the text in a banner format described below and shown in the examples on the next page. The function should do the following:

- The argument `text` can contain uppercase letters (`A` to `Z`) as well as lowercase letters (`a` to `z`). If `text` contains any characters or symbols other than those letters `A` to `Z` or `a` to `z`, the function should raise a `ValueError` with the message `"All characters in text must be alphabetic."`
- The function should read in and store all the data from the file `"banner_letters.txt"`.
- For each letter in the argument `text`, you must find the corresponding seven lines forming the banner letter in the `"banner_letters.txt"` file. Treat all letters as lowercase. As shown on the next page, for EXAMPLE 1 the argument `text` is `"hello"`. The banner letter for `"h"` can be found in the file at lines 50-56, the banner letter for `"e"` can be found at lines 29-35, and so on.
- All the banner letters, obtained as mentioned above, consist of number signs (`#`) and spaces. These banner letters should be joined together side by side with a single space between them. EXAMPLE 1 on the next page shows in green the single space between the banner letters. The string thus formed by joining all the banner letters with a space between them should be the **return value** of the function.
- The function `make_banner` should only return a string. It **must not use print()** to print anything to the screen. You should run the `banner_tester.py` file to check if the returned string is correct when displayed. (Hint: it is important to put newline characters (`\n`) in correct positions in the returned string.)
- You may write more functions if you wish, but the function `make_banner` must be present in your Python program.

Filename

- You must write this program in a file called `banner.py`
- You are also given the file `banner_tester.py` which can be used to test your code. Examples shown below are given in that file.

EXAMPLE 1: **(A single space between each banner letter is highlighted as a green box with a dot)**

```
>>> result = make_banner("hello")
>>> print(result)
##     ##·########·##      ·##      · #######
##     ##·##      ·##      ·##      ·##     ##
##     ##·##      ·##      ·##      ·##     ##
#########·######  ·##      ·##      ·##     ##
##     ##·##      ·##      ·##      ·##     ##
##     ##·##      ·##      ·##      ·##     ##
##     ##·########·########·########· #######
```

EXAMPLE 2:

```
>>> result = make_banner("QuickFox")
>>> print(result)
 #######  ##     ## ####  ######  ##    ## ########  #######  ##     ##
##     ## ##     ##  ##  ##    ## ##   ##  ##       ##     ##  ##   ##
##     ## ##     ##  ##  ##       ##  ##   ##       ##     ##   ## ##
##     ## ##     ##  ##  ##       #####    ######   ##     ##    ###
##  ## ## ##     ##  ##  ##       ##  ##   ##       ##     ##   ## ##
##    ##  ##     ##  ##  ##    ## ##   ##  ##       ##     ##  ##   ##
 ##### ##  #######  ####  ######  ##    ## ##        #######  ##     ##
```

Example 3:

```
>>> result = make_banner("ZEBRA")
>>> print(result)
######## ######## ########  ########     ###
     ##  ##       ##     ## ##     ##   ## ##
    ##   ##       ##     ## ##     ##  ##   ##
   ##    ######   ########  ########  ##     ##
  ##     ##       ##     ## ##   ##   #########
 ##      ##       ##     ## ##    ##  ##     ##
######## ######## ########  ##     ## ##     ##
```

Example 4:

```
>>> result = make_banner("hello123")
ValueError: All characters in text must be alphabetic.
```

Question 3 [40 points]

In this question, we will write some classes with methods to allow the user to play a small game involving going through a dungeon looking for treasure. We will define two **classes**: a **Player** class and a **Dungeon** class. When writing your code for this question, please use the accompanying `dungeon_template.py` file. It contains important helper code and has spaces for you to fill in, corresponding to each method below. (If you do not use the file and/or your class/method names are not identical to those in the file, you will lose marks.)

The **Player** class should contain the following methods:

1. A **`constructor`** that takes in one argument: a string with the name of the player. It should set the `name` attribute to be equal to the given argument, and also set a `health` attribute to the integer 10.
2. A **`get_name`** method that returns the name of the player.
3. A **`get_health`** method that returns the health of the player.
4. A **`set_health`** method that sets the player's health to be equal to the given argument. If the given argument is not an integer, or it is an integer but not greater or equal to zero, then you must raise a `ValueError`.

The player will explore a dungeon filled with traps and treasure. The dungeon geography will be specified by a dungeon map file on disk, which you will load in and parse as a list of lists (see below for more details). A sample dungeon map file (`d1.txt`) is provided to you in the accompanying files. You can create other map files if you wish, but it is not a requirement for this assignment. As you will see when looking in the `d1.txt` file, the dungeon map file contains a grid where the top-left character (first row, first column) represents the coordinate (0, 0) of the map. Here is the contents of the `d1.txt` file, to better illustrate the representation:

```
xxxxxxxxx
xx*xxxx*x
x***x***x
x*x*x*xxx
x**xx**xx
xx****xxx
xxxx*xxxx
xxxxT***x
xxxxxxxxx
```

The dungeon map conforms to the following format:

1. It must have the same number of rows as it has columns, and it must be of size at least `4x4`.

2. There are three valid characters in the map: `T` for treasure, `*` for empty space, and `x` for an impassable wall. Any other characters are invalid.

3. There must be at least one `T` (one treasure) in the map.

When testing your code, make sure that the `d1.txt` file (or other map file, if you have created one) is in the same folder as your Python code file, so that you can load it properly.

The **Dungeon** class should contain the following methods:

1. A **`constructor`** that takes in two arguments: a string representing the filename of the dungeon map file, and an integer representing the number of traps to add to the map. It should call the `read_map_file` method (see below) to read in the map at the given filename, and store the returned list of lists into a class attribute called `dungeon_map`. Then, it should check if the above properties of the dungeon map are respected or not. If there is any problem (non-square grid, grid smaller than 4x4, invalid character, no treasure), you must raise a `ValueError` and give an appropriate message for each case. You must also raise a `ValueError` if the number of empty spaces in the map is less than the specified `num_traps` argument, or if the `num_traps` argument is lower than zero. If there are no problems, then you should set an attribute `num_traps` equal to the argument of the same name, then call the `add_traps` method (the code for which is provided to you in the template file) in order to add traps to the map (which will be represented by the `^` character).

2. The **`read_map_file`** method takes in one argument: a string containing the filename of the map file to load. It should read in the file from disk, and return a list of lists corresponding to the dungeon map contained in the file. Note that the returned list should not contain any strings; rather, it should contain n lists (where n is the number of rows in the file), and each of those lists will contain m single characters (where m is the number of columns in the file).

3. The **`get_coords_of_empty_spaces`** method takes in no arguments, and returns the x, y coordinates of empty spaces (`'*'` characters) in the dungeon map as a list of tuples. If there are no empty spaces, it should return an empty list.

4. The **`get_char_at`** method takes in one argument: a tuple containing two elements, corresponding to an (x, y) coordinate, and returns the character in the dungeon map at that (x, y) coordinate.

5. The **`visit_square`** method takes in two arguments: a tuple containing two elements, corresponding to an (x, y) coordinate, and a player object. The method returns either True (if the game is over) or False (otherwise). It should call `get_char_at` to obtain the map character at that coordinate. The game is considered over if the current square contains treasure. If the square contains a trap (i.e., `^` character), the player's health should be decremented by 1. If the player reaches 0 health in doing so, then the game is also considered over. If the game

is over, the function should (before returning True) also print a message to the screen telling the player why the game is over.

Filename

You must write this program in a file called **dungeon.py**.