Assignment 2

COMP 202 Summer 2020

posted:	Saturday, May 23, 2020
due:	Monday, June 1, at 23:59

Primary Learning Objectives

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

- Validate user input using while loops and conditional statements.
- Present menus to the user so they can indicate which commands to perform.
- Index and traverse strings and lists using loops.
- Use lists in Python to maintain multiple user inputs and calculate properties of sequences.
- Perform operations on files, including loading data to use during the program, and saving the results of calculations.

Submission instructions

- This file is contained within a .zip file. Also in the .zip file are a series of Python code files. You will write the solutions to this assignment in these Python code files. (Make sure to unzip the zip file, as you cannot write solutions in files which are inside a zip file.)
- When ready to submit, upload each Python code file to codePost under the Assignment 2 heading. (Make sure to upload each code file individually, and not the entire folder at once.)
- You can submit multiple times on codePost, so don't worry if you realize that your current submission contains an error.
- Submit early, submit often! (Computer crashes do occur, and codePost may be overloaded during rush hours.)

Coding instructions

Please read the following instructions carefully to avoid losing marks.

- You must include **your name and McGill ID number** at the top of each `.py` file that you submit. By doing so, you are certifying that the code file is entirely your own, and represents the result of your sole effort.
- You are expected to **comment your code**, on average 1 comment for every 5-6 lines.
- You are expected to use **descriptive variable names** whenever possible. Do not use variable names like x, y or z (unless you are dealing with a mathematical function). Instead, use names like `user_input`, `sum_of_numbers`, or `average_value`.

- Some questions will ask you to print text to the screen in the exact same form as given in the examples. Make sure that the output of your program **exactly matches** the output of the examples in these cases or marks will be deducted.
- Some questions will ask you to define a function. The function you define must have the **same name, parameters and return value** as specified. Further, questions may have examples to show the output that your program is required to produce. Make sure that the output of your program **exactly matches** the output of the examples. (Input values are highlighted in gray, as they should be input by the user.)
- Those with prior programming knowledge may be able to solve a question with advanced Python concepts. Although it may be more efficient to do so, it defeats the purpose of the assignment as we are testing on specific language constructs. Therefore, **please only use concepts seen in class up to and including May 25**. Solutions that use Python concepts not seen in class, or seen in class after May 25, will be penalized. Further, the use of third-party modules is not permitted.

Policies

- 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 submits or resubmits after the due date 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.
- If your program does not work at all, e.g., gives an error and does not produce any output, zero points will be given for that question. If your program executes without errors but produces incorrect output, partial marks may be awarded based on the correctness of the code.
- If anything is unclear, it is up to you to seek clarification by either directly asking a TA during office hours or making a post on Piazza.
- Work submitted in this assignment must represent your own efforts. Assignments must be done individually; you must not work in groups. Do not ask friends or tutors to do this assignment for you. You must not copy any other person's work in any manner (electronically or otherwise), nor give a copy of your work to any other person. Code similarity detection software will be run on all assignments. Students under suspicion of plagiarism will be reported to their faculty's Disciplinary Officer.

Question 1 [15 points]

Mr. Fib O. Nacci is having trouble with his Fibonacci numbers. He is not sure what numbers are part of the Fibonacci sequence and which are not. We must help him by writing a program that lets the user enter a series of numbers, and tells them which of the numbers are part of the Fibonacci sequence.

We will do this by writing a series of functions in the file `fibs.py`, as follows.

Note: for this question, and all others, we will test each of the functions below individually, and check whether they are correctly implemented (correct parameters and return value).

Function name: `is_fibonacci`

Function parameters: An integer `number`

Return value: The function should return the boolean value `True` if `number` is one of the first 1,000 numbers in the Fibonacci sequence, and `False` otherwise. (Hint: You can adapt the Fibonacci code as seen in class for this function.)

Function name: `filter_fibonacci`

Function parameters: A list of integers `numbers`

Return value: A list containing all the integers from the `numbers` list which are Fibonacci numbers (up to the 1,000th term in the sequence).

Function name: `get_integers_from_user`

Function parameters: No parameters

Return value: The function should ask the user to enter in a series of positive numbers on one line, with each number separated by a comma. The function should check each comma-separated number to ensure that it is a valid positive number (i.e., it is not negative and it only contains digits from 0 to 9 and zero or one period '.'). If any of the numbers contain a period, the number should be converted to integer type (but we can assume that any number after the period will be a 0). The function should return a sorted list (in increasing order) of all the valid numbers entered, with each number being integer type.

Topics covered

Defining functions with parameters and return values, for loops, filter pattern, string traversal, input validation

Examples (as executed in Thonny)

Some code is already provided to you in the file `fibs.py`. The two lines of code at the bottom of the file will call your functions and print the result to the screen when you run the code file. Please do not alter these lines.

Note: As in all questions, we will use these inputs when testing your program but will also use other inputs, not shown here.

Example 1:

```
>>> %Run fibs.py
Enter numbers(s): 13,0,1,8,1,2,3,5
[0, 1, 1, 2, 3, 5, 8, 13]
```

Example 2:

```
>>> %Run fibs.py
Enter numbers(s): -13,-123123,qwerty333,21a,1,3.0,BOB
[1, 3]
```

Example 3:

```
>>> %Run fibs.py
Enter numbers(s): 5,    8     ,0,1
[0, 1, 5]
```

Question 2 [15 points]

A wave of nostalgia is gripping the world – everything old is becoming new again. Vinyl record sales are booming, old TV shows are being remade, and it's even been reported that people are sending faxes again. The next technology to make a resurgence is without a doubt the floppy disk. A floppy disk is a storage medium like a CD-ROM, Blu-Ray, or USB flash drive, but was invented in the 1960s and can only store a small amount of data. For example, the most common type of floppy disk, which measures 3 and a half inches (or 90 mm), is able to store exactly 1.44 MB (1.44 megabytes) – and thus cannot store a lot of the big files we use today (audio, video and images can and often are orders of magnitude larger than 1.44 MB).

To prepare for the impending floppy renaissance, we will write some code that loads a file specified by the user and breaks it up into 1.44 MB-sized pieces, so that we can store the file by putting individual parts of it on a series of floppy disks.

In particular, we will write two functions, in the file `floppy.py`:

Function name: `load_file`

Function parameters: A string `filename`

Return value: The data contained in the file of the given `filename`, as a string.

Function name: `break_into_segments`

Function parameters: A string `data` and an integer `segment_size`

Return value: A list of strings, where each string in the list is a sequential part of the `data` parameter and takes up memory equal to (or for the last segment, equal to or less than) the given `segment_size` parameter (which gives the maximum segment size in terms of byte). (For a 3 and a half inch floppy disk, the maximum size would be 1.44 MB, or 1440000 bytes). Note that in Python, each character of a string takes 1 byte of memory.

Function name: `save_segments`

Function parameters: A list of strings `segments`

Return value: No return value

What it should do: Save to disk files disk0.txt, disk1.txt, disk2.txt, and so on; each file should contain the string at the same position of the `segments` list. E.g., disk0.txt should contain the first string in the list (`segments[0]`).

Topics covered

Defining functions with parameters and return values, for loops, counter patterns, string traversal, file IO (reading and writing)

Examples (as executed in Thonny)

Note: This example uses the text file `example1.txt`, which contains the phrase '`The quick brown fox jumps over the lazy dog`'. We will test your code on this file as well as others (we provide the file `data.txt` for you to test with as well.)

Example 1:

```
>>> %Run floppy.py
>>> data = load_file("example1.txt")
>>> print(data)
The quick brown fox jumps over the lazy dog.

>>> segments = break_into_segments(data, 6)
>>> print(segments)
['The qu', 'ick br', 'own fo', 'x jump', 's over', ' the l', 'azy do', 'g.']

>>> save_segments(segments) # will produce files disk0.txt, ..., disk6.txt
```

Question 3 [30 points]

Since the dawn of network computing, it has been necessary for user accounts to be protected by a **password** in order to make accessible their contents only to the authorized owner of the account. Likewise, there has been an ever-present danger of a user's password being too simple and their account thus susceptible to intrusion by malicious actors.

Passwords can be faulty in many ways, but the worst passwords are of two types: either they are too short and contain only lowercase letters, or they are regular words which can be found in a dictionary. The first type of password is susceptible to a 'brute-force' attack, and the second to a 'dictionary' attack.

We will write code that demonstrates these two attacks by asking the user to enter a password, and check if it is susceptible to these attacks. We also provide in the assignment two encrypted zip files, and in the program we will have a facility that tries to find the password for the zip files using the two aforementioned methods.

Hopefully, by the time you finish this question, you will recognize how important it is to have strong passwords that cannot be guessed by these attacks.

A **brute-force** attack generates every possible combination of inputs and tries each one sequentially. For a smartphone which has a 4-digit passcode, we would try all the numbers from 0000 to 9999, until the correct passcode is found. In the case of a 4-digit passcode, there are 10,000 combinations (each of the 4 digits has 10 possibilities (the digits 0-9): $10^4 = 10000$. If a password can use all the letters of the alphabet and all digits, and is limited to 8 characters, we get $(26 + 10)^8 = 2821109907456$ combinations. It may seem like a lot, but it can take minutes to go through all combinations depending on computational power, and correlations in common passwords often make the time even shorter.

Given enough time and computing power, a brute force attack will always succeed. But due to recent improvements in login screen security (timeouts that rise with the number of failed attempts), if you are using a strong password you are most likely safe from this kind of attack.

A **dictionary** attack relies on a predefined list of words that are tried in sequence. The list of words is often taken from a dictionary, but can also be taken from analyzing large password datasets to find people's most common passwords. Although a dictionary attack is not guaranteed to succeed (since the password might not be contained within the list), it is much faster than a brute-force attack and surprisingly effective.

We will write our code by defining a few functions in the file `passwords.py`, as follows:

Function name: `brute_force`

Function parameters: A string `zip_filename` (e.g., 'file.zip')

Return value: No return value.

What it should do: Try to brute-force the password of the given zip file. You should try all combinations of the lowercase letters of the alphabet (a through z). Start with passwords of length 1 (i.e., a single letter), then passwords of length 2 (e.g., 'abc', until and including password length of

4 (e.g., 'qwer'). If the correct password is found, the brute-force attack should stop and the phrase `Password found: x` should be printed to the screen, where `x` is the password that was found. After trying all passwords of a given length, the program should output `No password found of length z`, where `z` is the length that was just finished, before moving on to the next length.

We provide the function `get_alphabet_combinations`, which takes an integer parameter `pass_length` and returns all the combinations of the letters of the alphabet of the given length. We also provide the function `extract_zip`, which takes two string parameters: `zip_filename` and `password`. This function returns `True` if it was able to open the specified zip file with the given password, and `False` otherwise.

Function name: `dictionary`

Function parameters: A string `zip_filename` (e.g., 'file.zip'), and a string `dict_filename`.

Return value: No return value.

What it should do: Open the file at the given dictionary filename, and read in the list of words (each line of the file will have exactly one word on it), and then try to open the zip file with each word. If it finds the correct password, the attack should stop and the message `Password found: x` should be printed to the screen, where `x` is the password that was found. If none of the words in the file are correct, the message `No password found.` should be printed.

Function name: `test_password_strength`

Function parameters: A string `password`, and a string `dict_filename`.

Return value: No return value.

What it should do: Check if the password is susceptible to brute-force or dictionary attacks. (You will re-use some of the code from your other functions here.) If the password was susceptible to a brute-force attack (of up to 4 characters), the message `Password is susceptible to brute-force attack.` should be printed. If the password was susceptible to a dictionary attack, the message `Password is susceptible to dictionary attack.` should be printed.

Function name: `menu`

Function parameters: No parameters

Return value: No return value.

What it should do: This function will serve as the main menu for our program. It should print some introductory messages and then the list of options (see example 1 below). It will then ask the user to enter a number from 1 to 4 corresponding to the menu options. It should then ask for the appropriate for the input from the user (zip filename, dictionary filename, password) depending on which number was entered, and then call the corresponding function, giving the user's inputs as arguments. When the function finishes, the menu options should be repeated to allow the user to select another option. If a 4 is entered (for `Exit`), the message `See ya!` should be printed and the program should end. If anything else is entered, the message `Invalid input.` should be printed.

Finally, your menu must include an important note at the top. Accessing secure computer systems or password-protected data without prior authorization is a federal crime in Canada, the United States, and most other countries. You must do research to find out what Canadian federal statute discusses unauthorized computer access. In your menu, at the top (above the listing of options), you must warn the user, giving them the name and section of the federal statute, as well as the maximum penalty for violating said law.

Topics covered

Defining functions; calling functions; while loops; for loops; string traversal; nested loops; file IO (reading).

Examples (as executed in Thonny)

Example 1:

```
>>> %Run passwords.py
Welcome to the Passwords Program.
Note: (Your note here)
What would you like to do?
1) Test a password
2) Find password for zip file using brute-force
3) Find password for zip file using dictionary
4) Exit
> 1
Enter a password: test
Enter dictionary file name: common_passwords.txt
Password is susceptible to brute-force attack.
Password is susceptible to dictionary attack.
----------
Welcome to the Passwords Program.
Note: (Your note here)
What would you like to do?
1) Test a password
2) Find password for zip file using brute-force
3) Find password for zip file using dictionary
4) Exit
> 2
Enter zip file name: brute_force.zip
No password found of length 1
No password found of length 2
Password found: password removed for example
----------
Welcome to the Passwords Program.
Note: (Your note here)
What would you like to do?
```

```
1) Test a password
2) Find password for zip file using brute-force
3) Find password for zip file using dictionary
4) Exit
> 3
Enter zip file name: dictionary.zip
Enter dictionary file name: common_passwords.txt
Password found: password removed for example
----------
Welcome to the Passwords Program.
Note: (Your note here)
What would you like to do?
1) Test a password
2) Find password for zip file using brute-force
3) Find password for zip file using dictionary
4) Exit
> 4
See ya!
```

Question 4 [40 points]

Wheel of Fortune is a US game show in which contestants must solve blank word puzzles. A blank word puzzle consists of a phrase that is initially hidden from the player. They see a series of blank characters (underscores) representing each letter of the hidden phrase. They must guess letters one at a time, until there are enough filled-in blanks to guess the entire phrase.

In the first part of this question, worth 25 points, we will write code to implement a basic Wheel of Fortune type game in Python. In the second part, worth 10 points, we will write further code to create a variant of the game called 'Wheel of Misfortune.' A further 5 points will be allocated to submissions that add specific additional features to the game.

We will represent the hidden phrase (e.g., '`TO BE OR NOT TO BE`') by a string. Initially, the hidden phrase will be presented to the player as a series of underscores (blank characters) (e.g., '__ __ __ ___ __ __', with spaces separating the words). In code, we will use a list of boolean values, of length equal to the hidden phrase. Each boolean value in the list will correspond to whether the user has guessed that particular letter of the phrase or not.

At the start, all the boolean values will be `False` as the player has not yet guessed any letters correctly, except for the indices corresponding to the spaces in the hidden phrase, which are revealed by default,. As the player guesses letters, which appear at indices i1, i2, ..., etc. in the hidden phrase, then the elements at the same indices in the boolean list will change to `True`.

For example, given the following hidden phrase:

```
GOOD LUCK
```

if the user has guessed letters O and C, they will be presented with the following blanks phrase:

```
_OO_ __C_
```

This blanks phrase corresponds to the following list of boolean values (note that the `True` elements are at the indices at which the letters were guessed, as well as the indices corresponding to spaces in the hidden phrase.)

```
[False, True, True, False, True, False, False, True, False]
```

Part one (25 points)

The main game function, `play()`, has already been provided to you. Please take a moment to familiarize yourself with it. Note how it uses a while loop to continue asking the user for a new letter, until the user guesses the final letter of the phrase. You will observe that the `play()` function makes calls to several functions. These functions, listed below, are the ones you must write for the game to work properly. Note that you must make sure that your functions operate exactly as described below, including parameters and return values.

Function name: `get_random_puzzle`

Function parameters: A list of string `puzzles`

Return value: A puzzle randomly chosen from the `puzzles` list.

Function name: `count_occurrences_of_letter`

Function parameters: A string `puzzle` (representing the hidden phrase), and a string `letter`

Return value: The number of times the `letter` occurs inside the `puzzle`.

Function name: `get_starting_blanks`

Function parameters: A string `puzzle` (representing the hidden phrase)

Return value: A list of boolean values, of length equal to the length of the `puzzle`. All values should be `False`, except for the elements at indices corresponding to spaces in the `puzzle`, which should be `True` (spaces are revealed to the player by default).

Function name: `blanks_to_string`

Function parameters: A list of boolean values `blanks`, and a string `puzzle` (representing the hidden phrase)

Return value: A string where every letter of the `puzzle` is replaced by underscores ('_'), except for the letters which have already been guessed, in which case the letter should remain in the string (you can tell if the letter at index `i` of `puzzle` has been guessed by checking the boolean value at index `i` of the `blanks` list).

Function name: `update_blanks`

Function parameters: A list of boolean values `blanks`, a string `puzzle` (representing the hidden phrase), and a string `letter`

Return value: No return value

What it should do: This function will go through the `puzzle` string, and at each index `i` at which `letter` occurs, the corresponding element at index `i` of the `blanks` list should be set to `True` (other values in the `blanks` list should remain unchanged).

Function name: `check_if_game_won`

Function parameters: A list of boolean values `blanks`

Return value: The function should return the boolean value `True` if the player has guessed all letters in the hidden phrase (i.e., if all values in the `blanks` list are `True`), and return `False` otherwise.

Function name: `menu`

Function parameters: No parameters.

Return value: No return value

What it should do: This function will serve as the main menu for our program. It should print some introductory messages and then the list of options (see example 1 below). It will then ask the user to enter a number from 1 to 3 corresponding to the menu options. It should then call the appropriate function, based on the number they entered, and then print the menu options again

(repeat). (Note: the `play()` function takes one boolean argument: `False` if the regular game should be played, and `True` if the Wheel of Misfortune variant should be played.) If 3 is entered (for `Exit`), the message `See ya!` should be printed and the program should end. If anything else is entered, the message `Invalid input.` should be printed.

Part two (10 points)

Wheel of Misfortune is a variant of Wheel of Fortune that plays very similar, except the computer tries to make the player lose at every opportunity.

In particular, given a particular puzzle, e.g.,

`BE READY AT NOON`,

and a particular game state, (e.g., if the user has already guessed an E),

`_E _E___ __ ____`,

then, at the user's next guess (e.g., a B), the computer will switch the puzzle, without telling the player, to a different hidden phrase that looks exactly the same from the player's point of view, but does not include a B. For example, the computer might switch the phrase to the following:

`HE READS AT CAMP`

Note how this hidden phrase has the same number of words as the original hidden phrase, and each word is the same length. Furthermore, the letters that had already been guessed correctly (in this case, just an E) occur in exactly the same places as in the original hidden phrase.

From the player's point of view, the game state will still be

`_E _E___ __ ____`,

except that there will no longer be a B in the puzzle. (So the player has wasted their guess.)

When the computer decides to swap the puzzle, they will let out an evil laugh (this is already written for you).

We will write two functions to enable this variant. The first will check if two puzzles can be swapped (i.e., that they are 'compatible' with each other):

Function name: `check_puzzle_compatibility`

Function parameters: A list of boolean values `blanks`, a string `puzzle` (representing the hidden phrase), and a second string `puzzle2` (representing the second hidden phrase that we would like to swap to)

Return value: The function should check if the two puzzles are compatible. If the puzzles are not the same length, the function should immediately return `False`. Otherwise, the function must iterate through each character of the `puzzle`. If the letter has been guessed (by checking in the corresponding position of the `blanks` list), then the function must check if the letter is the same as the corresponding letter (same index) of `puzzle2`. If they are not the same, then the function

should return `False`. Otherwise, if all guessed letters are the same, the function should return `True`.

The second function will determine the best hidden phrase to which to swap, given the player's current guess.

Function name: `get_best_compatible_puzzle`

Function parameters: A list of boolean values `blanks`, a string `puzzle` (representing the hidden phrase), a string `letter`, and a list of strings `puzzles`.

Return value: The hidden phrase in the `puzzles` list which contains the fewest occurrences of the given `letter`, while still being compatible with the original `puzzle`. (Hint: It would be useful to call your `check_puzzle_compatibility` and `count_occurrences_of_letter` functions here.) If there is no such hidden phrase to which we can swap, return the value `None` instead.

Part three (5 points)

If you have time, for an additional five points you may choose one or more of the following enhancements to add to the game. Please make sure to describe in a comment at the top of the program which enhancement(s) you have fulfilled.

- Allow more than one player to play the game at once (i.e., each player would have a turn to guess a letter).
- In addition to letting the user guess a letter, let them guess the full phrase.
- Keep track of the guessed letters, and show the player which letters have already been guessed before their guess each turn.
- Add more pairs of hidden phrases to the `puzzles` list, where each pair is a phrase of the same number and length of words (so that they can be swapped during play depending on the user's guess.)
- Add a new list corresponding to a wheel the user can spin. The values in the list correspond to dollar amounts. When the user guesses a letter, they will obtain that amount multiplied by the number of letters in the puzzle. Or, if the letter does not exist in the puzzle, they will lose that amount of dollars. Print out their current dollar amount at each turn and at the end of the game.

Topics covered

Defining functions with parameters and return values; calling functions; for loops; list traversal; reduction pattern; mapping pattern.

Examples (as executed in Thonny)

Example 1:

```
>>> %Run wheel.py
Welcome to Wheel of (Mis)Fortune!
Guess the phrase, one letter at a time.
What would you like to do?
1) Play a game of Wheel of Fortune
2) Play a game of Wheel of Misfortune
3) Exit
> 1
Here is your phrase:
_______ __ ________
Enter a letter (a - z or A - Z): T
T______ __ ________
Enter a letter (a - z or A - Z): L
T_L____ __ ________
Enter a letter (a - z or A - Z): K
T_LK___ __ ________
Enter a letter (a - z or A - Z): I
T_LKI__ __ I_______
Enter a letter (a - z or A - Z): N
T_LKIN_ _N I_______
Enter a letter (a - z or A - Z): G
T_LKING _N I_____G_
Enter a letter (a - z or A - Z): O
T_LKING ON I_____G_
Enter a letter (a - z or A - Z): S
T_LKING ON I__SS_G_
Enter a letter (a - z or A - Z): M
T_LKING ON IM_SS_G_
Enter a letter (a - z or A - Z): E
T_LKING ON IMESS_GE
Enter a letter (a - z or A - Z): A
TALKING ON IMESSAGE
Thanks for playing!
You had 11 correct and 1 incorrect guesses.
----------
Welcome to Wheel of (Mis)Fortune!
Guess the phrase, one letter at a time.
What would you like to do?
1) Play a game of Wheel of Fortune
2) Play a game of Wheel of Misfortune
3) Exit
```

```
> 3
See ya!
```

Example 2:

```
>>> %Run wheel.py
Welcome to Wheel of (Mis)Fortune!
Guess the phrase, one letter at a time.
What would you like to do?
1) Play a game of Wheel of Fortune
2) Play a game of Wheel of Misfortune
3) Exit
> 2
Here is your phrase:
__ _____ __ ____
Enter a letter (a - z or A - Z): N
Muahaha.
__ _____ __ ____
Enter a letter (a - z or A - Z): A
Muahaha.
__ __A__ __ ____
Enter a letter (a - z or A - Z): T
__ __A__ __ ____
Enter a letter (a - z or A - Z): S
__ S_A__ __ ____
Enter a letter (a - z or A - Z): H
__ SHA__ __ ____
Enter a letter (a - z or A - Z): D
__ SHAD_ __ ____
Enter a letter (a - z or A - Z): E
__ SHADE __ ____
Enter a letter (a - z or A - Z): N
N_ SHADE _N ___N
Enter a letter (a - z or A - Z): O
NO SHADE ON _OON
Enter a letter (a - z or A - Z): M
NO SHADE ON MOON
Thanks for playing!
You had 8 correct and 2 incorrect guesses.
----------
Welcome to Wheel of (Mis)Fortune!
Guess the phrase, one letter at a time.
What would you like to do?
1) Play a game of Wheel of Fortune
2) Play a game of Wheel of Misfortune
```

```
3) Exit
> 3
See ya!
```