1. Create a program that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn 100 years old. Note: for this exercise, the expectation is that you explicitly write out the year (and therefore be out of date the next year).

   Extras:

   - Add on to the previous program by asking the user for another number and printing out that many copies of the previous message. (Hint: order of operations exists in Python)
   - Print out that many copies of the previous message on separate lines. (Hint: the string ”\n is the same as pressing the ENTER button)

2. The exercise comes first (with a few extras if you want the extra challenge or want to spend more time), followed by a discussion. Enjoy!
   Ask the user for a number. Depending on whether the number is even or odd, print out an appropriate message to the user. Hint: how does an even / odd number react differently when divided by 2?

   Extras:

   - If the number is a multiple of 4, print out a different message.
   - Ask the user for two numbers: one number to check (call it num) and one number to divide by (check). If check divides evenly into num, tell that to the user. If not, print a different appropriate message.

3. Take a list, say for example this one:
   a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
   and write a program that prints out all the elements of the list that are less than 5.

   Extras:

   - Instead of printing the elements one by one, make a new list that has all the elements less than 5 from this list in it and print out this new list.
   - Write this in one line of Python.
   - Ask the user for a number and return a list that contains only elements from the original list a that are smaller than that number given by the user.

4. Create a program that asks the user for a number and then prints out a list of all the divisors of that number. (If you don't know what a divisor is, it is a number that divides evenly into another number. For example, 13 is a divisor of 26 because 26 / 13 has no remainder.)

5. Take two lists, say for example these two:
   a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
   b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
   and write a program that returns a list that contains only the elements that are common between the lists (without duplicates). Make sure your program works on two lists of different sizes.

   Extras:
   - Randomly generate two lists to test this
   - Write this in one line of Python (don't worry if you can't figure this out at this point - we'll get to it soon)

6. Ask the user for a string and print out whether this string is a palindrome or not. (A palindrome is a string that reads the same forwards and backwards.)

7. Let's say I give you a list saved in a variable: a = *[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]*. Write one line of Python that takes this list a and makes a new list that has only the even elements of this list in it.

8. Make a two-player Rock-Paper-Scissors game. (Hint: Ask for player plays (using input), compare them, print out a message of congratulations to the winner, and ask if the players want to start a new game)

   Remember the rules:
   - Rock beats scissors
   - Scissors beat paper
   - Paper beats rock

9. Generate a random number between 1 and 9 (including 1 and 9). Ask the user to guess the number, then tell them whether they guessed too low, too high, or exactly right.

   Extras:

   - Keep the game going until the user types “exit”
   - Keep track of how many guesses the user has taken, and when the game ends, print this out.

10. This week’s exercise is going to be revisiting an old exercise (see Exercise 5), except require the solution in a different way.
    Take two lists, say for example these two:
    a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
    b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
    and write a program that returns a list that contains only the elements that are common between the lists (without duplicates). Make sure your program works on two lists of different sizes. Write this in one line of Python using at least one list comprehension.

    Extra:

    - Randomly generate two lists to test this

11. Ask the user for a number and determine whether the number is prime or not. (For those who have forgotten, a prime number is a number that has no divisors.). Take this opportunity to practice using functions.

12. Write a program that takes a list of numbers (for example, *a = [5, 10, 15, 20, 25]*) and makes a new list of only the first and last elements of the given list. For practice, write this code inside a function.

13. Write a program that asks the user how many Fibonnaci numbers to generate and then generates them. Take this opportunity to think about how you can use functions. Make sure to ask the user to enter the number of numbers in the sequence to generate.(Hint: The Fibonnaci seqence is a sequence of numbers where the next number in the sequence is the sum of the previous two numbers in the sequence. The sequence looks like this: 1, 1, 2, 3, 5, 8, 13, …)

14. Write a program (function!) that takes a list and returns a new list that contains all the elements of the first list minus all the duplicates.

    Extras:

    - Write two different functions to do this - one using a loop and constructing a list, and another using sets.

15. Write a program (using functions!) that asks the user for a long string containing multiple words. Print back to the user the same string, except with the words in backwards order. For example, say I type the string:
    My name is Michele
    Then I would see the string:
    Michele is name My

16. Write a password generator in Python. Be creative with how you generate passwords - strong passwords have a mix of lowercase letters, uppercase letters, numbers, and symbols. The passwords should be random, generating a new password every time the user asks for a new password. Include your run-time code in a main method.

    Extra:

    - Ask the user how strong they want their password to be. For weak passwords, pick a word or two from a list.

17. Use the *BeautifulSoup* and *requests* Python packages to print out a list of all the article titles on the New York Times homepage. Link

18. Create a program that will play the “cows and bulls” game with the user. The game works like this:
    Randomly generate a 4-digit number. Ask the user to guess a 4-digit number. For every digit that the user guessed correctly in the correct place, they have a “cow”. For every digit the user guessed correctly in the wrong place is a “bull.” Every time the user makes a guess, tell them how many “cows” and “bulls” they have. Once the user guesses the

correct number, the game is over. Keep track of the number of guesses the user makes throughout teh game and tell the user at the end.

Say the number generated by the computer is 1038. An example interaction could look like this:

Welcome to the Cows and Bulls Game!

Enter a number:

>>> 1234

2 cows, 0 bulls

>>> 1256

1 cow, 1 bull

...

Until the user guesses the number.

19. Using the requests and BeautifulSoup Python libraries, print to the screen the full text of the article on this website:

    http://www.vanityfair.com/society/2014/06/monica-lewinsky-humiliation-culture.

    The article is long, so it is split up between 4 pages. Your task is to print out the text to the screen so that you can read the full article without having to click any buttons.

    This will just print the full text of the article to the screen. It will not make it easy to read, so next exercise we will learn how to write this text to a *.txt* file.

20. Write a function that takes an ordered list of numbers (a list where the elements are in order from smallest to largest) and another number. The function decides whether or not the given number is inside the list and returns (then prints) an appropriate boolean.

    Extras:

    - Use binary search.

21. Take the code from the *Exercise nr. 17* and instead of printing the results to a screen, write the results to a txt file. In your code, just make up a name for the file you are saving to.

    Extras:

    - Ask the user to specify the name of the output file that will be saved.

22. Given a .txt file that has a list of a bunch of names, count how many of each name there are in the file, and print out the results to the screen. I have a .txt file for you, if you want to use it!

    Extra:

    - Instead of using the .txt file from above (or instead of, if you want the challenge), take this .txt file Link, and count how many of each "category" of each image there are.

23. Given two .txt files that have lists of numbers in them, find the numbers that are overlapping. One .txt file Link has a list of all prime numbers under 1000, and the other .txt file Link has a list of happy numbers up to 1000.

24. This exercise is Part 1 of 4 of the Tic Tac Toe exercise series. Time for some fake graphics! Let's say we want to draw game boards that look like this:

```
 --- --- ---
|   |   |   |
 --- --- ---
|   |   |   |
 --- --- ---
|   |   |   |
 --- --- ---
```

    This one is 3x3 (like in tic tac toe). Obviously, they come in many other sizes (8x8 for chess, 19x19 for Go, and many more).
    Ask the user what size game board they want to draw, and draw it for them to the screen using Python's print statement.

25. In a previous exercise, we've written a program that "knows" a number and asks a user to guess it.
    This time, we're going to do exactly the opposite. You, the user, will have in your head a number between 0 and 100. The program will guess a number, and you, the user, will say whether it is too high, too low, or your number.

At the end of this exchange, your program should print out how many guesses it took to get your number.

As the writer of this program, you will have to choose how your program will strategically guess. A naive strategy can be to simply start the guessing at 1, and keep going (2, 3, 4, etc.) until you hit the number. But that's not an optimal guessing strategy. An alternate strategy might be to guess 50 (right in the middle of the range), and then increase / decrease by 1 as needed. After you've written the program, try to find the optimal strategy! (We'll talk about what is the optimal one next week with the solution.)

26. This exercise is Part 2 of 4 of the Tic Tac Toe exercise series.

    As you may have guessed, we are trying to build up to a full tic-tac-toe board. However, this is significantly more than half an hour of coding, so we're doing it in pieces.

    Today, we will simply focus on checking whether someone has WON a game of Tic Tac Toe, not worrying about how the moves were made.

    If a game of Tic Tac Toe is represented as a list of lists, like so:

    game = [[1, 2, 0],
    [2, 1, 0],
    [2, 1, 1]]

    where a 0 means an empty square, a 1 means that player 1 put their token in that space, and a 2 means that player 2 put their token in that space.

    Your task this week: given a 3 by 3 list of lists that represents a Tic Tac Toe game board, tell me whether anyone has won, and tell me which player won, if any. A Tic Tac Toe win is 3 in a row - either in a row, a column, or a diagonal. Don't worry about the case where TWO people have won - assume that in every board there will only be one winner.

    Here are some more examples to work with:

    winner_is_2 = [[2, 2, 0],
    [2, 1, 0],
    [2, 1, 1]]

    winner_is_1 = [[1, 2, 0],
    [2, 1, 0],
    [2, 1, 1]]

winner_is_also_1 = [[0, 1, 0],
[2, 1, 0],
[2, 1, 1]]

no_winner = [[1, 2, 0],
[2, 1, 0],
[2, 1, 2]]

also_no_winner = [[1, 2, 0],
[2, 1, 0],
[2, 1, 0]]

27. This exercise is Part 3 of 4 of the Tic Tac Toe exercise series.
In a previous exercise we explored the idea of using a list of lists as a “data structure” to store information about a tic tac toe game. In a tic tac toe game, the “game server” needs to know where the Xs and Os are in the board, to know whether player 1 or player 2 (or whoever is X and O won).
There has also been an exercise about drawing the actual tic tac toe gameboard using text characters.
The next logical step is to deal with handling user input. When a player (say player 1, who is X) wants to place an X on the screen, they can’t just click on a terminal. So we are going to approximate this clicking simply by asking the user for a coordinate of where they want to place their piece.
As a reminder, our tic tac toe game is really a list of lists. The game starts out with an empty game board like this:
game = [[0, 0, 0],
[0, 0, 0],
[0, 0, 0]]
The computer asks Player 1 (X) what their move is (in the format row,col), and say they type 1,3. Then the game would print out
game = [[0, 0, X],
[0, 0, 0],
[0, 0, 0]]
And ask Player 2 for their move, printing an O in that place.

Things to note:

- For this exercise, assume that player 1 (the first player to move) will always be X and player 2 (the second player) will always be O.
- Notice how in the example I gave coordinates for where I want to move starting from (1, 1) instead of (0, 0). To people who don't program, starting to count at 0 is a strange concept, so it is better for the user experience if the row counts and column counts start at 1. This is not required, but whichever way you choose to implement this, it should be explained to the player.
- Ask the user to enter coordinates in the form "row,col" - a number, then a comma, then a number. Then you can use your Python skills to figure out which row and column they want their piece to be in.
- Don't worry about checking whether someone won the game, but if a player tries to put a piece in a game position where there already is another piece, do not allow the piece to go there.

Bonus:

- For the "standard" exercise, don't worry about "ending" the game - no need to keep track of how many squares are full. In a bonus version, keep track of how many squares are full and automatically stop asking for moves when there are no more valid moves.

28. Implement a function that takes as input three variables, and returns the largest of the three. Do this without using the Python max() function!
    The goal of this exercise is to think about some internals that Python normally takes care of for us. All you need is some variables and if statements!
29. This exercise is Part 4 of 4 of the Tic Tac Toe exercise series. In 3 previous exercises, we built up a few components needed to build a Tic Tac Toe game in Python.
    The next step is to put all these three components together to make a two-player Tic Tac Toe game! Your challenge in this exercise is to use the functions from those previous exercises all together in the same program to make a two-player game that you can play with a friend. There are a lot of choices you will have to make when completing this exercise, so you can go as far or as little as you want with it.

Here are a few things to keep in mind:

- You should keep track of who won - if there is a winner, show a congratulatory message on the screen.
- If there are no more moves left, don't ask for the next player's move!

As a bonus, you can ask the players if they want to play again and keep a running tally of who won more - Player 1 or Player 2

30. This exercise is Part 1 of 3 of the Hangman exercise series.
    In this exercise, the task is to write a function that picks a random word from a list of words from the SOWPODS dictionary Link. Download this file and save it in the same directory as your Python code. This file is Peter Norvig's compilation of the dictionary of words used in professional Scrabble tournaments. Each line in the file contains a single word.

31. This exercise is Part 2 of 3 of the Hangman exercise series.
    Let's continue building Hangman. In the game of Hangman, a clue word is given by the program that the player has to guess, letter by letter. The player guesses one letter at a time until the entire word has been guessed. (In the actual game, the player can only guess 6 letters incorrectly before losing).
    Let's say the word the player has to guess is "EVAPORATE". For this exercise, write the logic that asks a player to guess a letter and displays letters in the clue word that were guessed correctly. For now, let the player guess an infinite number of times until they get the entire word. As a bonus, keep track of the letters the player guessed and display a different message if the player tries to guess that letter again. Remember to stop the game when all the letters have been guessed correctly! Don't worry about choosing a word randomly or keeping track of the number of guesses the player has remaining - we will deal with those in a future exercise.

    An example interaction can look like this:
    >>> Welcome to Hangman!

    _ _ _ _ _ _ _ _ _
    >>> Guess your letter: S
    Incorrect!
    >>> Guess your letter: E

E _ _ _ _ _ _ E

...

And so on, until the player gets the word.

32. This exercise is Part 3 of 3 of the Hangman exercise series.
    In this exercise, we will finish building Hangman. In the game of Hangman, the player only has 6 incorrect guesses (head, body, 2 legs, and 2 arms) before they lose the game.
    In Part 1, we loaded a random word list and picked a word from it. In Part 2, we wrote the logic for guessing the letter and displaying that information to the user. In this exercise, we have to put it all together and add logic for handling guesses.

    Copy your code from Parts 1 and 2 into a new file as a starting point. Now add the following features:

    - Only let the user guess 6 times, and tell the user how many guesses they have left.
    - Keep track of the letters the user guessed. If the user guesses a letter they already guessed, don't penalize them - let them guess again.

    Optional additions:

    - When the player wins or loses, let them start a new game.
    - Rather than telling the user "You have 4 incorrect guesses left", display some picture art for the Hangman. This is challenging - do the other parts of the exercise first!
    - Your solution will be a lot cleaner if you make use of functions to help you!

33. For this exercise, we will keep track of when our friend's birthdays are, and be able to find that information based on their name. Create a dictionary (in your file) of names and birthdays. When you run your program it should ask the user to enter a name, and return the birthday of that person back to them. The interaction should look something like this:

    >>> Welcome to the birthday dictionary. We know the birthdays of:
    Albert Einstein
    Benjamin Franklin

Ada Lovelace
>>> Who's birthday do you want to look up?
Benjamin Franklin
>>> Benjamin Franklin's birthday is 01/17/1706.
Happy coding!

34. This exercise is Part 2 of 4 of the birthday data exercise series.
    In the previous exercise we created a dictionary of famous scientists' birthdays. In this exercise, modify your program from Part 1 to load the birthday dictionary from a JSON file on disk, rather than having the dictionary defined in the program.

    Bonus:
    - Ask the user for another scientist's name and birthday to add to the dictionary, and update the JSON file you have on disk with the scientist's name. If you run the program multiple times and keep adding new names, your JSON file should keep getting bigger and bigger.
35. In the previous exercise we saved information about famous scientists' names and birthdays to disk. In this exercise, load that JSON file from disk, extract the months of all the birthdays, and count how many scientists have a birthday in each month.

    Your program should output something like:
    {
        "May": 3,
        "November": 2,
        "December": 1
    }

36. This exercise is Part 4 of 4 of the birthday data exercise series.
    In the previous exercise, we counted how many birthdays there are in each month in our dictionary of birthdays.
    In this exercise, use the *bokeh* Python library to plot a histogram of which months the scientists have birthdays in! Because it would take a long time for you to input the months of various scientists, you can use my scientist birthday JSON file. Just parse out

the months (if you don't know how, I suggest looking at the previous exercise or its solution) and draw your histogram.

37. Write a Program to create a class by name Students, and initialize attributes like name, age, and grade while creating an object.

38. Write a Program to create an empty valid class by name Students, with no properties.

39. Write a program, to create a child class Teacher that will inherit the properties of parent class Staff.

40. Write a Program, to create a class and using the class instance print all the writable attributes of that object.

41. Create a class Teacher with name, age, and salary attributes, where salary must be a private attribute that cannot be accessed outside the class.

42. Write a Python program that overloads the operator + and > for a custom class.

43. Write a Python program that checks if one class is a subclass of another.

44. Write a Python program that lists out all the default as well as custom properties of the class.

45. Write a Program in Python to implement a Stack Data Structure Link using Class and Objects, with push, pop, and traversal method.

46. Write a program that prints the class name using its object.

47. Write a Python class Square, and define two methods that return the square area and perimeter.
48. Write 2 **mixins** and implements their in one class. Explain how multiple inheritance works.

49. Declare an **interface** animal with 2 or 3 methods and implements interface in 2 different classes.

50. Encapsulate properties in a class using **property** decorator and private properties.