1. In an Ancient Kingdom, the currency consists of bronze coins, silver coins, and gold coins. There are 20 bronze coins in one silver coin and 15 silver coins in one gold coin. Write a **function** that will return a converted amount of bronze coins into the fewest amount of coins possible. Only return a string with the non-zero values, meaning don't return something similar to "0 silver coins". The argument for the function will be *bronze_coins* (how many bronze coins to convert)., if no argument is provided then the **default** should be 900 bronze coins

   Examples:

   - convert_bronze(32) → "1 silver 12 bronze",
   - convert_bronze(544) → "1 gold 4 silver 4 bronze",
   - convert_bronze(903) → "3 gold 3 bronze"
     Note: Do **not** output 3 gold 0 silver 3 bronze.
   - convert_bronze() → "3 gold"
     Note: Do **not** output 3 gold 0 silver 0 bronze.

2. (Game: heads or tails) Write a **function** that lets the user guess whether the flip of a coin results in heads or tails. The function randomly generates an integer 0 or 1, which represents head or tail. The function returns if the guess is correct or incorrect. The argument for the function will be *guess* (the guess of the user, 0 for heads and 1 for tails), if no argument is provided then the **default** should be 0 for heads.
   Hint: Use the following lines of code to create the function.

   ```
   from random import randint
   value = randint(0,1) #picks a random integer. Either 0 or 1.
   ```

   Examples:

   - toss_coin( ) → "Correct!" (if the random value is 0) or "Incorrect!" (if the random value is 1),
   - toss_coin(0) → "Correct!" (if the random value is 0) or "Incorrect!" (if the random value is 1),
   - toss_coin(1) → "Correct!" (if the random value is 1) or "Incorrect!" (if the random value is 0)

3. (Game: Odd or Even) Write a **function** that lets the user guess whether a randomly generated number is odd or even. The function randomly generates an integer between 0 and 9 (inclusive) and returns whether the user's guess is correct or incorrect. The argument for the function will be *guess* (the user's guess, either "odd" or "even"), if no argument is provided then the **default** guess should be even.
   Hint: Use the following lines of code to create the function.

   ```
   from random import randint
   value = randint(0,9) #picks a random integer between 0-9 inclusive
   ```

   Examples:

   - guess( ) → "Correct!" (if random value is even) or "Incorrect!" (if random value is odd)
   - guess("odd")→"Correct!" (if random value is odd) or "Incorrect!" (if random value is even)
   - guess("even") → "Correct!" (if random value is even) or "Incorrect!" (if random value is odd)

4. Write a **function** that returns the number of copies of the same number. The arguments for the function will be *num_1* (first number), *num_2* (second number), and *num_3* (third number), if no argument is provided then the **default** for all 3 values should be 0.

   Examples:

   - count_duplicates(2, 3, 2) → "There are 2 of the same number",
   - count_duplicates(4, 4, 4) → "There are 3 of the same number",
   - count_duplicates(1, 2, 3) → "Each number is unique"
   - count_duplicates(1) → "There are 2 of the same number"
   - count_duplicates(0) → "There are 3 of the same number"

5. Write a **function** to create a game of Rock, Paper, Scissors. The function will return the winner of the game played by two players. The arguments to the function will be *player*1 (the first player's choice) and *player*2 (the second player's choice), if no argument is provided then the **default** for either player should be Rock.
   Print the winner according to the following rules.

   - Rock beats Scissors
   - Scissors beats Paper
   - Paper beats Rock

   Examples:

   - find_winner("Rock", "Paper") → "Player 2 wins!",
   - find_winner("Scissors", "Paper") → "Player 1 wins!",
   - find_winner("Rock", "Rock") → "It's a tie!"
   - find_winner("Rock") → "It's a tie!"
   - find_winner( ) → "It's a tie!"
   - find_winner("Scissors") → "Player 2 wins!"

6. Luke Skywalker has friends and family, but he is getting older and having trouble remembering them all. Write a **function** that will return the relation defined in the table below. The arguments to the function will be *name* (name of the person related to Luke), if no argument is provided then the **default** should be nothing. That is, the empty word " ".

Person	Relation
Darth Vader	Father
Leia	Sister
Han	Brother in law
R2D2	Droid

   *If he types any other name, return "unknown".

   Examples:

   - find_relation("Darth Vader") → "Father",
   - find_relation("R2D2") → "Droid",
   - find_relation("Jabba the Hutt") → "Unknown"

- find_relation( ) → "Unknown"

7. Given a positive integer n, the following rules will always create a sequence that ends with 1, called Hailstone Sequence:

   (a) If n is even, divide by 2

   (b) If n is odd, multiply by 3 and add 1 (i.e. $3n + 1$)

   (c) Continue until n is 1

   Write a **function** that prints the hailstone sequence starting at n. The argument to the function will be n (the integer to start the sequence from), if no argument is provided then the **default** should be 40. **Examples:**

   - hailstone_seq(25) → 25, 76, 38, 19, 58 ... 8, 4, 2, 1,
   - hailstone_seq(40) → 40, 20, 10, 5, 16, 8, 4, 2, 1
   - hailstone_seq( ) → 40, 20, 10, 5, 16, 8, 4, 2, 1

8. Write a **function** that takes 3 numbers as arguments, *num_1* (first number), *num_2* (second number), and *num_3* (third number). *num_1* should be mandatory. If no arguments are provided for *num_2* or *num_3* then use 5 for *num_2* and 25 for *num_3*. Return a list of the integers in ascending order. You may **not** use the built-in functions *max*(), *min*(), *sort*(), or *sorted*().

   Examples:

   - ascending_order(2, 3, 1) → [1, 2, 3],
   - ascending_order(10, 1) → [1, 10, 25],
   - ascending_order(50) → [5, 25, 50]

9. Write a **function** that takes 3 numbers as arguments, *num_1* (first number), *num_2* (second number), and *num_3* (third number). *num_1* should be mandatory. If no arguments are provided for *num_2* or *num_3* then use 15 for *num_2* and 5 for *num_3*. Return a list of the integers in descending order. You may **not** use the built-in functions *max*(), *min*(), *sort*(), or *sorted*().

   Examples:

   - descending_order(2, 3, 1) → [3, 2, 1],
   - descending_order(10) → [15, 10, 5],
   - descending_order(2, 45) → [45, 5, 2]

10. Write a **function** that takes two arguments, a list and a value. The function should return the indices of all occurrences of the *value* in the list, if no argument is provided then the **default** should be to find 0.

    Examples:

    - get_indices([1, 0, 5, 0, 7]) → [1, 3]
    - get_indices([1, 5, 5, 2, 7], 7) → [4]
    - get_indices([1, 5, 5, 2, 7]) → [ ]
    - get_indices([1, 5, 5, 2, 7], 5) → [1, 2]
    - get_indices([1, 5, 5, 2, 7], 8) → [ ]
    - get_indices(["a", "a", "b", "a", "b", "a"], "a") → [0, 1, 3, 5]

11. Write a **function** that returns the factors of a given integer. The argument of the function will be *num* (integer to find factors for), if no argument is provided then the **default** should be 36.

    Examples:

    - find_factors(12) → 1, 2, 3, 4, 6, 12,
    - find_factors(17) → 1, 17,
    - find_factors(36) → 1, 2, 3, 4, 6, 9, 12, 18, 36
    - find_factors( ) → 1, 2, 3, 4, 6, 9, 12, 18, 36

12. Write a **function** that takes two numbers as arguments *num* and *length* and returns a list of multiples of *num* until the list length reaches *length*, if no argument is provided then the **default** for the list length should be 5.

    Examples:

    - list_of_multiples(7, 5) → [7, 14, 21, 28, 35]
    - list_of_multiples(12, 10) → [12, 24, 36, 48, 60, 72, 84, 96, 108, 120]
    - list_of_multiples(2) → [2, 4, 6, 8, 10]
    - list_of_multiples(2,3) → [2, 4, 6]