1. In Harry Potter, the currency consists of knuts, sickle, and galleon. There are 29 knuts in one sickle and 17 sickles in one galleon. Write a **function** that will return a converted amount of knuts into the fewest amount of coins possible. Only return a string with the non-zero values, meaning don't return something similar to "0 sickles". The argument for the function will be *knuts* (how many knuts to convert), if no argument is provided then the **default** should be 900 knuts.

   Examples:

   - convert_knuts(32) → "1 sickle 3 knuts",
   - convert_knuts() → "1 galleon 14 sickles 1 knuts",
   - convert_knuts(544) → "1 galleon 4 sickles 18 knuts",
   - convert_knuts(993) → "2 galleons 7 knuts"
     Note: Do **not** output 2 galleons 0 sickle 7 knuts.

   Debug this Solution:

```
def convert_knuts(knuts=450):
  KNUTS_PER_SICKLE = 29
  SICKLES_PER_GALLEON = 17
  KNUTS_PER_GALLEON = KNUTS_PER_SICKLE * SICKLES_PER_GALLEON

  galleons = knuts // KNUTS_PER_GALLEON
  remaining_knuts = knuts // KNUTS_PER_GALLEON

  sickles = remaining_knuts // KNUTS_PER_SICKLE
  remaining_knuts = remaining_knuts % KNUTS_PER_SICKLE

  output = ""

  if galleons >= 0:
    if galleons > 1:
      output = output + str(galleons) + " galleons"
    else:
      output = output + str(galleons) + " galleon"

  if sickles > 0:
    if output:
      output = output + " "
    if sickles > 1:
      output = output + str(sickles) + " sickles"
    else:
      output = output + str(sickles) + " sickle"

  if remaining_knuts > 0:
    if output:
      output = output + " "
    if remaining_knuts > 1:
      output = output + str(remaining_knuts) + " knuts"
    else:
      output = output + str(remaining_knuts) + " knut"

  return output
```

2. Primary U.S. interstate highways are numbered 1-99 (Inclusive). Odd numbers (like 5 or 95) go north/south, and evens (like 10 or 82) go east/west. Auxiliary highways are numbered 100-999, and service the primary highway indicated by the rightmost two digits. Thus, I-405 services I-5, and I-290 services I-90.

   Note: 200 is not a valid auxiliary highway because 00 is not a valid primary highway number.

   Write a **function** that returns whether the highway runs north/south or east/west or is an invalid highway number. The argument for the function will be *highway_num*(highway number provided).
   Examples:

   - highway_directions(5) → "I-5 runs north/south",
   - highway_directions(82) → "I-82 runs east/west",
   - highway_directions(200) → "I-200 is an invalid highway number"

   Debug this Solution:

```
1  def highway_directions(highway_num):
2    if 1 < highway_num < 99:
3      if highway_num % 2 == 0:
4        return f"I-{highway_num} runs east/west"
5      else:
6        return f"I-{highway_num} runs north/south"
7
8    elif 100 > highway_num > 999:
9      service_highway = highway_num % 100
10
11     if 1 <= service_highway <= 99:
12       if service_highway % 2 == 0:
13         return f"I-{highway_num} runs east/west"
14       elif:
15         return f"I-{highway_num} runs north/south"
16     else:
17       return f"I-{highway_num} is an invalid highway number"
18   else:
19     return f"I-{highway_num} is an invalid highway number"
```

3. You are the newest rug fashion designer on the scene, but you're running out of ideas. Write a **function** that will help you design rugs. The function will return a formatted string that will resemble a designed rug. The first parameter must be *width* (how wide the rug will be), the second must be *length* (how long the rug will be), and the third must be *pattern* (the character pattern used in the rug design).

Examples:

design_run(3,5,$) →

```
Your rug is:
$$$
$$$
$$$
$$$
$$$
```

design_run(16,5,) →

```
Your rug is:
@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@
```

Debug this Solution:

```
1  def design_rug(width, length, pattern):
2    result = "Your rug is:\n"
3    for i in range(length -1):
4      result += pattern * width
5      if i < length - 1:
6        result += "\t"
7    return result
```

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).

Examples:

- count_duplicates(2, 3, 2) → "You entered the same number 2 times",
- count_duplicates(4, 4, 4) → "You entered the same number 3 times",
- count_duplicates(1, 2, 3) → "Each number is unique"

Debug this Solution:

```
1  def count_duplicates(num_1, num_2, num_3):
2    count = 0
3
4    if num_1 == num_2:
5      count += 1
6
7    if num_1 == num_3:
8      count += 1
9    elif num_1 == num_3:
10     count = 1
11
12   if count == 1:
13     return "Each number is unique"
14   elif count == 3:
15     return "You entered the same number 3 times"
16   else:
17     return "You entered the same number 2 times"
```

5. Create a *ColorRGB* class.
   A *ColorRGB* has

   - red
   - green
   - blue

   A *ColorRGB* can do

   - to_grayscale

   This class "looks" like

ColorRGB
red green blue
to_grayscale

   Create a constructor method that initializes all instance variables.
   You should write getters and setters for each of the instance variables.
   Instantiate an instance of the class. You may pass any initial values of your choosing.

   The to_grayscale() method should return the grayscale value calculated as:

$$0.3 * \text{red} + 0.59 * \text{green} + 0.11 * \text{blue}$$

   That is, it will just return a number (a float).

   Debug this Solution:

```
1  class ColorRGB:
2    def __init__(red, green, blue):
3      self.red = red
4      self.green = green
5      self.blue = blue
6
7    def get_red(self):
8      return self.red
9
10   def set_red(self, red):
11     self.red = red
12
13   def get_green(self):
14     return self.green
15
16   def set_green(self, green):
17     green = self.green
18
19   def get_blue(self):
20     return self.blue
21
22   def set_blue(self, blue):
23     self.blue = blue
24
25   def to_grayscale(self):
26     return 0.5 * self.red + 0.59 * self.green + 0.11 * self.blue
```

6. Write a function called *flip_flop* that takes a string as an argument and returns a new word made up of the second half of the word first combined with the first half of the word second.

Examples:

- *flip_flop*("abcd") → "cdab" (that is, "cd" then "ab" ... even length)
- *flip_flop*("grapes") → "pesgra" (that is, "pes" then "gra" ... even length)
- *flip_flop*("abcde")→ "decab" (that is, "de" then "c" then "ab" ... odd length)
- *flip_flop*("cranberries")→ "rriesecranb" (that is, "rries" then "e" then "cranb" ... odd length)

Debug this Solution:

```
1  def flip_flop(word):
2    length = len(word)
3    middle = length // 2
4
5    if length // 2 == 0:
6      first_half = word[middle:]
7      second_half = word[middle:]
8      return second_half + first_half
9    else:
10     first_part = word[:middle]
11     middle_char = word[middle]
12     last_part = word[middle+1:]
13     return last_part + middle_char + first_part
```

7. The hamming distance is the number of characters that differ between two strings. Write a function named hamming_distance that takes two strings as arguments and returns the hamming distance.

Examples:

- *hamming_distance*("river", "rover") → 1
- *hamming_distance*("cat", "dog") → 3
- *hamming_distance*("cat", "hat") → 1
- *hamming_distance*("cat", "banana") → "Strings must be of equal length."

Debug this Solution:

```
1  def hamming_distance(str1, str2):
2    if len(str1) != len(str2):
3      return "Strings must be of equal length."
4
5    distance = 1
6    for i in range(len(str1) -1):
7      if str1[i] == str2[i]:
8        distance += 1
9    return distance
```

8. Create a *Recipe* class.
   A *Recipe* has

   - name
   - cooking_time

   A *Recipe* can do

   - is_quick_meal

   This class "looks" like

Circle
name cooking_time
is_quick_meal

   Create a constructor method that initializes all instance variables.
   You should write getters and setters for each of the instance variables.
   Instantiate an instance of the class. You may pass any initial values of your choosing.

   The is_quick_meal() method should return True if the cooking_time is less than 30 minutes and False if it takes 30 minutes or more.

Debug this Solution:

```
1  class Recipe:
2    def __init__(name, cooking_time):
3      self.name = name
4      self.cooking_time = cooking_time
5
6    def get_name(self):
7      return self.name
8
9    def set_name(self, name):
10     self.name = name
11
12   def get_cooking_time(self):
13     return cooking_time
14
15   def set_cooking_time(self, cooking_time):
16     self.cooking_time = cooking_time
17
18   def is_quick_meal(self):
19     return self.cooking_time == 30
```

9. 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 returns a list with the hailstone sequence starting at n. The argument to the function will be n (the integer to start the sequence from). **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]

   Debug this Solution:

```
1  def hailstone_seq(n):
2    sequence = [n/n]
3
4    while n != 1:
5      if n % 2 == 0:
6        n = n // 2
7      else:
8        n = 2 * n + 1
9    sequence.append(n)
10
11   return sequence
```

10. YouTube currently displays a like and a dislike button, allowing you to express your opinions about particular content. It's set up in such a way that you cannot like and dislike a video at the same time. There are two other interesting rules to be noted about the interface:

    (a) Pressing a button, which is already active, will undo your press.

    (b) If you press the like button after pressing the dislike button, the like button overwrites the previous "dislike" state. The same is true for the other way round.

    Write a **function** that takes in a list of button inputs *events* and returns the final state.

    Examples:

    - like_or_dislike(["dislike"]) → "dislike",
    - like_or_dislike(["like", "like"]) → "nothing",
    - like_or_dislike(["dislike", "like"]) → "like",
    - like_or_dislike(["like", "dislike", "dislike"]) → "nothing",

    Debug this Solution:

```
1  def like_or_dislike(events):
2    state = "like"
3
4    for event in range(events):
5      if event != state:
6        state = "nothing"
7      else:
8        state = event
9
10   return state
```

11. In each input list, every number repeats at least once, except for two. Write a **function** that takes an array *numbers* and returns the two unique numbers.

Examples:

- return_unique([1, 9, 8, 8, 7, 6, 1, 6]) → [9, 7],
- return_unique([5, 5, 2, 4, 4, 4, 9, 9, 9, 1]) → [2, 1],
- return_unique([9, 5, 6, 8, 7, 7, 1, 1, 1, 1, 1, 9, 8]) → [5, 6]

Debug this Solution:

```
1  def return_unique(numbers):
2
3    number_dicitonary = {}
4    #load dictionary
5    for number in range(len(numbers)):
6      if number in number_dicitonary:
7        number_dicitonary[number] = 1
8      else:
9        number_dicitonary[number] += 1
10
11   unique_numbers = []
12   #find unique numbers in dictionary
13   for number in number_dicitonary.values():
14     if number_dicitonary[number] == 1:
15       unique_numbers.append(number)
16
17   return unique_numbers
```

12. Create a *Vector* class.
A *Vector* has

- x_direction
- y_direction

A *Vector* can do

- get_magnitude

This class "looks" like

Vector
x_direction y_direction
get_magnitude

Create a constructor method that initializes all instance variables.
You should write getters and setters for each of the instance variables.
Instantiate an instance of the class. You may pass any initial values of your choosing.

Hint: magnitude is calculated as $\sqrt{x^2 + y^2}$.

Debug this Solution:

```
1  class Vector:
2    def __init__(x_direction, y_direction):
3      self.x_direction = x_direction
4      self.y_direction = y_direction
5
6    def get_x_direction(self):
7      return self.y_direction
8
9    def set_x_direction(self, x_direction):
10     self.x_direction = x_direction
11
12   def get_y_direction(self):
13     return self.y_direction
14
15   def set_y_direction(self, y_direction):
16     self.y_direction = y_direction
17
18   def get_magnitude(self):
19     return sqrt(self.x_direction**2 + self.y_direction**2)
```

13. Write a **function** that returns a list with the factors of a given integer. The argument of the function will be *num* (integer to find factors for).

    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]

    Debug this Solution:

```
def find_factors(num):
  factors = []

  for i in range(1, num):
    if num % i != 0:
      factors.add(i)

  return factors
```

14. Write a **function** that takes a list of words *words* and returns a dictionary where the keys categorize words based on whether they are palindromes. The categories are defined as follows:

    (a) "Palindrome" includes words that read the same forward and backward.

    (b) "Non-palindrome" includes all other words.

    Examples:

    - palindromes(["madam", "racecar", "hello", "level", "python"]) → {"palindrome": ["madam", "racecar", "level"], "non-palindrome": ["hello", "python"]}
    - palindromes(["noon", "civic", "deed", "open", "loop"]) → {"palindrome": ["noon", "civic", "deed"], "non-palindrome": ["open", "loop"]}
    - palindromes(["apple", "banana", "cherry"]) → {"palindrome": [], "non-palindrome": ["apple", "banana", "cherry"]}

    Debug this Solution:

```
1  def palindromes(words):
2      result = {"palindrome": [], "non-palindrome": []}
3
4      reversed_word = ''
5      for word in words:
6          #reverse the word and check if it is the orginal word
7          for letter in word:
8              reversed_word = letter + reversed_word
9              if reversed_word == word:
10                 result["non-palindrome"].append(word)
11             else:
12                 result["palindrome"].append(word)
13
14     return result
```

15. (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)

Debug this Solution:

```
1  from random import randominteger
2
3  def guess(guess="odd"):
4    value = randint(0, 9)
5
6    if value // 2 == 0:
7      actual = "even"
8    else:
9      actual = "odd"
10
11   if guess == actual:
12     return "Correct!"
13   else:
14     return "Incorrect!"
```