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.

   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


# Test the function with a sample input

print(convert_knuts(32)) # Expected output: "1 sickle 3 knuts"

print(convert_knuts()) # Expected output: "1 galleon 14 sickles 1 knuts"

print(convert_knuts(544)) # Expected output: "1 galleon 4 sickles 18 knuts"

print(convert_knuts(993)) # Expected output: "2 galleons 7 knuts"
# Note: convert_knuts(993) will not output 2 galleons 0 sickle 7 knuts
```

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).

   Debug this Solution:

```
def highway_directions(highway_num):
  if 1 <= highway_num <= 99:
    if highway_num % 2 == 0:
      return f"I-{highway_num} runs north/south"
    else:
      return f"I-{highway_num} runs east/west"


  elif 100 <= highway_num <= 999:
    service_highway = highway_num % 100

    if 1 <= service_highway <= 99:
      if service_highway % 2 == 0:
        return f"I-{highway_num} runs east/west"
      elif service_highway % 2 == 0:
        return f"I-{highway_num} runs north/south"
    else:
      return f"I-{highway_num} is an invalid highway number"
  else:
    return f"I-{highway_num} is an invalid highway number"

# Test the function with a sample input

print(highway_directions(5)) # Expected output: "I-5 runs north/south"
print(highway_directions(82)) # Expected output: "I-82 runs east/west"
print(highway_directions(200)) # Expected output: "I-200 is an invalid highway number"
print(highway_directions(353)) # Expected output: "I-353 runs north/south"
```

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
8
9  print(design_rug(3, 5, '$')) # Expected output: "Your rug is:\n$$$\n$$$\n$$$\n$$$\n$$$"
10 print(design_rug(16, 5)) # Expected output: "Your rug is:\n@@@@@@@@@@@@@@@@\
     n@@@@@@@@@@@@@@@@\n@@@@@@@@@@@@@@@@\n@@@@@@@@@@@@@@@@\n@@@@@@@@@@@@@@@@"
```

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"
18
19 # Test the function with a sample input
20
21 print(count_duplicates(2, 3, 2)) # Expected output: "You entered the same number 2
       times"
22 print(count_duplicates(4, 4, 4)) # Expected output: "You entered the same number 3
       times"
23 print(count_duplicates(1, 2, 3)) # Expected output: "Each number is unique"
```

5. 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.

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
14
15 # Test the function with a sample input
16 print(flip_flop("abcd")) # Expected output: "cdab" (that is, cd then ab ... even length
       )
17 print(flip_flop("grapes")) # Expected output: "pesgra" (that is, pes then gra ... even
       length)
18 print(flip_flop("abcde")) # Expected output: "decab" (that is, de then c then ab ...
       odd length)
19 print(flip_flop("cranberries")) # Expected output: "rriesecranb" (that is, rries then e
        then cranb ... odd length)
```

6. 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.

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
10
11 # Test the function with a sample input
12 print(hamming_distance("river", "rover")) # Expected output: 1
13 print(hamming_distance("cat", "dog")) # Expected output: 3
14 print(hamming_distance("cat", "hat")) # Expected output: 1
15 print(hamming_distance("cat", "banana")) # Expected output: Strings must be of equal
       length.
```

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 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).

   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 = 3 * n + 1
9    sequence.append(n)
10
11   return sequence
12
13
14 # Test the function with a sample input
15 print(hailstone_seq(25)) # Expected output: [25, 76, 38, 19, 58 ... 8, 4, 2, 1]
16 print(hailstone_seq(40)) # Expected output: [40, 20, 10, 5, 16, 8, 4, 2, 1]
```

8. 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.

   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
12 # Test the function with a sample input
13 print(like_or_dislike(["dislike"])) # Expected output: "dislike"
14 print(like_or_dislike(["like", "like"])) # Expected output: "nothing"
15 print(like_or_dislike(["dislike", "like"])) # Expected output: "like"
16 print(like_or_dislike(["like", "dislike", "dislike"])) # Expected output: "nothing"
```

9. 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.

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
18
19
20 # Test the function with a sample input
21 print(return_unique([1, 9, 8, 8, 7, 6, 1, 6])) # Expected output: [9, 7]
22 print(return_unique([5, 5, 2, 4, 4, 4, 9, 9, 9, 1])) # Expected output: [2, 1]
23 print(return_unique([9, 5, 6, 8, 7, 7, 1, 1, 1, 1, 1, 9, 8])) # Expected output: [5, 6]
```

10. 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).

Debug this Solution:

```
1  def find_factors(num):
2    factors = []
3
4    for i in range(1, num):
5      if num % i != 0:
6        factors.add(i)
7
8    return factors
9
10 # Test the function with a sample input
11 print(find_factors(12)) # Expected output: [1, 2, 3, 4, 6, 12]
12 print(find_factors(17)) # Expected output: [1, 17]
13 print(find_factors(36)) # Expected output: [1, 2, 3, 4, 6, 9, 12, 18, 36]
```

11. 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.

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
16 # Test the function with a sample input
17 print(palindromes(["madam", "racecar", "hello", "level", "python"]))
18 # Expected output: {'palindrome': ['madam', 'racecar', 'level'], 'non-palindrome': ['
       hello', 'python']}
19
20 print(palindromes(["noon", "civic", "deed", "open", "loop"]))
21 # Expected output: {'palindrome': ['noon', 'civic', 'deed'], 'non-palindrome': ['open',
       'loop']}
22
23 print(palindromes(["apple", "banana", "cherry"]))
24 # Expected output: {'palindrome': [], 'non-palindrome': ['apple', 'banana', 'cherry']}
```

12. (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
```

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   print ('random value: ' + actual)
12   print ('guess value: ' + guess)
13   if guess == actual:
14     return "Correct!"
15   else:
16     return "Incorrect!"
17
18 # Test the function with a sample input
19 print("\nFinal result: "+ guess()) # Expected output: "Correct!" (Only if random value
       is even) or "Incorrect!" (Only if random value is odd)
20 print(40*"-") # Separator for clarity
21
22 print("\nFinal result: "+ guess("odd"+"\n")) # Expected output: "Correct!" (Only if
       random value is odd) or "Incorrect!" (Only if random value is even)
23 print(40*"-") # Separator for clarity
24
25 print("\nFinal result: "+ guess("even"+"\n")) # Expected output: "Correct!" (Only if
       random value is even) or "Incorrect!" (Only if random value is odd)
26 print(40*"-") # Separator for clarity
```