Python Exercises - Part II [solutions]

Python and R for Data Science

Data Science and Management

LUISS

Exercise 1: find number of unique characters

Define a function `count_uniq` that:

- takes as arguments:
    - a string `s`
- returns:
    - the number of unique characters in `s`

In [154]:

```
# Solution goes here
```

Solution

In [155]:

```
def count_uniq(s):
    return len(set(s))
```

Test your code

Run this code to test your solution:

In [156]:

```
try: assert count_uniq("test") == 3 and count_uniq("Aejeje") == 3 and not print("
except: print('Test failed')
```

```
Test passed
```

Exercise 2: remove duplicates

Define a function `remove_duplicates` that:

- takes as arguments:
    - a list `s` of strings
- returns:
    - a copy of `s` without duplicate elements

```
In [157]: # Solution goes here
```

Solution

```
In [158]: def remove_duplicates(s):
              return list(set(s))
```

Test your code

Run this code to test your solution:

In [159]:

```
try: assert sorted(remove_duplicates(["test", "luiss", "data", "test", "science"]
except: print('Test failed')
```

```
Test passed
```

Exercise 3: find common elements

Define a function `common_elements` that:

- takes as arguments:
    - a set `s` of strings
    - a set `k` of strings
- returns:
    - a set containing all common elements between `s` and `k`

In [160]:

```
# Solution goes here
```

Solution

In [161]:

```python
def common_elements(s, k):
    return s.intersection(k)
```

Test your code

Run this code to test your solution:

In [162]:

```
friend1_companies = {'Google', 'Amazon', 'Apple', 'Microsoft'}
friend2_companies = {'Facebook', 'Google', 'Tesla', 'Amazon'}
try: assert common_elements(friend1_companies, friend2_companies) == {'Google', '
except: print('Test failed')
```

```
Test passed
```

Exercise 4: count word frequency

Define a function `word_freq` that:

- takes as arguments:
    - a string `s`
- returns:
    - a dictionary containing the frequency (value) within `s` of each word (key) contained in `s`.

NOTE:

- count the word case-insensitive.
- split `s` by space to obtain the words.

In [163]:

```
# Solution goes here
```

Solution

In [164]:

```
def word_freq(s):
    # Convert the input string 's' to lowercase and split it into individual word
    words = s.lower().split()

    # Create an empty dictionary to store the frequency of each word
    word_count = {}

    # Iterate over each word in the list of words
    for word in words:
        # Increment the word's count in the dictionary by 1
        # If the word is not already in the dictionary, set its count to 1
        word_count[word] = word_count.get(word, 0) + 1

    # Return the dictionary containing word frequencies
    return word_count
```

Test your code

Run this code to test your solution:

In [165]:

```
try: assert word_freq("Python is fun and learning Python is fun") == {'python': 2
except: print('Test failed')
```

```
Test passed
```

Exercise 5: track voting results

Define a function `update_votes` that:

- takes as arguments:
    - a dictionary `votes` having as key names of candidates and as value the number of votes received by each one of them
    - a list `new_votes` of names of candidates
- returns:
    - the `votes` dictionary updated with the new votes received

In [166]:

```
# Solution goes here
```

Solution

In [167]:

```python
def update_votes(votes, new_votes):
    # Iterate over each 'vote' in the list of 'new_votes'
    for vote in new_votes:
        # Update the count of each 'vote' in the 'votes' dictionary
        # If the 'vote' already exists in the dictionary, increment its count by
        # If the 'vote' does not exist, set its count to 1
        votes[vote] = votes.get(vote, 0) + 1

    # Return the updated 'votes' dictionary with the new vote counts
    return votes
```

Test your code

Run this code to test your solution:

In [168]:

```
votes = {
    'Alice': 120,
    'Bob': 150,
    'Charlie': 90
}

new_votes = ['Alice', 'Charlie', 'Charlie', 'Bob', 'Alice', 'Alice']
try: assert update_votes(votes, new_votes) == {'Alice': 123, 'Bob': 151, 'Charlie
except: print('Test failed')
```

```
Test passed
```

Exercise 6: find how many equal numbers

Define a function `count_equals` that:

- takes as arguments four numbers
- returns:
    - the maximum number of equal numbers between the four

Example:

- `count_equals(1,2,3,4)` should return `0`
- `count_equals(1,2,5,4)` should return `0`
- `count_equals(1,2,2,2)` should return `3` because there are three `2` in the sequence
- `count_equals(1,1,1,2)` should return `3` because there are three `1` in the sequence

In [169]:

```
# Solution goes here
```

Solution

In [170]:

```
def count_equals(a, b, c, d):
    # Create a list 'numbers' containing the four input values
    numbers = [a, b, c, d]

    # Initialize 'max_count' to store the highest occurrence of any number
    max_count = 0

    # Iterate over each 'num' in the 'numbers' list
    for num in numbers:
        # Initialize a 'count' variable to track how many times 'num' appears
        count = 0

        # Iterate over each 'other' number in 'numbers' to compare with 'num'
        for other in numbers:
            # If 'num' is equal to 'other', increment 'count' by 1
            if num == other:
                count += 1

        # If the current 'count' is greater than 'max_count', update 'max_count'
        if count > max_count:
            max_count = count

    # Return 'max_count' if it's greater than 1 (indicating duplicates exist)
    # Otherwise, return 0 (no duplicates found)
    return max_count if max_count > 1 else 0
```

Test your code

Run this code to test your solution:

```
In [171]: try: assert count_equals(1,2,3,4) == 0 and count_equals(1,5,3,4) == 0 and count_e
          except: print('Test failed')
```

```
Test passed
```

Exercise 7: Fibonacci’s sequence

Define a function `fibonacci` that:

- takes as arguments:
    - an integer number `n`
- returns:
    - a list containing the first `n` numbers of the Fibonacci's sequence

NOTE: The Fibonacci sequence is a series of numbers where each number is the sum of the two previous ones, starting with 0 and 1. To calculate it, you begin with 0 and 1, then add these to get the next number. Continue this process to generate the sequence. It goes 0, 1, 1, 2, 3, 5, 8, and so on.

In [172]:

```
# Solution goes here
```

Solution

```
In [173]: def fibonacci(n):
              # Initialize a list 'fib_sequence' with the first two Fibonacci numbers: 0 an
              fib_sequence = [0, 1]

              # If 'n' is 1, return a list containing only the first Fibonacci number [0]
              if n == 1:
                  return [0]

              # Loop from 2 to 'n-1' to generate the next Fibonacci numbers
              for i in range(2, n):
                  # Calculate the next Fibonacci number as the sum of the last two numbers
                  next_fib = fib_sequence[-1] + fib_sequence[-2]
                  # Append the new Fibonacci number to the 'fib_sequence' list
                  fib_sequence.append(next_fib)

              # Return the first 'n' numbers of the Fibonacci sequence (in case 'n' is less
              return fib_sequence[:n]
```

Test your code

Run this code to test your solution:

In [174]:

```
try: assert fibonacci(1) == [0] and fibonacci(3) == [0,1,1] and fibonacci(7) == [
except: print('Test failed')
```

```
Test passed
```

Exercise 8: zero-sum triplets

Define a function `zero_sum_triplets` that:

- takes as arguments:
    - a list of integers `numbers`
- returns:
    - the number of triplets whose sum is zero

Example:

- `zero_sum_triplets([1,-1,0,7,12])` should return `1` because the sum of 1,-1,0 is `0`
- `zero_sum_triplets([1,9,0,7,12])` should return `0` because there are no triplets that sum up to zero
- `zero_sum_triplets([1,-9,8,6,-14])` should return `2` because the sum of 1,-1,0 is `0` and the sum of 8,6,-14 is `0`

In [175]:

```
# Solution goes here
```

Solution

In [176]:

```
def zero_sum_triplets(numbers):
    # Get the length of the input list 'numbers'
    n = len(numbers)

    # Initialize 'count' to store the number of triplets that sum to zero
    count = 0

    # Use a triple nested loop to check all possible triplet combinations
    # Outer loop iterates from the first element to the third-last element (i)
    for i in range(n - 2):
        # Middle loop starts from the element after 'i' (j) and goes to the secon
        for j in range(i + 1, n - 1):
            # Inner loop starts from the element after 'j' (k) and goes to the la
            for k in range(j + 1, n):
                # Check if the sum of the three numbers is equal to zero
                if numbers[i] + numbers[j] + numbers[k] == 0:
                    # If so, increment the 'count' by 1
                    count += 1

    # Return the total count of zero-sum triplets found
    return count
```

Test your code

Run this code to test your solution:

In [177]:

```
try: assert zero_sum_triplets([1,-1,0,7,12]) == 1 and zero_sum_triplets([1,9,0,7,
except: print('Test failed')
```

```
Test passed
```

Exercise 9: Collatz

Define a function `collatz` that:

- takes as argument an integer number `n`
- returns:
    - a list containing all the numbers generated by the Collatz conjecture (stopping when reaching `1`)

NOTE: The Collatz Conjecture is a mathematical problem that starts with any positive integer. The process involves two steps: if the number is even, divide it by 2; if it's odd, multiply it by 3 and add 1. Repeat this process with the resulting number. The conjecture suggests that, no matter what number you start with, you'll eventually reach the number 1.

In [178]:

```
# Solution goes here
```

Solution

In [179]:

```python
def collatz(n):
    # Initialize an empty list 'sequence' to store the numbers in the Collatz seq
    sequence = []

    # Continue looping until 'n' becomes 1
    while n != 1:
        # If 'n' is even, divide it by 2
        if n % 2 == 0:
            n = n // 2
        # If 'n' is odd, update 'n' to 3*n + 1
        else:
            n = 3 * n + 1
        # Append the new value of 'n' to the sequence
        sequence.append(n)

    # Return the generated Collatz sequence
    return sequence
```

Test your code

Run this code to test your solution:

In [180]:

```
try: assert collatz(12) == [6,3,10,5,16,8,4,2,1] and collatz(1) == [] and collatz
except: print('Test failed')
```

```
Test passed
```

Exercise 10: Greatest Common Divisor (GCD)

Define a function `gcd` that:

- takes as argument two integer numbers `a` and `b`
- returns:
  - the gcd between `a` and `b`

In [181]:

```
# Solution goes here
```

Solution

In [182]:

```
def gcd(x, y):
    # Continue looping until 'y' becomes 0
    while y != 0:
        # Update 'x' to 'y' and 'y' to the remainder of 'x' divided by 'y'
        # This is the core step of the Euclidean algorithm
        (x, y) = (y, x % y)

    # When 'y' becomes 0, 'x' will hold the greatest common divisor (GCD)
    return x
```

Test your code

Run this code to test your solution:

In [183]:

```
try: assert gcd(1,2) == 1 and gcd(7,2) == 1 and gcd(4,2) == 2 and gcd(15,25) == 5
except: print('Test failed')
```

```
Test passed
```

Exercise 11: Factorial of a number

Define a function `factorial` that:

- takes as argument two integer numbers `a`
- returns:
  - the factorial of `a` (i.e., `n! = n * (n-1) * (n-2) * ... * 1`)

In [184]:

```
# Solution goes here
```

Solution

In [185]:

```python
def factorial(x):
    # Initialize 'result' to 1, which will store the product of numbers
    result = 1

    # Loop from 0 to 'x-1', multiplying 'result' by 'x-i' in each iteration
    for i in range(0, x):
        result *= x - i  # Multiply 'result' by the decreasing value 'x-i'

    # Return the final computed factorial value
    return result
```

Test your code

Run this code to test your solution:

In [186]:

```
try: assert factorial(1) == 1 and factorial(0) == 1 and factorial(5) == 120 and n
except: print('Test failed')
```

```
Test passed
```

Exercise 12: Count vowels in a string

Define a function `vowels_counter` that:

- takes as argument a string `a`
- returns:
  - a dictionary with as key each vowel and as values the occurrencies of each vowel

In [187]:

```
# Solution goes here
```

Solution

In [188]:

```
def vowels_counter(x):
    # Initialize an empty dictionary 'result' to store the count of each vowel
    result = {}

    # Create a list 'vowels' containing the vowel characters to check against
    vowels = ["a", "e", "i", "o", "u"]

    # Iterate over each character 'i' in the input string 'x'
    for i in x:
        # Check if the character 'i' is a vowel
        if i in vowels:
            # If the vowel is already in 'result', increment its count by 1
            if i in result:
                result[i] = result[i] + 1
            # If the vowel is not in 'result', add it with an initial count of 1
            else:
                result[i] = 1

    # Return the dictionary 'result', which contains the counts of vowels in the
    return result
```

Test your code

Run this code to test your solution:

In [189]:

```
try: assert vowels_counter("ciao") == {'i': 1, 'a': 1, 'o': 1} and vowels_counter
except: print('Test failed')
```

```
Test passed
```

Exercise 13: Find missing number in a sequence

Define a function `find_missing` that:

- Takes a list of `n-1` integers, which represents a sequence of numbers from 1 to `n`, but one number is missing.
- Returns the missing number.

NOTE: Suppose that there is always only one number missing

Example:

- `find_missing([1, 2, 4, 5])` should return `3`.
- `find_missing([2, 3, 4, 6, 1])` should return `5`.

In [190]:

```
# Solution goes here
```

Solution

```
In [191]: def find_missing(nums):
              # Calculate the expected length of the complete list, which includes one miss
              n = len(nums) + 1

              # Calculate the total sum of the first 'n' natural numbers using the formula
              total_sum = n * (n + 1) // 2

              # Calculate the actual sum of the numbers present in the input list 'nums'
              actual_sum = sum(nums)

              # The missing number is the difference between the total sum and the actual s
              return total_sum - actual_sum
```

Test your code

Run this code to test your solution:

In [192]:

```
try: assert find_missing([1, 2, 4, 5]) == 3 and find_missing([2, 3, 4, 6, 1]) ==
except: print('Test failed')
```

```
Test passed
```

Exercise 14: Longest Substring Without Repeating Characters

Define a function `longest_unique_substring` that:

- Takes a string as input.
- Returns the length of the longest substring that contains only unique characters.

Example:

- `longest_unique_substring("abcabcbb")` should return `3` (substring "abc").
- `longest_unique_substring("bbbbb")` should return `1`.

In [193]:

```
# Solution goes here
```

Solution

In [194]:

```python
def longest_unique_substring(s):
    # Initialize 'max_length' to store the length of the longest unique substring
    max_length = 0

    # Outer loop iterates over each character in the string 's'
    for i in range(len(s)):
        # Initialize an empty set 'seen_chars' to keep track of characters in the
        seen_chars = set()

        # Inner loop starts from index 'i' and iterates through the string
        for j in range(i, len(s)):
            # If the current character 's[j]' is already in 'seen_chars', break t
            if s[j] in seen_chars:
                break
            # Add the current character 's[j]' to the set of seen characters
            seen_chars.add(s[j])

        # Update 'max_length' with the maximum value between the current 'max_len
        max_length = max(max_length, j - i)

    return max_length
```

Test your code

Run this code to test your solution:

In [195]:

```
try: assert longest_unique_substring("abcabcbb") == 3 and longest_unique_substrin
except: print('Test failed')
```

```
Test passed
```

Exercise 15: Find the Majority Element

Define a function `majority_element` that:

- Takes a list of integers as input.
- Returns the element that appears more than half of the time in the list (if it exists). If no such element exists, return `None`.

Example:

- `majority_element([3, 3, 4, 2, 3, 3, 5])` should return `3`.
- `majority_element([1, 2, 3, 4, 5])` should return `None`.

In [196]:

```
# Solution goes here
```

Solution

In [197]:

```
def majority_element(nums):
    # Initialize an empty dictionary 'count' to store the frequency of each numbe
    count = {}

    # Get the length of the input list 'nums'
    n = len(nums)

    # Iterate over each number in the list 'nums'
    for num in nums:
        # If the number 'num' is already in the count dictionary, increment its c
        if num in count:
            count[num] = count[num] + 1
        # If the number 'num' is not in the count dictionary, add it with a count
        else:
            count[num] = 1

        # Check if the count of the current number exceeds half of the list lengt
        if count[num] > n // 2:
            # If so, return the current number as the majority element
            return num

    # If no majority element is found, return None
    return None
```

Test your code

Run this code to test your solution:

In [198]:

```
try: assert majority_element([3, 3, 4, 2, 3, 3, 5]) == 3 and majority_element([1,
except: print('Test failed')
```

```
Test passed
```

Exercise 16: Check Palindrome

Define a function `is_palindrome` that:

- Takes a string as input.
- Returns `True` if the string is a palindrome, and `False` otherwise.

NOTE: A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward (ignoring spaces, punctuation, and capitalization).

Example:

- `is_palindrome("racecar")` should return `True`.
- `is_palindrome("hello")` should return `False`.
- `is_palindrome("A man, a plan, a canal, Panama")` should return `True`.

In [199]:

```
# Solution goes here
```

Solution

In [200]:

```
def is_palindrome(s):
    # Initialize an empty string 's2' to hold the filtered and normalized charact
    s2 = ""

    # Iterate over each character in the input string 's'
    for i in range(len(s)):
        # Check if the character is an alphabetic character
        if s[i].isalpha():
            # Convert the character to lowercase and add it to 's2'
            s2 += s[i].lower()

    # Check if the filtered string is a palindrome by comparing characters
    for i in range(0, int(len(s2) / 2)):
        # If characters at symmetric positions do not match, it's not a palindrom
        if s2[i] != s2[len(s2) - i - 1]:
            return False

    # If all symmetric characters match, return True indicating it's a palindrome
    return True
```

Test your code

Run this code to test your solution:

In [201]:

```
try: assert is_palindrome("racecar") and not is_palindrome("hello") and is_palind
except: print('Test failed')
```

```
Test passed
```

Exercise 17: Implement ROT13 Cipher

Define a function `rot13` that:

- Takes a string as input.
- Returns a new string where each letter is replaced by the letter 13 positions after it in the alphabet. If the shift passes the end of the alphabet, it wraps around to the beginning. Non-alphabetic characters should remain unchanged.

NOTE:

- ROT13 is a special case of the Caesar cipher, which is a simple substitution cipher where each letter in the plaintext is shifted a certain number of places down or up the alphabet. In the case of ROT13, the shift is 13 places.
- Consider an alphabet of 26 letters: `"abcdefghijklmnopqrstuvwxyz"`
- `ord(c)` returns an integer representing `c`.
- `chr(x)` returns the character associated with integer `x`.

Example:

- `rot13("hello")` should return `"uryyb"`.
- `rot13("uryyb")` should return `"hello"`.
- `rot13("hello, world!")` should return `"uryyb, jbeyq!"`.

In [202]:

```
# Solution goes here
```

Solution

```
In [203]: def rot13(s):
              # Initialize an empty list 'result' to store the transformed characters
              result = []

              # Iterate over each character in the input string 's'
              for char in s:
                  # Check if the character is a lowercase letter
                  if 'a' <= char <= 'z':
                      # Apply ROT13 transformation: shift character by 13 positions within
                      result.append(chr((ord(char) - ord('a') + 13) % 26 + ord('a')))
                  # Check if the character is an uppercase letter
                  elif 'A' <= char <= 'Z':
                      # Apply ROT13 transformation: shift character by 13 positions within
                      result.append(chr((ord(char) - ord('A') + 13) % 26 + ord('A')))
                  # If the character is neither lowercase nor uppercase, keep it unchanged
                  else:
                      result.append(char)

              # Join the list of transformed characters into a single string and return it
              return ''.join(result)
```

Test your code

Run this code to test your solution:

In [204]:

```
try: assert rot13("hello") == "uryyb" and rot13("uryyb") == "hello" and rot13("he
except: print('Test failed')
```

```
Test passed
```