Lesson 1: Week 4

Created: 2024-05-27T01:25:47.394393+10:00

Exercise 1: Outlier remover

When you calculate the mean of a collection of numbers the result can be adversely affected by outliers - values that are extreme, either extremely small or extremely large.Write a program in which you define a function that takes a list of numbers as argument and returns their mean, but when it calculates the mean it ignores the smallest number and the largest number. Give your function an informative name.Illustrate the use of the function by applying it to some example lists.Some things to be careful of:Make sure your function doesn't change the list that it receives.Make sure your function can handle lists that have fewer than 3 elements.Make sure your function can handle lists that contain non-numeric values.

Exercise 2: Memoizer

Suppose you have a function that takes a value and returns a value but the calculation it uses consumes a lot of time and resources.If the function always returns the same value when given the same value, then it can be a good idea to get the function to remember the results of its calculations. Then, if the function is passed a value that it has already had, it can simply recall the result of the calculation rather than doing the calculation again. This technique is called memoization.Write a program in which you define a function that finds the cube of a number. Get it to use memoization. Memoization is not really needed in this case, because the calculation is not time or resource intensive, but it will illustrate the principles.Here is one way to proceed:Define your function as normalAdd an attribute to the function, whose value is a dictionary.When the function is called with a number it can check this dictionary to see if it has a result saved. If it does, it can return this saved result. Otherwise, it can calculate the result, add it to the dictionary, and return the result.

Exercise 3: List mapper

Write a program in which you define a function map(), which takes a function and a list and applies the function to each element of the list, returning the results as a list.Examples:x = map(abs, [-2, 4, -6, -8]) print(x) [2, 4, 6, 8]

Exercise 4: Roman numerals

Write a program that prints out the decimal value of a Roman numeral.Your program should accept the Roman numeral from the command line arguments. Click on Terminal to activate the terminal. You may assume the Roman numeral is in the "standard" form, i.e., any digits involving 4 and 9 will always appear in the subtractive form.Sample interactionspython roman_numerals.py II 2python roman_numerals.py IV 4python roman_numerals.py IX 9python roman_numerals.py XIX 19python roman_numerals.py XX 20python roman_numerals.py MDCCLXXVI 1776python roman_numerals.py MMXIX 2019Hints:Use a loop to iterate through the Roman numeral to figure out their value.Use a list of tuples to store the string characters and their respective values. Compare the characters from the input to this list.Use a while loop so you can manually control the indices.

Exercise 5: Finding particular sequences of prime numbers

Insert your code into consecutive_primes.py to find all sequences of 6 consecutive prime 5-digit numbers, say (a, b, c, d, e, f), with $b = a + 2$, $c = b + 4$, $d = c + 6$, $e = d + 8$, and $f = e + 10$.a, b, c, d, e, and f are therefore all 5-digit prime numbers and no number between a and b, between b and c, between c and d, between d and e, and between e and f is prime.If you are stuck, but only when you are stuck, then use consecutive_primes_scaffold.py.

Exercise 6: Special products

Insert your code into special_products.py to find all triples of positive integers (i, j, k) such that i, j, and k are two digit numbers, no digit occurs more than once in i, j, and k, and the set of digits that occur in i, j, or k is equal to the set of digits that occur in the product of i, j, and k.If you are stuck, but only when you are stuck, then use special_products_scaffold_1.py. If you are still stuck, but

only when you are still stuck, then use special_products_scaffold_2.py.

Exercise 7: Finding particular sequences of triples of the form (n, n+1, n+2)

Write a program called special_triples.py that finds all triples of consecutive positive three-digit integers each of which is the sum of two squares, that is, all triples of the form $(n,n+1,n+2)$ such that:n, $n+1$ and $n+2$ are integers at least equal to 100 and at most equal to 999;each of n, $n+1$ and $n+2$ is of the form a^2+b^2.Hint: As we are not constrained by memory space for this problem, we might use a list that stores an integer for all indexes n in $[100,999]$, equal to 1 in case n is the sum of two squares, and to 0 otherwise. Then it is just a matter of finding three consecutive 1's in the list. This idea can be refined (by not storing 1s, but suitable nonzero values) to not only know that some number is of the form a^2+b^2, but also know such a pair (a,b).If an integer n is of the form a^2+b^2, then the decomposition is not necessarily unique. We want each decomposition that is output to be the minimal one w.r.t. the natural ordering of pairs of integers (that is, $(a,b)If you are stuck, but only when you are stuck, then use special_triples_scaffold.py.

Exercise 8: Number of trailing 0s in a factorial

To illustrate, 15!, the factorial of 15, is equal to 1307674368000, hence has 3 trailing 0s.There are at least three methods to compute the number of trailing 0s in the factorial of a number N at least equal to 5:Divide N! by 10 for as long as it yields no remainder. Note that for a positive integer x, x // 10 "removes" the rightmost digit from x, that digit being equal to x % 10.Convert N! into a string and find the rightmost occurrence of a character different to 0. A Google search, or executing dir(str) at the python prompt, suggests which string method to use. Note that negative indexes (-1 being the index of the last character in a string, -2 the index of the penultimate character in a string, etc.) is particularly convenient here.Python computes such huge numbers as 1000!, either iteratively multiplying all numbers from 1 up to 1000 or using factorial() from the math module (executing import math and then dir(math) at the python prompt confirms that this function is available), and the first two methods work for such numbers, but there is a much better method that operates on N rather N!, hence that does not suffer the limitations of the first two, and is very efficient. The number of trailing 0s in N! is equal to the number of times N! is a multiple of 10, so to the number of times N! is a multiple of 2 x 5. It is easy to verify that N! has at least as many multiples of 2 as multiples of 5. Hence the number of trailing 0s in N!is equal to the number of times N! is a multiple of 5which is equal to the number of times 5 occurs in the prime decompositions of 1, 2, ..., N-1 and Nwhich is equal to the number of times 5 occurs at least once in the prime decompositions of 1, 2, ..., N-1 and N, plus the number of times 5 occurs at least twice in the prime decompositions of 1, 2, ..., N-1 and N, plus the number of times 5 occurs at least thrice in the prime decompositions of 1, 2, ..., N-1 and N...which is equal to the number of multiples of 5 at most equal to N, plus the number of multiples of 5^2 at most equal to N, plus the number of multiples of 5^3 at most equal to N ...Insert your code into trailing_0s_in_factorials.py so that the program prompts the user for a non-negative integer N. If the input is incorrect then the program outputs an error message and exits. Otherwise the program computes 5! three times, using the three methods just described. See sample outputs for details on input and ouput. If you are stuck, but only when you are stuck, then use trailing_0s_in_factorial_scaffold_1.py. If you are still stuck, but only when you are still stuck, then use trailing_0s_in_factorial_scaffold_2.py.