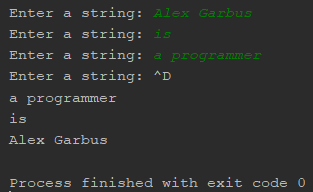
**Program 1**

Example:



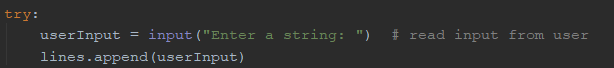
2 initial variables are established: a Boolean for checking whether input is complete and a list for collecting the input strings.



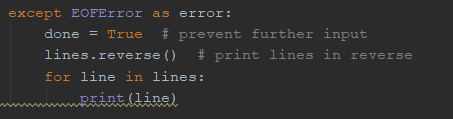
A loop is used to collect repeated user input.



Each time the loop is run, the program asks for input and adds it to the *lines* list.

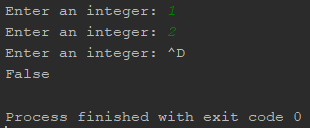
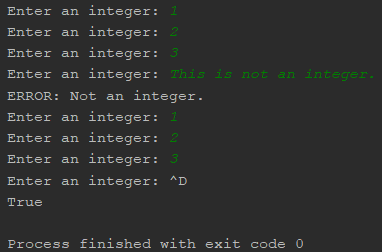


When end-of-input is reached, *done* is set to True in order to prevent the loop from running again. The *reverse()* function reverses the items of the *lines* list, and then a for loop prints each item of the new reversed list.

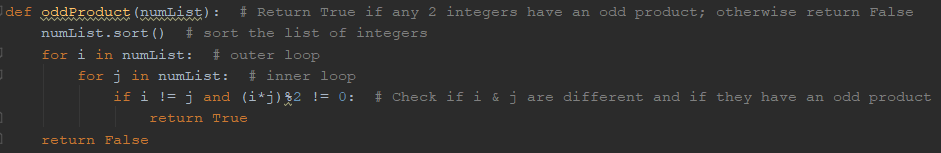


**Program 2**

Examples:



The *oddProduct()* method takes a list *numList* as a parameter. The items in the list are sorted from least to greatest. Then a nested for loop is used to compare the items of the list with each other. If any 2 numbers are different and have an odd product, the method immediately returns True. If it reaches the end of the loop, it returns False.



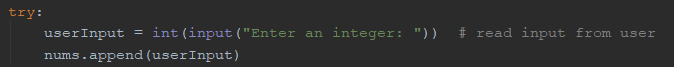
2 initial variables are established, a list for storing input and a Boolean for checking whether input has completed.



A loop is used to collect repeated user input.



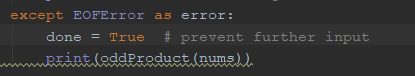
Each time the loop is run, it asks the user to input an integer. If an integer is successfully input, it is added to the *nums* list.



If any value other than an integer is entered, an error message is printed. This ensures that errors won’t be encountered when calling *oddProduct()*.

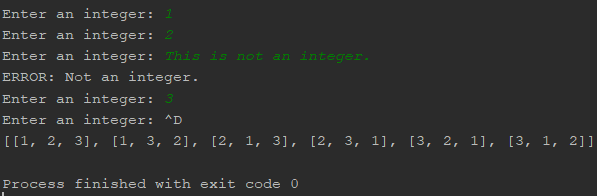


When end-of-input is reached, *done* is set to True in order to prevent the loop from running again. Then *oddProduct()* is called with *nums* as its parameter, and the result is printed.



**Program 3**

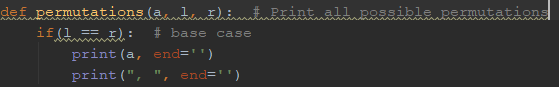
Example:



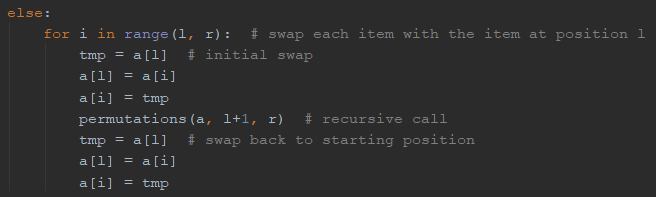
The program imports *sys* to deal with output (seen later in the program).



The *permutations()* method takes in *a, l, and r*. *a* should be the list of input integers. *l* and *r* should be the left and right boundaries of the array. The method is recursive. When the left and right positions are the same, the base case, the current permutation is printed, followed by a comma for organization.



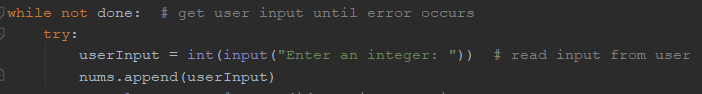
Otherwise, the method goes through the active portion of the list using a for loop. It first swaps the items at positions *l* and *i*. It then calls itself recursively, incrementing the left position by 1. After the recursive call reaches its base case (when the permutation is printed), the initial swap is reversed.



The program establishes 2 variables: a list for storing the input integers and a Boolean for determining when input is complete.



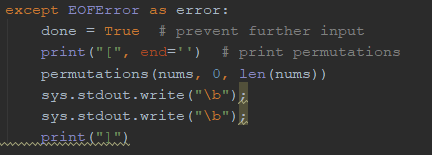
A loop is used to collect repeated user input. Each time the loop is run, it asks the user to input an integer. If an integer is successfully input, it is added to the *nums* list.



If any value other than an integer is entered, an error message is printed. This ensures that errors won’t be encountered later.

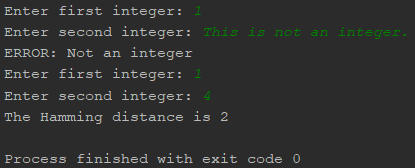


When end-of-input is reached, *done* is set to True in order to prevent the loop from running again. Then the permutations are printed in the form *[[permutation 1], [permutation 2], … [permutation len(nums)].* The permutations themselves are printed using *permutations()*, but the surrounding brackets are printed outside of the method, with *sys.stdout.write(“\b”)* being used to delete the extra comma at the end of the output.

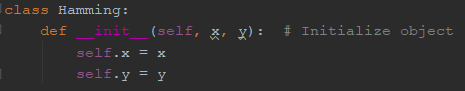


**Program 4**

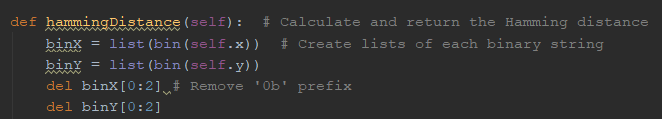
Examples:



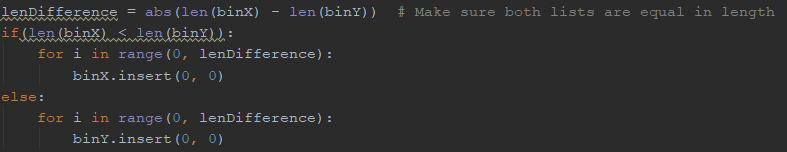
|  |  |
| --- | --- |
| **Class** | Hamming |
| **Fields** | x, y |
| **Behaviors** | hammingDistance() |



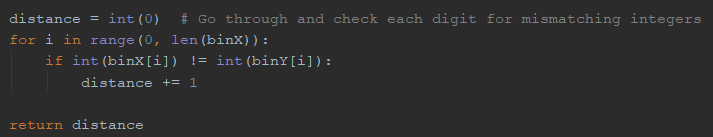
The *hammingDistance()* method first converts *x* and *y* to binary strings, with each character becoming an item of a list. Because “0b” gets added to the beginning of strings when converted to binary, these 2 characters must be deleted through code.



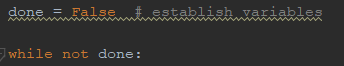
Before calculating the Hamming difference, both strings must be equal in length. If one of the 2 strings are found to be shorter than the other, then a “0” will be added to the beginning of the shorter string. This is repeated until both strings are equal.



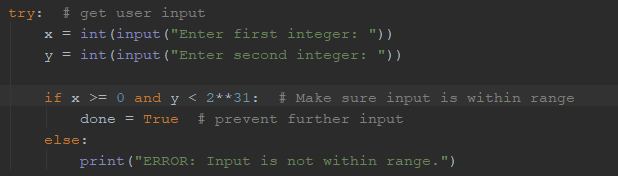
Once the strings are equal, a for loop is used to compare them. If any 2 characters are different, *distance* is incremented by 1. Once the loop has completed, *distance* is returned.



A while loop is used for gathering input. It is controlled by a *Boolean.*



Each time the loop runs, the user is asked for 2 integers. The first integer must be at least 0, and the second integer must be less than 231. If the integers do meet these requirements, input ends. Otherwise, an error message is printed.



If any value other than an integer is entered, an error message is printed.



A new *Hamming* object is constructed, with its *x* and *y* fields being set to the 2 input integers. The Hamming distance is then calculated and printed.

