#### Snippet 3

*# Pull request 3*

def check\_for\_val(self, val):

*"""This member function checks to see if val exists in the class member*

*values and returns True if found"""*

found = False

for i in range(len(self.values)):

if self.values[i] == val:

found = True

return found

Feedback:

The method above is very clean, easily understood and well documented with a descriptive doc string. Nicely, done using self.values to access the attributes within a class, I find myself frequently forgetting the ‘self’ prefix when dealing with classes. With that said, I am assuming that self.values is an attribute of the class being used. It would help to see the entire ‘\_\_init\_\_’ method

to ensure the class is constructed properly and that self.values is not a typo and implemented correctly.

If you were looking to speed up the implementation used, you could add a ‘break’ after ‘found = True’. In the event the target ‘val’ was located earlier in self.values members (best case scenario), this would return True quicker, eliminating the need for each value to be checked and making the code faster.

Snippet 4

*# Pull request 4*

def get\_val\_index(arr, val):

*"""Searches arr for val and returns the index if found, otherwise -1"""*

index = -1

for i in range(len(arr)):

if arr[i] == val:

index = i

break

return index

Feedback: The code above works great and successfully accomplished the intended task as described in the docstring. Although is there a built-in python method which could be used to make the code appear cleaner and more concise? The answer is yes! When working with lists, .index() can be used to return the index location of a given value. Combining .index() with a simple try/except method allows for handling of ValueErrors which might occur when a value is not present within the array, returning -1 in this case. The method above is very simple to follow and works wonderfully, but if you are looking to limit code space, check out the alternative method below.

# Snippet 4 - revised code: use the .index() built-in

def get\_val\_index(arr, val):

"""Searches arr for val and returns the index if found, otherwise -1"""

try: return arr.index(val)

except ValueError: return -1

print(get\_val\_index([99,11,22,-1,6,5], -261561))

# -------------------------------------------------------------------------------

*# Pull request 5*

def double\_array(arr):

*"""sort the list (ascending) and double the value of each element of*

*the list and return without changing the state of the original list"""*

arr.sort()

for i in range(len(arr)):

arr[i] = arr[i] \* 2

return arr

Feedback: The code function above is well written, easily readable, and properly documented. Although, one error I did find is regarding the ‘*return without changing the state of the original list*’ statement and use of sort(). Under its current implementation this requirement is not met as it appears the state of the original list is modified once returned. The use of sort() will permanently modify the original list. But, did you know there is a builit-in python method, similar to sort(), which can return a copied array, leaving the originally passed ‘arr’ unchanged? I’ve found .sorted() method works well here. It creates and returns an ascendingly sorted list, while allowing the state of the original array to remain untouched.

Combining list comprehension with the newly created sorted() list allows for each value within the list to be doubled and provides a shorted version of the code above. An added bonus to using .sorted() is the optional parameter ‘reverse = True’ which can return a sorted list in descending order, if you ever wanted to use said functionality. Check out the modified version of ‘double\_array’ below.

# snippet 5 - revised code: use list comprehension + sorted() fx to return copy of arr

# Pull request 5

def double\_array(arr):

"""sort the list (ascending) and double the value of each element of

return sorted(val\*2 for val in arr)