**Python Notes**

1. i = bisect.bisect\_left( list, num ) The returned insertion point i partitions the array a into two halves so that all(val < x for val in a[lo : i]) for the left side and all(val >= x for val in a[i : hi]) for the right side. => Basically it finds the first rightful position in the array .
2. bisect\_right: The returned insertion point i partitions the array a into two halves so that all(val <= x for val in a[lo : i]) for the left side and all(val > x for val in a[i : hi]) for the right side. Basically it finds the last rightful position of the element in the array in case of a tie.
3. Python generators

|  |
| --- |
| def flatten\_list(nested\_list):  for nested\_integer in nested\_list:  if nested\_integer.isInteger():  yield nested\_integer.getInteger()  else:  for integer in flatten\_list(nested\_integer.getList()):  yield integer  def flatten\_list(nested\_list):  for nested\_integer in nested\_list:  if nested\_integer.isInteger():  yield nested\_integer.getInteger()  else:  yield from flatten\_list(nested\_integer.getList())  class NestedIterator:  def \_\_init\_\_(self, nestedList: [NestedInteger]):  # Get a generator object from the generator function, passing in  # nestedList as the parameter.  self.\_generator = self.\_int\_generator(nestedList)  # All values are placed here before being returned.  self.\_peeked = None  # This is the generator function. It can be used to create generator  # objects.  def \_int\_generator(self, nested\_list) -> "Generator[int]":  # This code is the same as Approach 1. It's a recursive DFS.  for nested in nested\_list:  if nested.isInteger():  yield nested.getInteger()  else:  # We always use "yield from" on recursive generator calls.  yield from self.\_int\_generator(nested.getList())  # Will automatically raise a StopIteration.    def next(self) -> int:  # Check there are integers left, and if so, then this will  # also put one into self.\_peeked.  if not self.hasNext(): return None  # Return the value of self.\_peeked, also clearing it.  next\_integer, self.\_peeked = self.\_peeked, None  return next\_integer    def hasNext(self) -> bool:  if self.\_peeked is not None: return True  try: # Get another integer out of the generator.  self.\_peeked = **next(self.\_generator)**  return True  **except: # The generator is finished so raised StopIteration.**  return False |

1. OrderedDict, move\_to\_end(key), popitem(last=False)
2. Custom Exceptions

|  |
| --- |
| 1. class ValidationError(Exception): 2. def \_\_init\_\_(self, message, errors): 3. # Call the base class constructor with the parameters it needs 4. super().\_\_init\_\_(message) 6. # Now for your custom code... 7. self.errors = errors   You could also override the \_\_str\_\_ method to print the appropriate error string |