```
# First Elixir Homework
#
# Rodrigo Benavente García
# 04/10/21
#


defmodule Lists do
  #First exercise, we recieve a list, and two elements A and B -------------
-------
  def swapper(list, first, second), do: do_swapper(list, first, second,[])
  #When the list is empty, return result
  defp do_swapper([], first, second, result),
   do: result
  #Recursion
  #Change the first element for the second element
  defp do_swapper([head | tail], first, second, result) when head == first,
    do: do_swapper(tail, first, second, result ++ [second])
  #Change the second element for the first element
  defp do_swapper([head | tail], first, second, result) when head == second,
    do: do_swapper(tail, first, second, result ++ [first])
  #Don't change any element for anything
  defp do_swapper([head | tail], first, second, result),
    do: do_swapper(tail, first, second, result ++ [head])


  #Second exercise, we take a list of tuples and inverse their orders ------
---------------------------
  def invert_pairs(list), do: do_invert_pairs(list, [])
  #Function that inverts tuple
  defp invert_tuple({a,b}), do: {b,a}
  #When the list is empty, return result
  defp do_invert_pairs([], result),
   do: result
  defp do_invert_pairs([head | tail], result), do:
    do_invert_pairs(tail, result ++ [invert_tuple(head)])

  #Third exercise, lists as arguments --------------------------------------
----------------------
  def deep_reverse(list), do: do_deep_reverse(list, [])

  defp do_deep_reverse([], result),#Return the result when the list is empty
    do: result
  defp do_deep_reverse([head | tail], result) when is_list(head),#Reverse
the elements of a nested list
    do: do_deep_reverse(tail, [deep_reverse(head) | result])

  defp do_deep_reverse([head | tail], result), #Reverse the elements of a
list
    do: do_deep_reverse(tail, [head | result])

  #Fourth exercise, we take a list of numbers as arguments -----------------
------------------
  def mean(list), do: do_mean(list, 0, 0)
  #If the list starts empty
  defp do_mean([], 0, 0),
    do: 0
  # End of list case
  defp do_mean([], result, count),
    do: result/count#Returns the mean
  # Recursion
  defp do_mean([head | tail], result, count),
    do: do_mean(tail, head + result, count + 1)#Add +1 to the counter of
elements

  #Fifth exercise, we take a list of numbers as arguments ------------------
--------------------------
  # Add all the elements (numbers) in a list
  def sum([]), do: 0
  def sum([head | tail]), do: head + sum(tail)

  #Base Case
  def std_dev([]), do: 0
  def std_dev(list) do
    numE = Enum.count(list)#Get number of elements
    mean = mean(list)#Get the mean
    total = :math.sqrt(sum(Enum.map(list, fn x -> (x - mean) * (x - mean)
end))/numE)
    total#Return the total
  end

end #End of my module
```