########################################################################################################################  
#add\_5\_higher\_order  
  
def function(array\_data):  
 return list(map(lambda x: x + 5, array\_data))  
  
  
print(function([0, 5, 10]))  
  
  
  
########################################################################################################################  
#add\_5\_iterative  
  
def function(array\_data):  
 result = []  
 for element in array\_data:  
 result.append(element + 5)  
 return result  
  
  
print(function([0, 5, 10]))  
  
  
  
########################################################################################################################  
#add\_5\_list\_comprehension  
  
def function(array\_data):  
 return [data + 5 for data in array\_data]  
  
  
print(function([0, 5, 10]))  
  
  
  
########################################################################################################################  
#add\_5\_recursive  
  
def function(array\_data):  
 if len(array\_data) == 0:  
 return []  
 return [array\_data[0] + 5] + function(array\_data[1:])  
  
  
print(function([0, 5, 10]))  
  
  
  
########################################################################################################################  
#apply\_higher\_order  
  
def function(array\_data, func):  
 return list(map(func, array\_data))  
  
  
print(function([1, 2, 3], lambda x: x \*\* 2 + x))  
  
  
  
########################################################################################################################  
#apply\_iterative  
  
def function(array\_data, func):  
 results = []  
 for data in array\_data:  
 results.append(func(data))  
 return results  
  
  
print(function([1, 2, 3], lambda x: x \*\* 2 + x))  
  
  
  
########################################################################################################################  
#apply\_list\_comprehension  
  
def function(array\_data, func):  
 return [func(data) for data in array\_data]  
  
  
print(function([1, 2, 3], lambda x: x \*\* 2 + x))  
  
  
  
########################################################################################################################  
#apply\_recursive  
  
def function(array\_data, func):  
 if len(array\_data) == 0:  
 return []  
 else:  
 return [func(array\_data[0])] + function(array\_data[1:], func)  
  
  
print(function([1, 2, 3], lambda x: x \*\* 2 + x))  
  
  
  
########################################################################################################################  
#Computer\_higher\_order  
  
class Computer:  
 id = 0  
  
 def \_\_init\_\_(self, cpu, gpu, ram):  
 self.cpu = cpu  
 self.gpu = gpu  
 self.ram = ram  
 self.id = Computer.id  
 Computer.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
def function(computer\_array):  
 return list(filter(lambda x: "AMD" in x.cpu and "NVIDIA GTX30" in x.gpu and x.ram >= 16, computer\_array))  
  
  
computer\_array = []  
computer\_array.append(Computer("INTEL i7-860", "NVIDIA GTX3080", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3080", 32))  
computer\_array.append(Computer("INTEL i9-10900T", "NVIDIA GTX1070", 8))  
computer\_array.append(Computer("AMD 5900x", "AMD RX6900", 8))  
computer\_array.append(Computer("AMD 5700", "AMD RX6900", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3090", 64))  
computer\_array.append(Computer("INTEL i5-8400", "NVIDIA GTX1060", 4))  
print(function(computer\_array))  
  
  
  
########################################################################################################################  
#Computer\_iterative  
  
class Computer:  
 id = 0  
  
 def \_\_init\_\_(self, cpu, gpu, ram):  
 self.cpu = cpu  
 self.gpu = gpu  
 self.ram = ram  
 self.id = Computer.id  
 Computer.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
def function(computer\_array):  
 results = []  
 for computer in computer\_array:  
 if "AMD" in computer.cpu and "NVIDIA GTX30" in computer.gpu and computer.ram >= 16:  
 results.append(computer)  
 return results  
  
  
computer\_array = []  
computer\_array.append(Computer("INTEL i7-860", "NVIDIA GTX3080", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3080", 32))  
computer\_array.append(Computer("INTEL i9-10900T", "NVIDIA GTX1070", 8))  
computer\_array.append(Computer("AMD 5900x", "AMD RX6900", 8))  
computer\_array.append(Computer("AMD 5700", "AMD RX6900", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3090", 64))  
computer\_array.append(Computer("INTEL i5-8400", "NVIDIA GTX1060", 4))  
print(function(computer\_array))  
  
  
  
########################################################################################################################  
#Computer\_list\_comprehension  
  
class Computer:  
 id = 0  
  
 def \_\_init\_\_(self, cpu, gpu, ram):  
 self.cpu = cpu  
 self.gpu = gpu  
 self.ram = ram  
 self.id = Computer.id  
 Computer.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
def function(computer\_array):  
 return [computer for computer in computer\_array if "AMD" in computer.cpu and "NVIDIA GTX30" in computer.gpu and computer.ram >= 16]  
  
  
computer\_array = []  
computer\_array.append(Computer("INTEL i7-860", "NVIDIA GTX3080", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3080", 32))  
computer\_array.append(Computer("INTEL i9-10900T", "NVIDIA GTX1070", 8))  
computer\_array.append(Computer("AMD 5900x", "AMD RX6900", 8))  
computer\_array.append(Computer("AMD 5700", "AMD RX6900", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3090", 64))  
computer\_array.append(Computer("INTEL i5-8400", "NVIDIA GTX1060", 4))  
print(function(computer\_array))  
  
  
  
########################################################################################################################  
#Computer\_recursive  
  
class Computer:  
 id = 0  
  
 def \_\_init\_\_(self, cpu, gpu, ram):  
 self.cpu = cpu  
 self.gpu = gpu  
 self.ram = ram  
 self.id = Computer.id  
 Computer.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
def function(computer\_array):  
 if len(computer\_array)==0:  
 return []  
 if "AMD" in computer\_array[0].cpu and "NVIDIA GTX30" in computer\_array[0].gpu and computer\_array[0].ram >= 16:  
 return [computer\_array[0]] + function(computer\_array[1:])  
 else:  
 return function(computer\_array[1:])  
  
  
computer\_array = []  
computer\_array.append(Computer("INTEL i7-860", "NVIDIA GTX3080", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3080", 32))  
computer\_array.append(Computer("INTEL i9-10900T", "NVIDIA GTX1070", 8))  
computer\_array.append(Computer("AMD 5900x", "AMD RX6900", 8))  
computer\_array.append(Computer("AMD 5700", "AMD RX6900", 16))  
computer\_array.append(Computer("AMD 5900x", "NVIDIA GTX3090", 64))  
computer\_array.append(Computer("INTEL i5-8400", "NVIDIA GTX1060", 4))  
print(function(computer\_array))  
  
  
  
########################################################################################################################  
#condition\_sum\_higher\_order  
  
from functools import reduce  
  
  
def function(n):  
 array\_data = filter(lambda x: x % 3 == 0 or x % 4 == 0, range(2, n + 1))  
 return reduce(lambda x, y: x + y, array\_data)  
  
  
print(function(14))  
  
  
  
########################################################################################################################  
#condition\_sum\_iterative  
  
def function(n):  
 array\_data = []  
 for i in range(2, n + 1):  
 if i % 3 == 0 or i % 4 == 0:  
 array\_data.append(i)  
 result = 0  
 for value in array\_data:  
 result += value  
 return result  
  
  
print(function(14))  
  
  
  
########################################################################################################################  
#condition\_sum\_list\_comprehension  
  
from functools import reduce  
  
  
def function(n):  
 array\_data = [value for value in range(2, n + 1) if value % 3 == 0 or value % 4 == 0]  
 total = 0  
 scanned = [total := total + x for x in array\_data]  
 return scanned[-1]  
  
  
print(function(14))  
  
  
  
########################################################################################################################  
#condition\_sum\_recursive  
  
def function(n):  
 if n == 1:  
 return 0  
 if n % 3 == 0 or n % 4 == 0:  
 return n + function(n - 1)  
 else:  
 return function(n - 1)  
  
  
print(function(14))  
  
  
  
########################################################################################################################  
#find\_higher\_order  
  
from functools import reduce  
  
  
def function(data\_array, element):  
 index\_data\_array = enumerate(data\_array)  
 filtered\_array = map(lambda x\_tuple: x\_tuple[0] if x\_tuple[1] == element else 0, index\_data\_array)  
 return reduce(lambda x, y: x + y, filtered\_array)  
  
  
print(function([1, 2, 3, 4, 5], 3))  
  
  
  
########################################################################################################################  
#find\_iterative  
  
def function(data\_array, element):  
 for idx, value in enumerate(data\_array):  
 if value == element:  
 return idx  
 return -1  
  
  
print(function([1, 2, 3, 4, 5], 3))  
  
  
  
########################################################################################################################  
#find\_list\_comprehension  
  
from functools import reduce  
  
  
def function(data\_array, element):  
 return [idx for (idx, value) in enumerate(data\_array) if value == element][0]  
  
  
print(function([1, 2, 3, 4, 5], 3))  
  
  
  
########################################################################################################################  
#find\_recursive  
  
def function(data\_array, element):  
 if len(data\_array) == 0:  
 return -1  
 elif data\_array[0] == element:  
 return 0  
 else:  
 idx = function(data\_array[1:], element)  
 if idx == -1:  
 return -1  
 else:  
 return idx + 1  
  
  
print(function([1, 2, 3, 4, 5], 3))  
  
  
  
########################################################################################################################  
#is\_prime\_higher\_order  
  
def function(number):  
 number\_array = range(1, number + 1)  
 prime\_array = filter(lambda x: number % x == 0, number\_array)  
 return len(list(prime\_array)) == 2  
  
  
print(function(7))  
  
  
  
########################################################################################################################  
#is\_prime\_iterative  
  
def function(number):  
 if number == 1:  
 return False  
 for check\_num in range(2, int(number / 2) + 1):  
 if number % check\_num == 0:  
 return False  
 return True  
  
  
print(function(7))  
  
  
  
########################################################################################################################  
#is\_prime\_list\_comprehension  
  
def function(number):  
 return len([value for value in range(1, number+1) if number % value == 0]) == 2  
  
  
print(function(7))  
  
  
  
########################################################################################################################  
#is\_prime\_recursive  
  
def function(number, current=None):  
 if current is None:  
 current = int(number / 2)  
 if number == 1:  
 return False  
 if current == 1:  
 return True  
 if number % current == 0:  
 return False  
 return function(number, current - 1)  
  
  
print(function(7))  
  
  
  
########################################################################################################################  
#LinkedList\_higher\_order  
  
from functools import reduce  
from itertools import accumulate  
  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
 def \_\_repr\_\_(self):  
 return str(self.data)  
  
 def \_\_iter\_\_(self):  
 self.current = self  
 return self  
  
 def \_\_next\_\_(self):  
 if self.current is None:  
 raise StopIteration  
 else:  
 tmp = self.current  
 self.current = self.current.next  
 return tmp  
  
  
def function(node\_list):  
 filtered\_list = filter(lambda x: x[0] % 2 == 0, enumerate(node\_list))  
 mapped\_list = map(lambda x: x[1].data, filtered\_list)  
 return reduce(lambda x, y: x + y, mapped\_list)  
  
  
node1 = Node(2)  
node2 = Node(5)  
node3 = Node(7)  
node4 = Node(4)  
node5 = Node(1)  
node6 = Node(3)  
node7 = Node(6)  
node1.next = node2  
node2.next = node3  
node3.next = node4  
node4.next = node5  
node5.next = node6  
node6.next = node7  
print(function(node1))  
  
  
  
########################################################################################################################  
#LinkedList\_iterative  
  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
 def \_\_repr\_\_(self):  
 return str(self.data)  
  
 def \_\_iter\_\_(self):  
 self.current = self  
 return self  
  
 def \_\_next\_\_(self):  
 if self.current is None:  
 raise StopIteration  
 else:  
 tmp = self.current  
 self.current = self.current.next  
 return tmp  
  
  
def function(node\_list):  
 result = 0  
 for idx, node in enumerate(node\_list):  
 if idx % 2 == 0:  
 result += node.data  
 return result  
  
node1 = Node(2)  
node2 = Node(5)  
node3 = Node(7)  
node4 = Node(4)  
node5 = Node(1)  
node6 = Node(3)  
node7 = Node(6)  
node1.next = node2  
node2.next = node3  
node3.next = node4  
node4.next = node5  
node5.next = node6  
node6.next = node7  
print(function(node1))  
  
  
  
########################################################################################################################  
#LinkedList\_list\_comprehension  
  
from functools import reduce  
from itertools import accumulate  
  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
 def \_\_repr\_\_(self):  
 return str(self.data)  
  
 def \_\_iter\_\_(self):  
 self.current = self  
 return self  
  
 def \_\_next\_\_(self):  
 if self.current is None:  
 raise StopIteration  
 else:  
 tmp = self.current  
 self.current = self.current.next  
 return tmp  
  
  
def function(node\_list):  
 filtered\_list = [node.data for (idx, node) in enumerate(node\_list) if idx % 2 == 0]  
 total = 0  
 scanned = [total := total + x for x in filtered\_list]  
 return scanned[-1]  
  
  
node1 = Node(2)  
node2 = Node(5)  
node3 = Node(7)  
node4 = Node(4)  
node5 = Node(1)  
node6 = Node(3)  
node7 = Node(6)  
node1.next = node2  
node2.next = node3  
node3.next = node4  
node4.next = node5  
node5.next = node6  
node6.next = node7  
print(function(node1))  
  
  
  
########################################################################################################################  
#LinkedList\_recursive  
  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
 def \_\_repr\_\_(self):  
 return str(self.data)  
  
  
def function(node\_list, odd=False):  
 if node\_list is None:  
 return 0  
 if not odd:  
 return node\_list.data + function(node\_list.next, True)  
 else:  
 return function(node\_list.next, False)  
  
  
node1 = Node(2)  
node2 = Node(5)  
node3 = Node(7)  
node4 = Node(4)  
node5 = Node(1)  
node6 = Node(3)  
node7 = Node(6)  
node1.next = node2  
node2.next = node3  
node3.next = node4  
node4.next = node5  
node5.next = node6  
node6.next = node7  
print(function(node1))  
  
  
  
########################################################################################################################  
#max\_higher\_order  
  
from functools import reduce  
  
  
def function(array\_data):  
 return reduce(lambda x, y: x if x >= y else y, array\_data)  
  
  
print(function([0, 5, 2]))  
  
  
  
########################################################################################################################  
#max\_iterative  
  
def function(array\_data):  
 if len(array\_data) == 0:  
 return None  
  
 max\_value = array\_data[0]  
 for value in array\_data:  
 if value > max\_value:  
 max\_value = value  
 return max\_value  
  
  
print(function([0, 5, 2]))  
  
  
  
########################################################################################################################  
#max\_list\_comprehension  
  
from functools import reduce  
  
  
def function(array\_data):  
 max\_value = array\_data[0]  
 scanned = [max\_value := x for x in array\_data if x > max\_value]  
 return scanned[-1]  
  
  
print(function([0, 5, 2]))  
  
  
  
########################################################################################################################  
#max\_recursive  
  
def function(array\_data, current=None):  
 if len(array\_data) == 0:  
 return current  
 elif current is None:  
 return function(array\_data[1:], array\_data[0])  
 elif current >= array\_data[0]:  
 return function(array\_data[1:], current)  
 else:  
 return function(array\_data[1:], array\_data[0])  
  
  
print(function([0, 5, 2]))  
  
  
  
########################################################################################################################  
#node\_higher\_order  
  
from itertools import product  
from functools import reduce  
  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
 def pre\_order\_iter(self):  
 result = [self]  
 if self.left is not None:  
 left = self.left.pre\_order\_iter()  
 result = result + left  
 if self.right is not None:  
 right = self.right.pre\_order\_iter()  
 result = result + right  
 return result  
  
  
def function(node):  
 duos = product(node.pre\_order\_iter(), node.pre\_order\_iter())  
 multiplies = map(lambda x: x[0].value \* x[1].value, duos)  
 return reduce(lambda x, y: x + y, multiplies)  
  
  
n = Node(2)  
n.left = Node(1)  
n.right = Node(3)  
print(function(n))  
  
  
  
########################################################################################################################  
#node\_iterative  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
 def pre\_order\_iter(self):  
 stack = []  
 result = []  
 stack.append(self)  
  
 while len(stack) != 0:  
 node = stack.pop()  
 result.append(node)  
  
 if node.right is not None:  
 stack.append(node.right)  
  
 if node.left is not None:  
 stack.append(node.left)  
  
 return result  
  
  
def function(node):  
 result = 0  
 for current1 in node.pre\_order\_iter():  
 for current2 in node.pre\_order\_iter():  
 result += current1.value \* current2.value  
 return result  
  
  
n = Node(2)  
n.left = Node(1)  
n.right = Node(3)  
print(function(n))  
  
  
  
########################################################################################################################  
#node\_list\_comprehension  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
 def pre\_order\_iter(self):  
 result = [self]  
 if self.left is not None:  
 left = self.left.pre\_order\_iter()  
 result = result + left  
 if self.right is not None:  
 right = self.right.pre\_order\_iter()  
 result = result + right  
 return result  
  
  
def function(node):  
 multiplies = [x.value \* y.value for x in node.pre\_order\_iter() for y in node.pre\_order\_iter()]  
 total = 0  
 scanned = [total := total + x for x in multiplies]  
 return scanned[-1]  
  
  
n = Node(2)  
n.left = Node(1)  
n.right = Node(3)  
print(function(n))  
  
  
  
########################################################################################################################  
#node\_recursive  
  
class Node:  
 def \_\_init\_\_(self, value):  
 self.value = value  
 self.left = None  
 self.right = None  
  
 def pre\_order\_iter(self):  
 result = []  
 result.append(self)  
 if self.left is not None:  
 left = self.left.pre\_order\_iter()  
 result = result + left  
 if self.right is not None:  
 right = self.right.pre\_order\_iter()  
 result = result + right  
 return result  
  
  
def helper(first, second, second\_len=None):  
 if len(first) == 0:  
 return []  
  
 if len(second) == 0:  
 return []  
  
 if second\_len is None:  
 return helper(first, second, len(second))  
  
 result = [(first[0], second[0])]  
 if len(second) == second\_len:  
 inner = helper(first, second[1:], second\_len)  
 outer = helper(first[1:], second, second\_len)  
 result = result + inner + outer  
 else:  
 inner = helper(first, second[1:], second\_len)  
 result = result + inner  
 return result  
  
  
def function(powerset):  
 if len(powerset) == 0:  
 return 0  
 return powerset[0][0].value \* powerset[0][1].value + function(powerset[1:])  
  
  
n = Node(2)  
n.left = Node(1)  
n.right = Node(3)  
print(function(helper(n.pre\_order\_iter(), n.pre\_order\_iter())))  
  
  
  
########################################################################################################################  
#prime\_factors\_higher\_order  
  
def helper(number):  
 number\_array = range(1, number + 1)  
 prime\_array = filter(lambda x: number % x == 0, number\_array)  
 return len(list(prime\_array)) == 2  
  
  
def function(number):  
 primes = filter(helper, range(2, number + 1))  
 prime\_factors = filter(lambda x: number % x == 0, primes)  
 return list(prime\_factors)  
  
  
print(function(18))  
  
  
  
########################################################################################################################  
#prime\_factors\_iterative  
  
def helper(number):  
 if number == 1:  
 return False  
 for check\_num in range(2, int(number / 2) + 1):  
 if number % check\_num == 0:  
 return False  
 return True  
  
  
def function(number):  
 result = []  
 for idx\_num in range(2, number + 1):  
 if helper(idx\_num) and number % idx\_num == 0:  
 result.append(idx\_num)  
 return result  
  
  
print(function(18))  
  
  
  
########################################################################################################################  
#prime\_factors\_list\_comprehension  
  
def helper(number):  
 return len([value for value in range(1, number + 1) if number % value == 0]) == 2  
  
  
def function(number):  
 return [x for x in range(2, number + 1) if helper(x) and number % x == 0]  
  
  
print(function(18))  
  
  
  
########################################################################################################################  
#prime\_factors\_recursive  
  
def helper(number, current=None):  
 if current is None:  
 current = int(number / 2)  
 if number == 1:  
 return False  
 if current == 1:  
 return True  
 if number % current == 0:  
 return False  
 return helper(number, current - 1)  
  
  
def function(number, current=None):  
 if current is None:  
 return function(number, 2)  
  
 elif number == current and number % current == 0:  
 return [current]  
  
 elif number <= current:  
 return []  
  
 elif not helper(current):  
 return function(number, current + 1)  
  
 elif number % current == 0:  
 return [current] + function(number / current, current + 1)  
  
 else:  
 return function(number, current + 1)  
  
  
print(function(18))  
  
  
  
########################################################################################################################  
#quad\_mul\_higher\_order  
  
from functools import reduce  
  
  
def function(n):  
 array\_data = map(lambda x: x \*\* 2, range(1, n + 1))  
 return reduce(lambda x, y: x \* y, array\_data)  
  
  
print(function(3))  
  
  
  
########################################################################################################################  
#quad\_mul\_iterative  
  
def function(n):  
 array\_data = []  
 for i in range(1, n + 1):  
 array\_data.append(i \*\* 2)  
 result = 1  
 for value in array\_data:  
 result \*= value  
 return result  
  
  
print(function(3))  
  
  
  
########################################################################################################################  
#quad\_mul\_list\_comprehension  
  
from functools import reduce  
  
  
def function(n):  
 array\_data = [x\*\*2 for x in range(1, n + 1)]  
 total = 1  
 scanned = [total := total \* x for x in array\_data]  
 return scanned[-1]  
  
  
print(function(3))  
  
  
  
########################################################################################################################  
#quad\_mul\_recursive  
  
def function(n):  
 if n == 1:  
 return 1  
 return n \*\* 2 \* function(n - 1)  
  
  
print(function(3))  
  
  
  
########################################################################################################################  
#Store\_higher\_order  
  
from itertools import product  
  
  
class Store:  
 id = 0  
  
 def \_\_init\_\_(self, name, city):  
 self.name = name  
 self.city = city  
 self.orders = []  
 self.id = Store.id  
 Store.id += 1  
  
 def add\_order(self, order\_id):  
 self.orders.append(order\_id)  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
class Order:  
 id = 0  
  
 def \_\_init\_\_(self, product\_name, number, price):  
 self.product\_name = product\_name  
 self.magnitude = number  
 self.price = price  
 self.id = Order.id  
 Order.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
def function(store\_array, order\_array):  
 selected\_stores = filter(lambda x: x.name == "Edeka", store\_array)  
 selected\_orders = filter(lambda x: x.product\_name == "sour cream", order\_array)  
 selected\_product = product(selected\_stores, selected\_orders)  
 selected\_stores\_with\_products = filter(lambda tuple: tuple[1].id in tuple[0].orders, selected\_product)  
 return list(map(lambda x: x[0], selected\_stores\_with\_products))  
  
  
store\_array = []  
store\_array.append(Store("Edeka", "Leipzig"))  
store\_array.append(Store("Rewe", "MÃ¼nchen"))  
store\_array.append(Store("Lidl", "Leipzig"))  
store\_array.append(Store("Edeka", "Berlin"))  
  
order\_array = []  
order\_array.append(Order("sour cream", 100, 0.9))  
order\_array.append(Order("cheese", 230, 1.2))  
order\_array.append(Order("apples", 40, 0.5))  
order\_array.append(Order("potatoes", 2000, 0.2))  
order\_array.append(Order("pans", 10, 10.9))  
  
store\_array[0].add\_order(0)  
store\_array[0].add\_order(2)  
store\_array[0].add\_order(4)  
store\_array[1].add\_order(1)  
store\_array[1].add\_order(3)  
store\_array[2].add\_order(2)  
store\_array[2].add\_order(0)  
store\_array[2].add\_order(1)  
store\_array[3].add\_order(0)  
  
print(function(store\_array, order\_array))  
  
  
  
########################################################################################################################  
#Store\_iterative  
  
from itertools import product  
  
  
class Store:  
 id = 0  
  
 def \_\_init\_\_(self, name, city):  
 self.name = name  
 self.city = city  
 self.orders = []  
 self.id = Store.id  
 Store.id += 1  
  
 def add\_order(self, order\_id):  
 self.orders.append(order\_id)  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
class Order:  
 id = 0  
  
 def \_\_init\_\_(self, product\_name, number, price):  
 self.product\_name = product\_name  
 self.magnitude = number  
 self.price = price  
 self.id = Order.id  
 Order.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
def function(store\_array, order\_array):  
 stores = []  
 for store in store\_array:  
 if store.name == "Edeka":  
 stores.append(store)  
  
 orders = []  
 for order in order\_array:  
 if order.product\_name == "sour cream":  
 orders.append(order)  
  
 result = []  
 for store in stores:  
 for order in orders:  
 if order.id in store.orders:  
 result.append(store)  
  
 return result  
  
  
store\_array = []  
store\_array.append(Store("Edeka", "Leipzig"))  
store\_array.append(Store("Rewe", "MÃ¼nchen"))  
store\_array.append(Store("Lidl", "Leipzig"))  
store\_array.append(Store("Edeka", "Berlin"))  
  
order\_array = []  
order\_array.append(Order("sour cream", 100, 0.9))  
order\_array.append(Order("cheese", 230, 1.2))  
order\_array.append(Order("apples", 40, 0.5))  
order\_array.append(Order("potatoes", 2000, 0.2))  
order\_array.append(Order("pans", 10, 10.9))  
  
store\_array[0].add\_order(0)  
store\_array[0].add\_order(2)  
store\_array[0].add\_order(4)  
store\_array[1].add\_order(1)  
store\_array[1].add\_order(3)  
store\_array[2].add\_order(2)  
store\_array[2].add\_order(0)  
store\_array[2].add\_order(1)  
store\_array[3].add\_order(0)  
  
print(function(store\_array, order\_array))  
  
  
  
########################################################################################################################  
#Store\_list\_comprehension  
  
from itertools import product  
  
  
class Store:  
 id = 0  
  
 def \_\_init\_\_(self, name, city):  
 self.name = name  
 self.city = city  
 self.orders = []  
 self.id = Store.id  
 Store.id += 1  
  
 def add\_order(self, order\_id):  
 self.orders.append(order\_id)  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
class Order:  
 id = 0  
  
 def \_\_init\_\_(self, product\_name, number, price):  
 self.product\_name = product\_name  
 self.magnitude = number  
 self.price = price  
 self.id = Order.id  
 Order.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
def function(store\_array, order\_array):  
 selected\_stores = [store for store in store\_array if store.name == "Edeka"]  
 selected\_orders = [order for order in order\_array if order.product\_name == "sour cream"]  
 return [store for store in selected\_stores for order in selected\_orders if order.id in store.orders]  
  
  
store\_array = []  
store\_array.append(Store("Edeka", "Leipzig"))  
store\_array.append(Store("Rewe", "MÃ¼nchen"))  
store\_array.append(Store("Lidl", "Leipzig"))  
store\_array.append(Store("Edeka", "Berlin"))  
  
order\_array = []  
order\_array.append(Order("sour cream", 100, 0.9))  
order\_array.append(Order("cheese", 230, 1.2))  
order\_array.append(Order("apples", 40, 0.5))  
order\_array.append(Order("potatoes", 2000, 0.2))  
order\_array.append(Order("pans", 10, 10.9))  
  
store\_array[0].add\_order(0)  
store\_array[0].add\_order(2)  
store\_array[0].add\_order(4)  
store\_array[1].add\_order(1)  
store\_array[1].add\_order(3)  
store\_array[2].add\_order(2)  
store\_array[2].add\_order(0)  
store\_array[2].add\_order(1)  
store\_array[3].add\_order(0)  
  
print(function(store\_array, order\_array))  
  
  
  
########################################################################################################################  
#Store\_recursive  
  
from itertools import product  
  
  
class Store:  
 id = 0  
  
 def \_\_init\_\_(self, name, city):  
 self.name = name  
 self.city = city  
 self.orders = []  
 self.id = Store.id  
 Store.id += 1  
  
 def add\_order(self, order\_id):  
 self.orders.append(order\_id)  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
  
class Order:  
 id = 0  
  
 def \_\_init\_\_(self, product\_name, number, price):  
 self.product\_name = product\_name  
 self.magnitude = number  
 self.price = price  
 self.id = Order.id  
 Order.id += 1  
  
 def \_\_repr\_\_(self):  
 return str(self.id)  
  
def helper1(order\_array):  
 if len(order\_array) == 0:  
 return []  
 if order\_array[0].product\_name == "sour cream":  
 return [order\_array[0]] + helper1(order\_array[1:])  
 else:  
 return helper1(order\_array[1:])  
  
  
def helper2(store, order\_array):  
 if len(order\_array) == 0:  
 return False  
 if order\_array[0].id in store.orders:  
 return True  
 else:  
 return helper2(store, order\_array[1:])  
  
  
def function(store\_array, order\_array):  
 if len(store\_array)==0:  
 return []  
 if store\_array[0].name == "Edeka":  
 orders = helper1(order\_array)  
 if helper2(store\_array[0], orders):  
 return [store\_array[0]] + function(store\_array[1:], order\_array)  
  
 return function(store\_array[1:], order\_array)  
  
  
store\_array = []  
store\_array.append(Store("Edeka", "Leipzig"))  
store\_array.append(Store("Rewe", "MÃ¼nchen"))  
store\_array.append(Store("Lidl", "Leipzig"))  
store\_array.append(Store("Edeka", "Berlin"))  
  
order\_array = []  
order\_array.append(Order("sour cream", 100, 0.9))  
order\_array.append(Order("cheese", 230, 1.2))  
order\_array.append(Order("apples", 40, 0.5))  
order\_array.append(Order("potatoes", 2000, 0.2))  
order\_array.append(Order("pans", 10, 10.9))  
  
store\_array[0].add\_order(0)  
store\_array[0].add\_order(2)  
store\_array[0].add\_order(4)  
store\_array[1].add\_order(1)  
store\_array[1].add\_order(3)  
store\_array[2].add\_order(2)  
store\_array[2].add\_order(0)  
store\_array[2].add\_order(1)  
store\_array[3].add\_order(0)  
  
print(function(store\_array, order\_array))  
  
  
  
########################################################################################################################  
#students\_higher\_order  
  
class Student:  
 def \_\_init\_\_(self, age, name):  
 self.age = age  
 self.name = name  
  
 def \_\_repr\_\_(self):  
 return str(self.name)  
  
  
def function(student\_array):  
 return list(filter(lambda x: x.age >= 18, student\_array))  
  
  
students = []  
students.append(Student(17, "Hans"))  
students.append(Student(21, "Jasmin"))  
students.append(Student(32, "Florian"))  
print(function(students))  
  
  
  
########################################################################################################################  
#students\_iterative  
  
class Student:  
 def \_\_init\_\_(self, age, name):  
 self.age = age  
 self.name = name  
  
 def \_\_repr\_\_(self):  
 return str(self.name)  
  
  
def function(student\_array):  
 result = []  
 for student in student\_array:  
 if student.age >= 18:  
 result.append(student)  
 return result  
  
  
students = []  
students.append(Student(17, "Hans"))  
students.append(Student(21, "Jasmin"))  
students.append(Student(32, "Florian"))  
print(function(students))  
  
  
  
########################################################################################################################  
#students\_list\_comprehension  
  
class Student:  
 def \_\_init\_\_(self, age, name):  
 self.age = age  
 self.name = name  
  
 def \_\_repr\_\_(self):  
 return str(self.name)  
  
  
def function(student\_array):  
 return [student for student in student\_array if student.age >= 18]  
  
  
students = []  
students.append(Student(17, "Hans"))  
students.append(Student(21, "Jasmin"))  
students.append(Student(32, "Florian"))  
print(function(students))  
  
  
  
########################################################################################################################  
#students\_recursive  
  
class Student:  
 def \_\_init\_\_(self, age, name):  
 self.age = age  
 self.name = name  
  
 def \_\_repr\_\_(self):  
 return str(self.name)  
  
  
def function(student\_array):  
 if len(student\_array) == 0:  
 return []  
 elif student\_array[0].age >= 18:  
 return [student\_array[0]] + function(student\_array[1:])  
 else:  
 return function(student\_array[1:])  
  
  
students = []  
students.append(Student(17, "Hans"))  
students.append(Student(21, "Jasmin"))  
students.append(Student(32, "Florian"))  
print(function(students))