```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 27 08:04:06 2019

@author: hernan
LIA - MODO International Lab
EA Seminar
Version 1.0 31-07-2019

Multi-objective optimization basic methods
"""

import ea_base as ea



def dominates(find1, find2):
    """Determines whether fitness vector find1 dominates
    fitness vector find2
    If find1 dominates find2 returns True
    Otherwise returns False

    Parameters
    ----------
    find1: touple or list
        Vector of fitness values of individual 1
    find2: touple or list
        Vector of fitness values of individual 2

    Returns
    -------
    Boolean
        True if find1 dominates find2. False otherwise

    """
    dom = False
    better = 0
    better_or_equal = 0
    nobj = len(find1)

    for i in range(0,nobj):
        if (find1[i] >= find2[i]):
            better_or_equal += 1
            if find1[i] > find2[i]:
                better += 1

    if (better_or_equal == nobj) and better >= 1:
        dom = True
    return dom
```

```
def get_non_dominated_solutions(pop):
    """Extract non-dominated solutions from pop and return them
    Dominated solutions remain in pop

    Parameters
    ----------
    pop: Population
        Population fron which the set of non-dominated solutions will be
        extracted

    Returns
    -------
    Population
        The set of non-dominated solutions

    """
    size = len(pop)
    dom_count = [0] * size
    """
    Count how many times a solution has been dominated
    """
    for i in range(0,size):
        for j in range(0,size):
            if i != j:
                if dominates(pop[i].fitness, pop[j].fitness):
                    dom_count[j] += 1

    """
    A solutions i with dom_count[i] == 0 is non-dominated
    """

    ndpop = ea.Population()
    for i in range(size-1, -1, -1):
        if dom_count[i] == 0:
            # add non-dominated solution to ndpop
            ndpop.insert(0, pop[i])
            # remove non-dominated solution from pop
            pop.remove(pop[i])
    return ndpop


def non_dominated_sorting(pop):
    """Extract non-dominated fronts from pop and include them in a list
    The first front in the lits is the top front
    pop is empty after all fronts have been extracted
    Returns the list of non-diminated fronts
```

```
    Parameters
    ----------
    pop: Population
        Population fron which sets of non-dominated solutions will be
        extracted

    Returns
    -------
    List
        The sets of non-dominated solutions
    """

    fronts = []
    while (len(pop) > 0):
        fi  = get_non_dominated_solutions(pop)
        fronts.append(fi)
    return fronts

def front_rank(front, drank, i_index=0):
    """Ranks all solutions of a front with the specified drank
    A solutions can have two ranks: rank[0] and rank[0]
    By default, i_index=0  so drank is assigned to rank[0]

    Parameters
    ----------
    front: Population
        Front of solutions to be ranked
    drank: integer
        The rank assigned to all solutions of the front
    i_index:
        The index of the rank being set. Default is 0


    Returns
    -------
    None

    """
    # all solutions in the same front are assigned the same rank
    # the rank = front number
    for ind in front:
        ind.rank[i_index] = drank

def crowding_distance (front, nobj,i_index=1):
    """Computes crowding distance of a set of non-dominated solutions

    Parameters
    ----------
    front: Population
        Front of solutions to compute crowding distance
```

```
    nobj: integer
        The number of objectives
    i_index:
        The index of the rank corrsponding to crowding distance.
        Default is 1

    Returns
    -------
    None
    """

    nind = len(front)
    INFINITY = 1e+15
    EPS = 1e-10

    # Initialize cd = 0 to for all individuals in the front
    for i in range(0, nind):
        front[i].rank[i_index] = 0.0

    for i in range (0, nobj):
        # sort population by ith objective
        front.sort(key = lambda ind: ind.fitness[i], reverse = True)
        # get max and min fitness in the front
        maxf = front[0].fitness[i]
        minf = front[nind-1].fitness[i]

        # add a very large distance to extreme solutions
        front[0].rank[i_index] += INFINITY
        front[nind-1].rank[i_index] += INFINITY
        # add distance in objective i from solution j-1 to j+1
        for j in range(1, nind-1):
            d = front[j-1].fitness[i] - front[j+1].fitness[i]
            front[j].rank[i_index] += abs(d)/abs(EPS + maxf - minf)
```