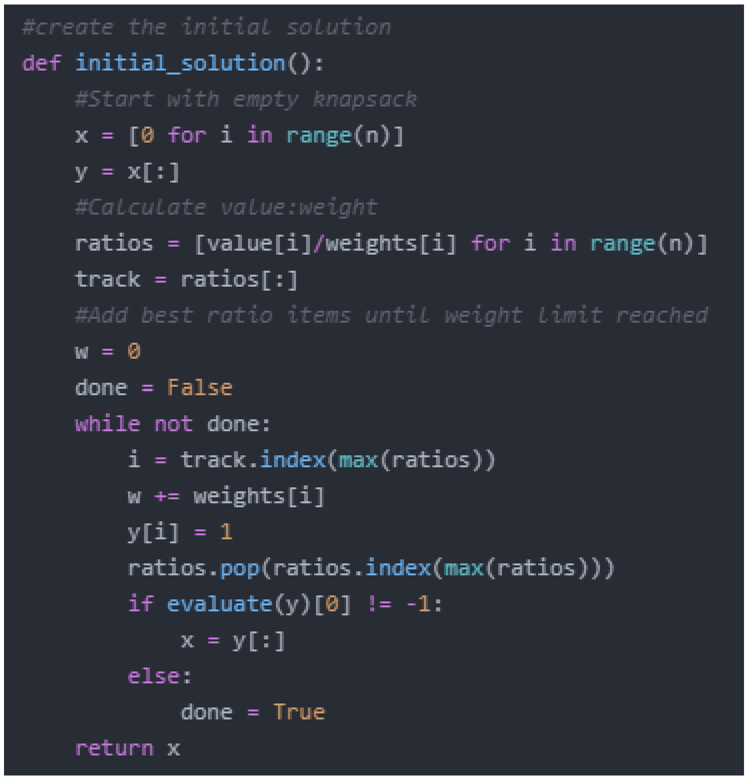
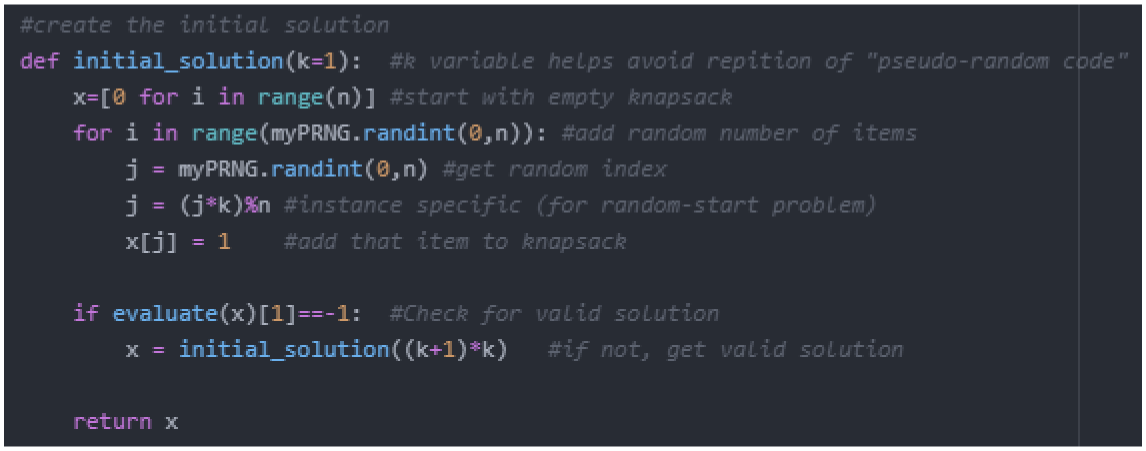
Method	Iterations	# Items Selected	Weight	Objective
Hill Climb – Best Improvement	2,550	18	2,495.9	27,152
Hill Climb – First Improvement	395	20	2,497.9	13,814.2
Hill Climb with Random Restart – Best Imp, k=1000	1,463,550	23	2,498.5	30,629.7
Hill Climb with Random Walk – Best Imp, p=.8	3	2,400	17	2,497.2

1a) We know our knapsack's only limitation is weight and our goal is to maximize value. So, a good strategy for determining an initial solution to the knapsack problem is adding items based off their *value to weight* ratio until our weight limit is reached. This will ensure we include the items that provide the most value for the weight they are taking up in the knapsack. (Logic pictured below)



```python
#create the initial solution
def initial_solution():
    #Start with empty knapsack
    x = [0 for i in range(n)]
    y = x[:]
    #Calculate value:weight
    ratios = [value[i]/weights[i] for i in range(n)]
    track = ratios[:]
    #Add best ratio items until weight limit reached
    w = 0
    done = False
    while not done:
        i = track.index(max(ratios))
        w += weights[i]
        y[i] = 1
        ratios.pop(ratios.index(max(ratios)))
        if evaluate(y)[0] != -1:
            x = y[:]
        else:
            done = True
    return x
```

Note: This will obviously not work for our random restart algorithm. Also, it resulted in most of our neighborhoods not being able to find a more optimized solution – resulting in uninteresting results (as far as seeing differences in the techniques are involved). So, we will be using the random initial solution for the given solutions. (Pictured below)



```python
#create the initial solution
def initial_solution(k=1):  #k variable helps avoid repition of "pseudo-random code"
    x=[0 for i in range(n)] #start with empty knapsack
    for i in range(myPRNG.randint(0,n)): #add random number of items
        j = myPRNG.randint(0,n) #get random index
        j = (j*k)%n #instance specific (for random-start problem)
        x[j] = 1    #add that item to knapsack

    if evaluate(x)[1]==-1:  #Check for valid solution
        x = initial_solution((k+1)*k)   #if not, get valid solution

    return x
```

Further, here are the results for all the problems (excluding random restart) with the ratio-based initial solution:

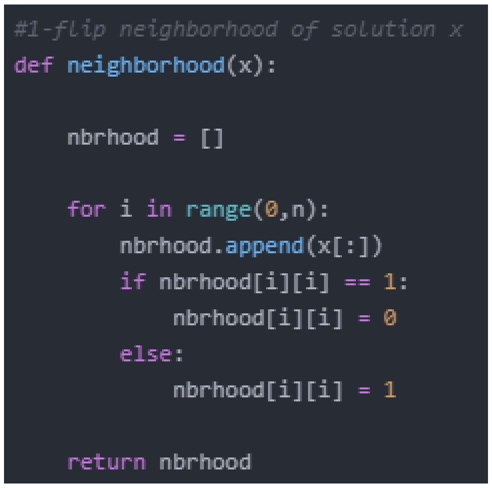
Iterations: 150

Items: 35

Value: 41,456.6

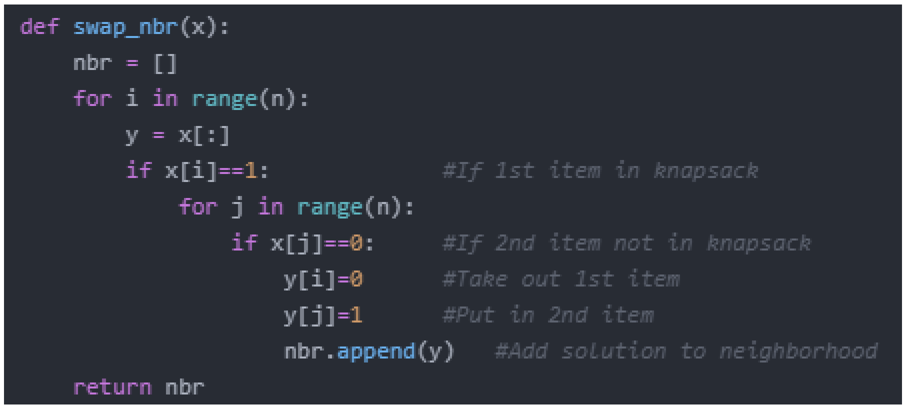
Weight: 2484.2

1b) Structure 1: 1-Flip neighborhood. This neighborhood is made up of n lists of size n where the ith value of the ith list is flipped from 1 to 0 or 0 to 1. It's size is n^2.



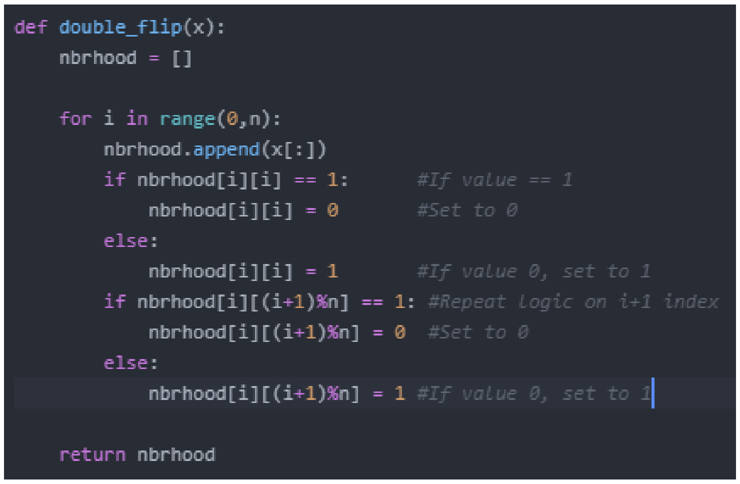
```python
#1-flip neighborhood of solution x
def neighborhood(x):

    nbrhood = []

    for i in range(0,n):
        nbrhood.append(x[:])
        if nbrhood[i][i] == 1:
            nbrhood[i][i] = 0
        else:
            nbrhood[i][i] = 1

    return nbrhood
```

Structure 2: Swap neighborhood. This neighborhood structure is made up of solutions where for each item included in your current solution ("in" the knapsack) you swap it for each item "out" of the knapsack. This neighborhood will return k lists of n-k solutions of size k, where k is the number of items included in a solution. It's size is k*n*(n-k)



```python
def swap_nbr(x):
    nbr = []
    for i in range(n):
        y = x[:]
        if x[i]==1:              #If 1st item in knapsack
            for j in range(n):
                if x[j]==0:      #If 2nd item not in knapsack
                    y[i]=0       #Take out 1st item
                    y[j]=1       #Put in 2nd item
                    nbr.append(y)   #Add solution to neighborhood
    return nbr
```

Structure 3: Double flip neighborhood. This neighborhood, similar to the 1-flip neighborhood, is made up of solutions where we swap both the ith and i+1 value for each index. It also returns n lists of size n so it's size is n^2.



```python
def double_flip(x):
    nbrhood = []

    for i in range(0,n):
        nbrhood.append(x[:])
        if nbrhood[i][i] == 1:      #If value == 1
            nbrhood[i][i] = 0       #Set to 0
        else:
            nbrhood[i][i] = 1       #If value 0, set to 1
        if nbrhood[i][(i+1)%n] == 1: #Repeat logic on i+1 index
            nbrhood[i][(i+1)%n] = 0  #Set to 0
        else:
            nbrhood[i][(i+1)%n] = 1 #If value 0, set to 1

    return nbrhood
```

1c) If a candidate solution is infeasible one solution would be to assign it a negative value, because our objective function is looking to maximize the value, it will avoid this solution.

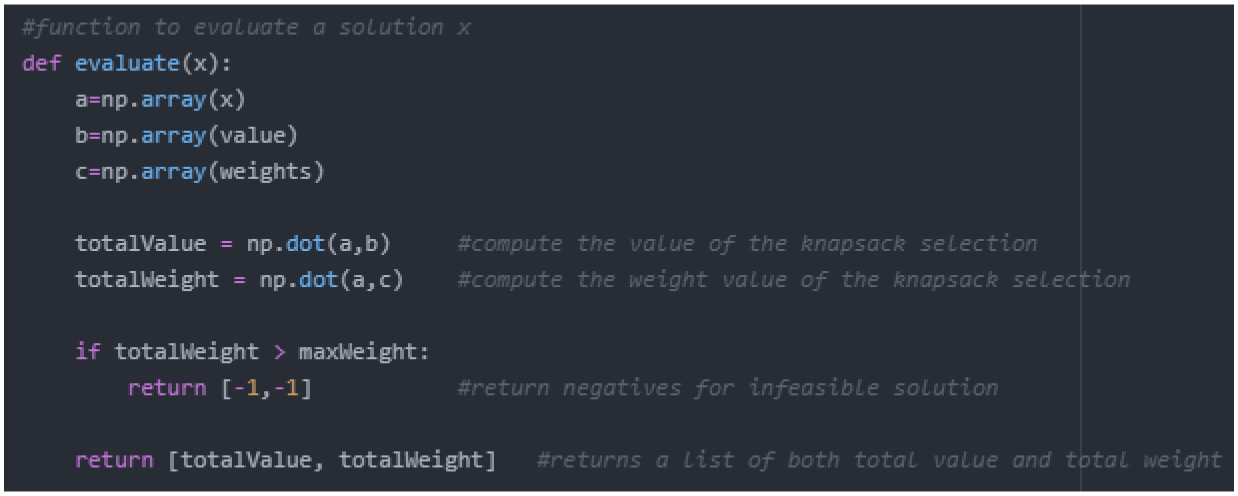
Another solution could be to have an additional condition in our "if" statement that checks to see if we found a better solution. That is:

Change "if evaluate(s)[0] > f_best[0]:" to "if evaluate(s)[0] > f_best[0] and evaluate(s)[1]<=maxWeight":

This change would prevent any infeasible solution from being set as our best solution.

Question 2:

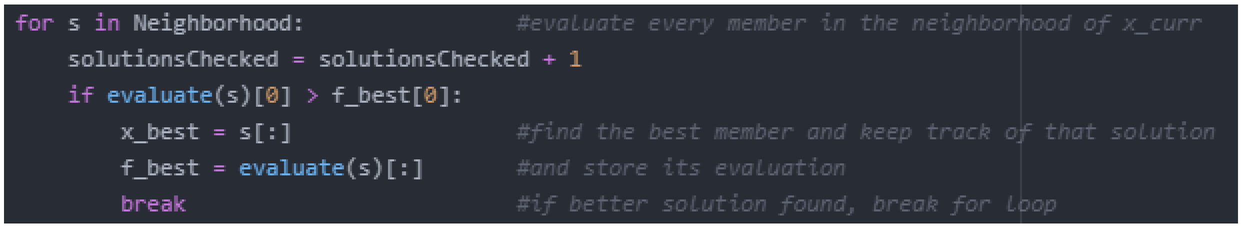
To finish the given code, we needed to provide logic to generate an initial solution (Random Initial Solution pictured in 1a). We also needed to provide a way to deal with infeasible solutions. We decided to use the negative value method described in 1c. We use the 1-flip method to determine our neighborhood.



```
#function to evaluate a solution x
def evaluate(x):
    a=np.array(x)
    b=np.array(value)
    c=np.array(weights)

    totalValue = np.dot(a,b)     #compute the value of the knapsack selection
    totalWeight = np.dot(a,c)    #compute the weight value of the knapsack selection

    if totalWeight > maxWeight:
        return [-1,-1]           #return negatives for infeasible solution

    return [totalValue, totalWeight]   #returns a list of both total value and total weight
```

Question 3:

To change from best improvement to first improvement we can simply add a "break" statement in our for loop that checks solutions in a returned neighborhood. This will result in no further solutions being evaluated in a given neighborhood once a single improvement is reached. We used the 1-Flip method to determine our neighborhood.

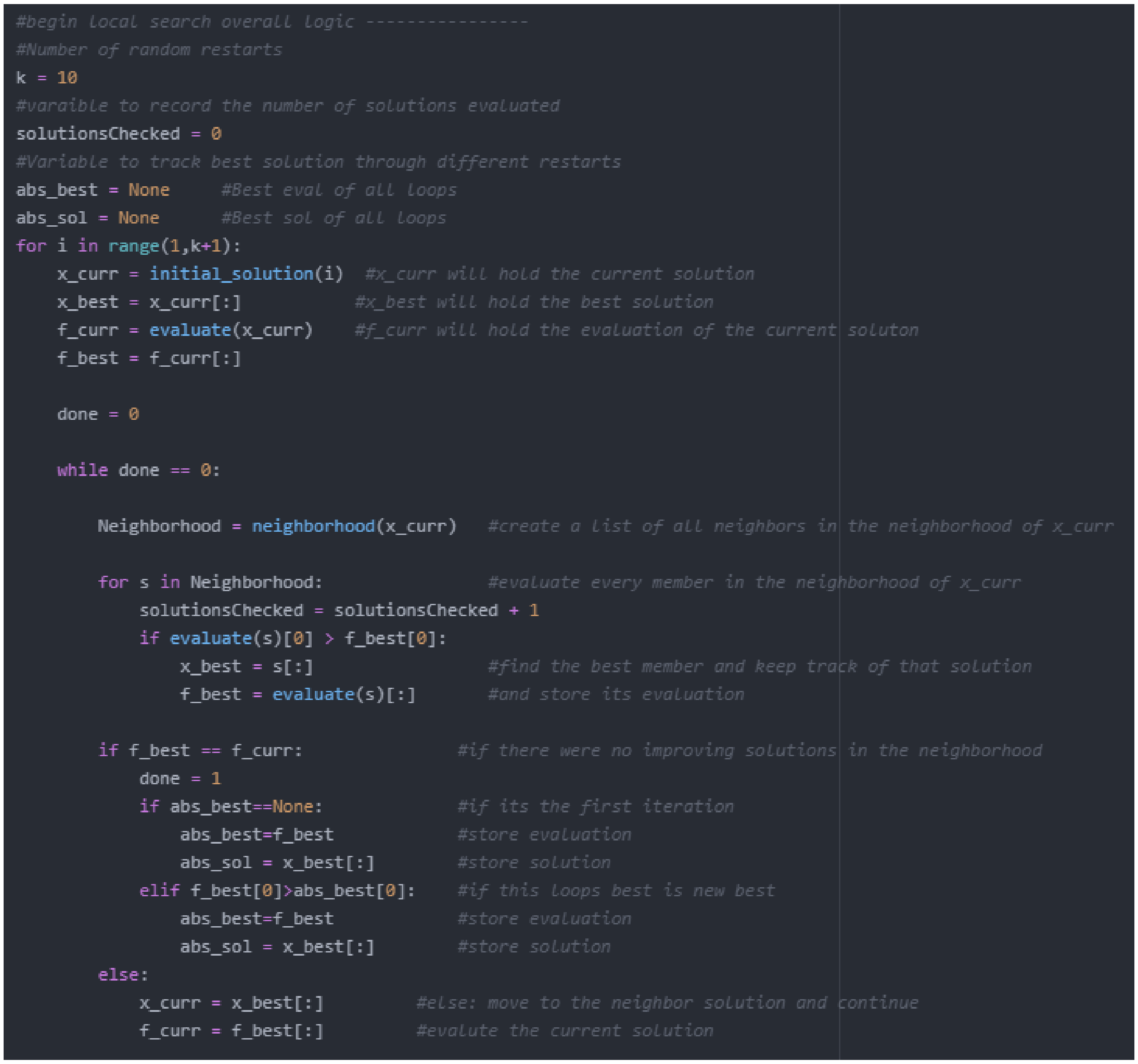


```
for s in Neighborhood:                 #evaluate every member in the neighborhood of x_curr
    solutionsChecked = solutionsChecked + 1
    if evaluate(s)[0] > f_best[0]:
        x_best = s[:]                  #find the best member and keep track of that solution
        f_best = evaluate(s)[:]        #and store its evaluation
        break                          #if better solution found, break for loop
```

Question 4:

For the random start we run the main loop k times, storing the best overall result in a variable outside of the loop. The current iteration is also pushed into our random initial solution equation to ensure we avoid the same starting spot due to the “pseudo-random” number generator. We used the 1-Flip method to determine our neighborhoods and the best improvement method to determine our next solution.

K	Iterations	# Items Selected	Weight	Objective
10	16,200	19	2,493.2	28,787.6
1000	1,463,550	23	2,498.5	30,629.7

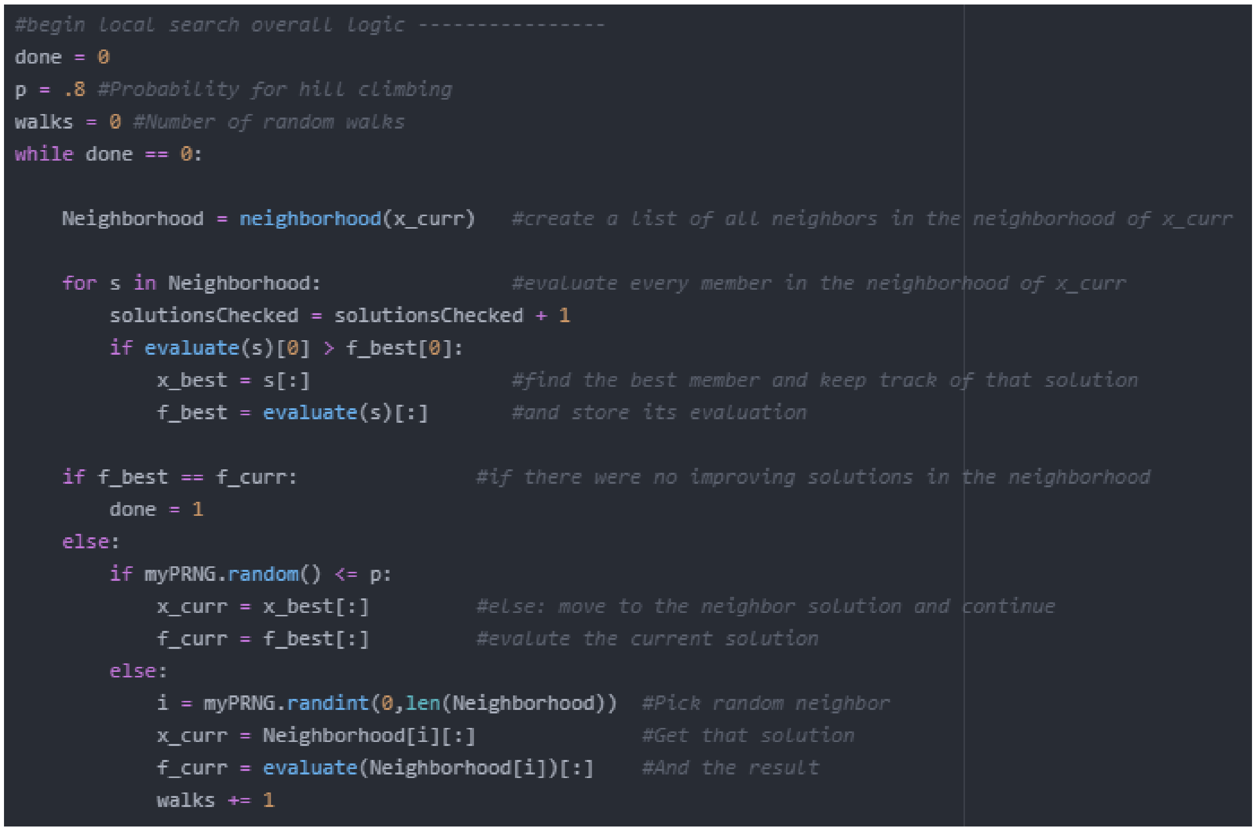


```python
#begin local search overall logic ----------------
#Number of random restarts
k = 10
#varaible to record the number of solutions evaluated
solutionsChecked = 0
#Variable to track best solution through different restarts
abs_best = None     #Best eval of all loops
abs_sol = None      #Best sol of all loops
for i in range(1,k+1):
    x_curr = initial_solution(i)  #x_curr will hold the current solution
    x_best = x_curr[:]           #x_best will hold the best solution
    f_curr = evaluate(x_curr)    #f_curr will hold the evaluation of the current soluton
    f_best = f_curr[:]

    done = 0

    while done == 0:

        Neighborhood = neighborhood(x_curr)   #create a list of all neighbors in the neighborhood of x_curr

        for s in Neighborhood:                #evaluate every member in the neighborhood of x_curr
            solutionsChecked = solutionsChecked + 1
            if evaluate(s)[0] > f_best[0]:
                x_best = s[:]                 #find the best member and keep track of that solution
                f_best = evaluate(s)[:]       #and store its evaluation

        if f_best == f_curr:               #if there were no improving solutions in the neighborhood
            done = 1
            if abs_best==None:             #if its the first iteration
                abs_best=f_best            #store evaluation
                abs_sol = x_best[:]        #store solution
            elif f_best[0]>abs_best[0]:    #if this loops best is new best
                abs_best=f_best            #store evaluation
                abs_sol = x_best[:]        #store solution
        else:
            x_curr = x_best[:]         #else: move to the neighbor solution and continue
            f_curr = f_best[:]         #evalute the current solution
```

Question 5:

For the hill climb with random walk algorithm we set x_curr and f_curr (found using best improvement method) with probability p. Otherwise, we select a random solution from the neighborhood. This logic can be seen below in the “if/else” statements. We used 1-Flip method to determine our neighborhoods and used the best improvement method to determine our next solution.

p	Walks	Iterations	# Items	Weight	Objective
.8	3	2,400	17	2,497.2	23,517.7
.2	31	5,850	19	2,496.9	20,685.2



```
#begin local search overall logic ----------------
done = 0
p = .8 #Probability for hill climbing
walks = 0 #Number of random walks
while done == 0:

    Neighborhood = neighborhood(x_curr)   #create a list of all neighbors in the neighborhood of x_curr

    for s in Neighborhood:                #evaluate every member in the neighborhood of x_curr
        solutionsChecked = solutionsChecked + 1
        if evaluate(s)[0] > f_best[0]:
            x_best = s[:]                 #find the best member and keep track of that solution
            f_best = evaluate(s)[:]       #and store its evaluation

    if f_best == f_curr:               #if there were no improving solutions in the neighborhood
        done = 1
    else:
        if myPRNG.random() <= p:
            x_curr = x_best[:]         #else: move to the neighbor solution and continue
            f_curr = f_best[:]         #evalute the current solution
        else:
            i = myPRNG.randint(0,len(Neighborhood))  #Pick random neighbor
            x_curr = Neighborhood[i][:]              #Get that solution
            f_curr = evaluate(Neighborhood[i])[:]    #And the result
            walks += 1
```