<u>Explanation of code</u>

<u>Function read_fasta:</u>

With the following function I insert the sequences in a list.

```
def read_fasta(filename):

    with open(filename) as file:
        lines = file.readlines()
        seq = [line.replace("\n","") for line in lines]

    return seq
```

<u>Function find_kmers(seq,k):</u>

With the following function I cut every sequence in kmers.

```
def find_kmers(seq,k):

# There is a case where the sequence is not a multiple of k.
# For example seq = ACACT and k = 2 -> kmers = AC, AC, T so I have to consider the last item.
# So I make dummy = AC, AC, T and then I make the list of kmers = AC, AC

    dummy = list(set([seq[i:i+k] for i in range(0, len(seq), k)]))
    kmers = [i for i in dummy if len(i)==k]

    return kmers
```

<u>Function pwm_kmers(sequences,k):</u>

With the following function I cut calculate the PWM from a set of aligned sequences of the same length. I copied from your example.

```
def pwm_kmers(sequences,k):

    nuc = ['A', 'C', 'G', 'T']
    profile=[[0 for i in range(len(sequences[0]))] for j in range(len(nuc))]

    for instance in sequences:
        for j in range(len(instance)):
            residue=instance[j]
            profile[nuc.index(residue)][j]+=1
            profile[nuc.index(residue)][j]=float(profile[nuc.index(residue)][j])

    pwm = np.array(profile)
    pwm = pwm/len(sequences)

    return pwm
```

Function select_rand_kmers(sequences,k):

The following function select randomly one kmer from every sequence.

```
def select_rand_kmers(sequences,k):

    rand_kmers = []

    for seq in sequences:
        kmers = find_kmers(seq,k)
        rand_kmers.append(random.choice(kmers))

    return rand_kmers
```

Function scores_pwm(kmers,pwm):

The following function calculate the score for every kmer based on the values of PWM.

```
def scores_pwm(kmers,pwm):

    nuc = ['A', 'C', 'G', 'T']
    scores = [0 for i in kmers]

    for i, kmer in enumerate(kmers):
        for j, nucleotide in enumerate(kmer):
            scores[i] += pwm[nuc.index(nucleotide)][j]

    return scores
```

Function random_better_scores(scores_1,kmers_1,sequences,pwm,k):

The following function take the kmers (kmers_1) and scores (scores_1) that I had. It selects with the same way new random kmers (kmers_2) and calculates the new scores (scores_2). If the function finds a higher scored kmer from the selected sequence, it replace the old kmer with the new one.

```
def random_better_scores(scores_1,kmers_1,sequences,pwm,k):

    kmers_2 = select_rand_kmers(sequences,k)
    scores_2 = scores_pwm(kmers_2,pwm)

    for i in range(len(scores_1)):
        if scores_1[i]<scores_2[i]:
            kmers_1[i] = kmers_2[i]
            scores_1[i]=scores_2[i]

    return kmers_1,scores_1
```

Function pwm_Entropy_Information(pwm):

This function calculates the mean information based on the entropy’s formula using as data the PWM.

```
def pwm_Entropy_Information(pwm):

    k = pwm.shape[1]

    information = np.zeros([1,k]) #computing the information of each position

    for i in range(k):
        information[0,i] = 2-abs(sum([elem*np.log2(elem) for elem in pwm[:,i] if elem > 0]))

    sum_information = np.sum(information)
    mean_information = sum_information/k

    return mean_information
```

Function Gibbs(filename,I,k):

This function is the implementation of Gibb’s Algorithm for k with values from 3 to 8. It takes the threshold (I) and I want to returns the PWM for every single one k, that has greater score (mean_information) than the threshold (I).

```
def Gibbs(filename,I,k):

    sequences = read_fasta(filename)

    kmers = select_rand_kmers(sequences,k)
    pwm = pwm_kmers(kmers,k)
    scores = scores_pwm(kmers,pwm)
    max_score = pwm_Entropy_Information(pwm)

    while max_score < I:

        kmers,scores = random_better_scores(scores,kmers,sequences,pwm,k)
        pwm = pwm_kmers(kmers,k)
        max_score = pwm_Entropy_Information(pwm)

    return pwm,max_score
```