Frequency_analysis

```
# letterset records the letters a-zA-Z

import string
import operator
import random
from itertools import combinations


letterset = frozenset(string.ascii_letters)
```

In python, there are currently two builtin set types, set and frozenset. The set type is mutable -- ***the contents can be changed using methods like add() and remove()***. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set.

The frozenset type is immutable and hashable -- ***its contents cannot be altered after is created***; however, it ***can be used as a dictionary key or as an element of another set.***

```
# documentation on frozen set https://docs.python.org/2.4/lib/types-set.html
type(letterset)
```

```
    <type 'frozenset'>
```

```
# letterfreq lists the letters a-z, listed in decreasing order of frequency, as given in HPS.
letterfreq = list("etaonrishdlfcmugypwbvkxjqz")
letterprops = [['e','.131'], ['t','.105'], ['a','.082'], ['o', '.080'], ['n', '.071'],
['r','.068'], ['i','.064'], ['s','.061'],['h', '.053'],['d','.038'],['l','.034'],['f','.029'],
['c','.028'],['m','.025'],['u','.025'],['g','.020'],['y','.020'],['p','.020'],['w','.015'],
['b','.014'],['v','.009'],['k','.004'],['x','.002'],['j','.001'],['q','.001'],['z','.001']]
```

```
letterprops[1:3]
```

```
    [['t', '.105'], ['a', '.082']]
```

Frequency of Bigrams and Trigrams

```
bigramfreq = [('th',168),('he',132),('an',92),('re',91),('er',88),('in',86),('on',71),('at',68),
('nd',61),('st',53),('es',52),('en',51),('of',49),('te',46),('ed',46)]  # bigram frequency
```

```
trigramfreq = ['the', 'and', 'tha', 'ent', 'ing', 'ion', 'tio', 'for', 'nde', 'has', 'nce',
'edt', 'tis', 'oft', 'sth', 'men'] # trigram frequency
```

```
# this function removes all punctuation and spaces from X, and makes all the letters lowercase.
def onlyletters(X):
    return filter(letterset.__contains__, X).lower()
```

```
plainquotes = ["in modern cryptographic language the ring settings did not actually contribute
entropy to the key used for encrypting the message rather the ring settings were part of a
separate key along with the rest of the setup such as wheel order and plug settings used to
encrypt an initialization vector for the message the session key consisted of the complete setup
except for the ring settings plus the initial rotor positions chosen arbitrarily by the sender
the message setting the important part of this session key was the rotor positions not the ring
positions however by encoding the rotor position into the ring position using the ring settings
additional variability was added to the encryption of the initialization vectorenigma was
designed to be secure even if the rotor wiring was known to an opponent although in practice
there was considerable effort to keep the wiring secret if the wiring is secret the total number
of possible configurations has been calculated to be around approximately bits with known wiring
and other operational constraints this is reduced to around  bits  users of enigma were confident
of its security because of the large number of possibilities it was not then feasible for an
adversary to even begin to try every possible configuration in a brute force attack", "most of
the key was kept constant for a set time period typically a day however a different initial rotor
position was used for each message a concept similar to an initialisation vector in modern
```

```
cryptography the reason for this is that were a number of messages to be encrypted with identical
or near identical settings termed in cryptanalysis as being in depth it would be possible to
attack the messages using a statistical procedure such as friedmans index of coincidence the
starting position for the rotors was transmitted just before the ciphertext usually after having
been enciphered the exact method used was termed the indicator procedure it was design weakness
and operator sloppiness in these indicator procedures that were two of the main reasons that
breaking enigma messages was possible  one of the earliest indicator procedures was used by
polish cryptanalysts to make the initial breaks into the enigma the procedure was for the
operator to set up his machine in accordance with his settings list which included a global
initial position for the rotors meaning ground setting perhaps the operator turned his rotors
until aoh was visible through the rotor windows at that point the operator chose his own
arbitrary starting position for that particular message an operator might select ein and these
became the message settings for that encryption session the operator then typed ein into the
machine twice to allow for detection of transmission errors the results were an encrypted
indicator the ein typed twice might turn into nonsense which would be transmitted along with the
message finally the operator then spun the rotors to his message settings ein in this example and
typed"]
```

Caesar and Vinegere Cipher

```
# this function encrypts the string X using a shift cipher (also known as a Caesar cipher).
# it currently prevents a shift of 0.

def randomsub():
    '''
    Return random substring
    '''
    return subencrypt(random.choice(plainquotes))

def shiftencrypt(X):
   X = onlyletters(X).upper()
   shiftamt = random.randint(1,25)
   output = ''.join(chr((ord(ch)-ord('A')+shiftamt)%26+ord('A')) for ch in X)
   return output

def vigenereencrypt(X, keyword = ""):
      X = onlyletters(X).upper()
      if len(keyword) != 0:
            keyword = onlyletters(keyword).lower()
            keylength = len(keyword)
            shifts = [ord(ch) - ord('a') for ch in keyword]
      else:
            keylength = random.randint(5,8)
            shifts = []
            for i in range(0,keylength):
                  shifts.append(random.randint(1,25))
      output = ''.join(chr((ord(X[i])-ord('A')+shifts[i%keylength])%26+ord('A')) for i in
range(0,len(X)))
      return output
```

Frequency Analysis Function

```
# this function counts substrings of the string X of length n.  It returns a dictionary whose
keys are the
# substrings which occur and whose values are the number of times they occur.
def countsubstrings(X,n):
    subdict = {}
    for i in range(0,len(X)-n+1):
        if X[i:i+n] in subdict:
            subdict[X[i:i+n]] += 1
        else:
            subdict[X[i:i+n]] = 1
    return subdict

def ranksubstrings(X,n,entries = 20):
```

```
    tempdict = countsubstrings(X,n)
    return sorted(tempdict.iteritems(), key=operator.itemgetter(1), reverse = True)[0:entries]
```

```
def countsubstrings(X,n):
    subdict = {}
    for i in range(0,len(X)-n+1):
        if X[i:i+n] in subdict:
            subdict[X[i:i+n]] += 1
        else:
            subdict[X[i:i+n]] = 1
    return subdict

def randomshift():
    '''
    Generate random text

    '''
    return shiftencrypt(random.choice(plainquotes))
```

```
X = randomshift()  # generate random encrioted message ffrom plain quotes
```

```
countsubstrings(X,1)
```

```
{'A': 2,
 'B': 19,
 'C': 2,
 'D': 66,
 'E': 20,
 'F': 32,
 'G': 26,
 'H': 133,
 'I': 23,
 'J': 35,
 'K': 42,
 'L': 99,
 'N': 8,
 'O': 26,
 'P': 11,
 'Q': 86,
 'R': 90,
 'S': 31,
 'U': 76,
 'V': 80,
 'W': 124,
 'X': 25,
 'Y': 8,
 'Z': 18}
```

```
countsubstrings(X,2)
```

```
WARNING: Output truncated!
full_output.txt


{'AF': 1,
 'AL': 1,
 'BD': 1,
 'BE': 3,
 'BF': 2,
 'BH': 2,
 'BS': 5,
 'BW': 3,
 'BX': 1,
 'BZ': 2,
 'CD': 2,
 'DE': 3,
 'DF': 3,
 'DG': 3,
 'DJ': 4,
 'DL': 1,
```

```
 'DO': 10,
 'DQ': 7,
 'DS': 2,
 'DU': 10,
 'DV': 10,
 'DW': 10,
 'DX': 1,
 'DZ': 2,
 'EB': 2,
 'EH': 7,
 'EL': 5,
 'EO': 4,
 'EU': 1,
 'EX': 1,
 'FD': 2,
 'FH': 4,
 'FK': 2,
 'FN': 1,
 'FO': 1,
 'FR': 9,
 'FU': 6,
 'FW': 4,
 'FX': 3,
 'GD': 1,
 'GE': 1,
 'GG': 2,
 'GH': 7,
 'GI': 1,
 'GL': 3,
 'GQ': 1,
 'GR': 2,
 'GS': 1,
 'GW': 5,
 'GX': 1,
 'GY': 1,
 'HA': 1,
 'HB': 4,
 'HD': 3,
 'HF': 11,
 'HG': 8,
 'HH': 7,
 'HI': 3,
 'HJ': 1,

...

 'UE': 2,
 'UF': 1,
 'UG': 1,
 'UH': 10,
 'UI': 1,
 'UJ': 1,
 'UL': 14,
 'UQ': 1,
 'UR': 11,
 'US': 3,
 'UV': 2,
 'UW': 8,
 'UX': 1,
 'UZ': 1,
 'VD': 6,
 'VE': 1,
 'VF': 2,
 'VG': 2,
 'VH': 21,
 'VK': 2,
 'VL': 17,
 'VN': 1,
 'VQ': 2,
 'VR': 1,
 'VS': 1,
 'VU': 1,
```

```
 'VV': 11,
 'VW': 6,
 'VX': 3,
 'VZ': 3,
 'WB': 2,
 'WD': 6,
 'WH': 7,
 'WI': 1,
 'WK': 35,
 'WL': 25,
 'WR': 23,
 'WS': 1,
 'WU': 5,
 'WV': 4,
 'WW': 11,
 'WX': 3,
 'WZ': 1,
 'XD': 2,
 'XF': 2,
 'XJ': 2,
 'XO': 1,
 'XP': 2,
 'XQ': 2,
 'XS': 2,
 'XU': 4,
 'XV': 6,
 'XW': 2,
 'YD': 1,
 'YH': 7,
 'ZD': 6,
 'ZH': 3,
 'ZK': 1,
 'ZL': 6,
 'ZQ': 2}
```

full_output.txt

```
ranksubstrings(X,1) # most recurring string
```

```
[('H', 133),
 ('W', 124),
 ('L', 99),
 ('R', 90),
 ('Q', 86),
 ('V', 80),
 ('U', 76),
 ('D', 66),
 ('K', 42),
 ('J', 35),
 ('F', 32),
 ('S', 31),
 ('G', 26),
 ('O', 26),
 ('X', 25),
 ('I', 23),
 ('E', 20),
 ('B', 19),
 ('Z', 18),
 ('P', 11)]
```

```
ranksubstrings(X,2)
```

```
[('WK', 35),
 ('KH', 31),
 ('LQ', 28),
 ('HU', 27),
 ('WL', 25),
 ('WR', 23),
 ('RQ', 23),
 ('VH', 21),
 ('QJ', 21),
 ('LW', 19),
 ('VL', 17),
 ('HQ', 16),
```

```
 ('HW', 15),
 ('RU', 14),
 ('HV', 14),
 ('UL', 14),
 ('LR', 14),
 ('RW', 13),
 ('UD', 12),
 ('JV', 11)]
```

```
ranksubstrings(X,2,35)
```

```
[('WK', 35),
 ('KH', 31),
 ('LQ', 28),
 ('HU', 27),
 ('WL', 25),
 ('WR', 23),
 ('RQ', 23),
 ('VH', 21),
 ('QJ', 21),
 ('LW', 19),
 ('VL', 17),
 ('HQ', 16),
 ('HW', 15),
 ('RU', 14),
 ('HV', 14),
 ('UL', 14),
 ('LR', 14),
 ('RW', 13),
 ('UD', 12),
 ('JV', 11),
 ('WW', 11),
 ('HF', 11),
 ('UR', 11),
 ('VV', 11),
 ('RI', 10),
 ('UH', 10),
 ('SR', 10),
 ('DO', 10),
 ('DV', 10),
 ('DW', 10),
 ('DU', 10),
 ('RV', 9),
 ('FR', 9),
 ('QW', 9),
 ('QL', 9)]
```

```
def shiftdecrypt(X,n):
    Y = ''.join(chr((ord(ch)-ord('A')+n)%26+ord('A')) for ch in X)
    print Y
```

Since the most ocurring letter is h from our frequency analysis we conclude that this corresponds to "e " and we encrypt it as below:

e = 5

h = 11

Therefore we use -3 in the encryption. If it was B that corresponde to the most common cipher word we use 3 (as in 5-2)

```
shiftdecrypt(X,-3) # we decrypt without knowing the shift
```

```
INMODERNCRYPTOGRAPHICLANGUAGETHERINGSETTINGSDIDNOTACTUALLYCONTRIBUTE\
ENTROPYTOTHEKEYUSEDFORENCRYPTINGTHEMESSAGERATHERTHERINGSETTINGSWEREP\
ARTOFASEPARATEKEYALONGWITHTHERESTOFTHESETUPSUCHASWHEELORDERANDPLUGSE\
TTINGSUSEDTOENCRYPTANINITIALIZATIONVECTORFORTHEMESSAGETHESESSIONKEYC\
ONSISTEDOFTHECOMPLETESETUPEXCEPTFORTHERINGSETTINGSPLUSTHEINITIALROTO\
RPOSITIONSCHOSENARBITRARILYBYTHESENDERTHEMESSAGESETTINGTHEIMPORTANTP\
ARTOFTHISSESSIONKEYWASTHEROTORPOSITIONSNOTTHERINGPOSITIONSHOWEVERBYE\
NCODINGTHEROTORPOSITIONINTOTHERINGPOSITIONUSINGTHERINGSETTINGSADDITI\
ONALVARIABILITYWASADDEDTOTHEENCRYPTIONOFTHEINITIALIZATIONVECTORENIGM\
AWASDESIGNEDTOBESECUREEVENIFTHEROTORWIRINGWASKNOWNTOANOPPONENTALTHOU\
GHINPRACTICETHEREWASCONSIDERABLEEFFORTTOKEEPTHEWIRINGSECRETIFTHEWIRI\
```

```
    NGISSECRETTHETOTALNUMBEROFPOSSIBLECONFIGURATIONSHASBEENCALCULATEDTOB\
    EAROUNDAPPROXIMATELYBITSWITHKNOWNWIRINGANDOTHEROPERATIONALCONSTRAINT\
    STHISISREDUCEDTOAROUNDBITSUSERSOFENIGMAWERECONFIDENTOFITSSECURITYBEC\
    AUSEOFTHELARGENUMBEROFPOSSIBILITIESITWASNOTTHENFEASIBLEFORANADVERSAR\
    YTOEVENBEGINTOTRYEVERYPOSSIBLECONFIGURATIONINABRUTEFORCEATTACK
```

```
X = randomshift()
```

```
def compareplots(Y,shift_amount):
      # This was adapted from code found at http://www.packtpub.com/article/plotting-data-sage
      import numpy
      import matplotlib.pyplot as plt
      data = numpy.array([8.15, 1.44, 2.76, 3.79,13.11, 2.92, 1.99, 5.26, 6.35, .13, .42, 3.39,
2.54, 7.1, 8, 1.98, .12, 6.83, 6.10, 10.47, 2.46, .92, 1.54, .17, 1.98, .08])
      listholder = []
      for i in range(0,26):
          listholder.append(i+.5)
      bar_centers = numpy.array(listholder)

      # Plot
      fig = plt.figure(figsize=(7,4)) # size in inches
      plt.bar(bar_centers, data,
              width=1.0, align='center', color='orange', ecolor='black')
      plt.ylabel('% frequency in English')

      # Label ticks on x axis
      axes = fig.gca()
      axes.set_xticks(bar_centers)
      axes.set_xticklabels(['a',
'b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'])

      plt.savefig('Bar_Chart.png')
      plt.close()

      cipherfreq = countsubstrings(Y,1)
      for ch in letterset:
         if ch.isupper() and ch not in cipherfreq:
            cipherfreq[ch] = 0
      totalchars = len(Y)
      heightlist = []
      ticklabels = []
      # The +.001 is just a hack to prevent the fact that if Z = 0, then the plot rescaled to
take up 25 units, not 26.
      for i in range(ord('A'),ord('Z')+1):
         heightlist.append((100*cipherfreq[chr(i)]/totalchars)+.001)
         ticklabels.append(chr(i))

      data = numpy.array(heightlist)

      # Plot
      fig = plt.figure(figsize=(7,4)) # size in inches
      plt.bar(bar_centers, data,
              width=1.0, align='center', color='blue', ecolor='black')
      plt.ylabel('% frequency in the ciphertext')

      # Label ticks on x axis
      axes = fig.gca()
      axes.set_xticks(bar_centers)
      axes.set_xticklabels(ticklabels)
      plt.savefig('Bar_Chart2.png')
      plt.close()
```

```
def breakshiftcipher(X):
   '''
   Compare frequency analysis
   '''
   @interact
   def _(shift_amount = selector(range(-25,26), default=0)):
```

```
      cipherfreq = countsubstrings(X,1)
      Y = ''.join(chr((ord(ch)-ord('A')+shift_amount)%26+ord('A')) for ch in X)
      compareplots(Y,shift_amount)
      print Y
```

```
breakshiftcipher(X) # This was decrypted using only test Analysis
```

shift_amount 0

BREAKING VINEGERE CIPHER

```
def vigenereencrypt(X, keyword = ""):
      '''
      Vinegere cipher
      '''
      X = onlyletters(X).upper()
      if len(keyword) != 0:
            keyword = onlyletters(keyword).lower()
            keylength = len(keyword)
            shifts = [ord(ch) - ord('a') for ch in keyword]
      else:
            keylength = random.randint(5,8)
            shifts = []
            for i in range(0,keylength):
                  shifts.append(random.randint(1,25))
      output = ''.join(chr((ord(X[i])-ord('A')+shifts[i%keylength])%26+ord('A')) for i in
range(0,len(X)))
      return output

def randomvigenere():
    '''
    generate random vinegere
    '''
    return vigenereencrypt(random.choice(plainquotes))
```

```
# The following function finds all substrings of length n which occur multiple times and then
identifies the gaps separating # the multiple occurrences.
def findgaplengths(X,n):
      strdict = {}
      for i in range(0,len(X)-n+1):
            if X[i:i+n] in strdict:
                  strdict[X[i:i+n]].append(i)
            else:
                  strdict[X[i:i+n]] = [i]
      gapholder = []
      for key in strdict:
            gapholder += [abs(a-b) for a,b in combinations(strdict[key],2)]
      gapholder.sort()
      return gapholder
```

```
X = randomvigenere()
```

```
X
```

```
'RHGVSMYASMIFBTGMIWYVCPWAFGHHSYFLSVXPRXDGVPTWHATPHTZNCHITMJSDJOSTEKN\
YTGVLSMWPMANTZTSATKDQWPYBCPAHXNGGHMTKSCGORXGUENJTQQRJJIHUMTNEOTXVFGW\
PMANTZKWHYBCPZLHMCTMURHRGVUHKMRXVLKORLFYASTIHXHBHSYYAWUMZYAOVALWXOPY\
TGXFQJTJLGCKLXMCDILSVFATAJWKKXONWSPXPHTZQVUJTFKHLSMWEESXXHVMULLHGVTJ\
WWPGYDIHCRHQRGKWHXUSKRNNGRGTAMBHYSBQWPGTVXLWDPLYHOVXHHDHJITJLGCKLXNG\
KRNFLHCXPXMWEESUKCEIKZKSUYJMTGHVPJWACRZNGRGBVKVCKRJNWSPGLYASUXHWMWPK\
WTLWVMVSYCTXOJKCVSYXPOUXYFGGOMAYXRLYZYUSHSYJMVGGPUASTXLCMIUYHQEMCJAJ\
KVCZPSZPGIUJGQKTOJKSFXOJXLCGARXHJSKZLSFAHXMSTQLIMVGMUIBQCXVWIFQGLINF\
```

```
GMABTGFIZNZBYIHPGSUWHSWCRIYFMCTWSTIDKRLXLWPXOJLSKRKNVOVSYUKCEIKZKSUX\
OFMKGVLYPCQJAMXACMUWXOUSUXMVCXIWXOMMULXBKKTFFSUWHLXGYEZUHGUMIQXCPIVK\
MVGIHWEWGWANGRKGHYHFRVVHXRWVLXPOUYZJWPATVQBGJGYDIHCRHQRGVWATFOMIAMXW\
PMANTZDVLFDGKRATMVGIUNZACXOJIFQGLINFGAHXYCTXOJHDGVHYHFVSZJMIRLPXFOEL\
PSXWPEJHHFFEUHXKKXOMBGUIAYBBIWSNLHYLPHAWPGSZWSFENQHPCPPSBHKESUHGKXPT\
GTQVAMXFQXVWLAGEUNGUIVVZGRUIAYBBITLWAORWAMXCRIYFMCTXBWGSFLPXKCVSYXNB\
VMSFHVYEZABGKFSJMVTSBLAHJIYTMCTAPSWCYWHYMVCXWTBBVXOJHDGVHYHFELVXXVKW\
VBGOTFPYKOTCZYTFVMULICUMANHBHSYYAOVTHWMWEYSFKAGWZFZSCRVUXFCXVWFWILAX\
XZGGAJBBCRKYASUIIJVOOIAMXAGWZFZSUIAYBBIWMTKHJEAJGQTCWYBCPWLXLWQRAMXC\
RIYFMCTXOJGHATLIXWPMUYHHJITFVVKRLYPWEIATTZNSDKHFFIAJVHKSUTYHTEUXFWUW\
PTGSTVVWLHJIYJLINXZBXFGEUJGQTCWYXRKRKNVOVSYYASGMUYRDGHABBQGQPLAHVYYS\
BBVSUTGGGRZJPVKGOBHINHIJMFCRZRBHVIKFECPKDNMVVLLRXGUENJYWPESQRHJIVUXF\
CXVWMVGRZUNBVLLWHHQVZYHVKWTJLGCKLXXHVMULLSKRPSMVKWLCTARPLFGRVCWJW'
```

```
findgaplengths(X,5)# find all pieces of ciphertext of length 5 that occur multiple times
                   #and it will report the length of the gaps between them. Most of the time, this
gap length will
                   # be        divisible by the keyword #length.
```

```
[48,
 48,
 48,
 60,
 60,
 60,
 72,
 72,
 72,
 96,
 114,
 114,
 114,
 120,
 120,
 120,
 120,
 210,
 210,
 210,
 210,
 210,
 210,
 210,
 210,
 210,
 210,
 228,
 228,
 228,
 228,
 228,
 228,
 228,
 228,
 228,
 240,
 240,
 240,
 240,
 240,
 246,
 246,
 246,
 246,
 246,
 246,
 282,
 282,
 282,
 282,
 282,
 282,
 282,
 282,
```

```
 360,
 360,
 360,
 360,
 372,
 372,
 372,
 450,
 450,
 450,
 450,
 450,
 450,
 450,
 450,
 522,
 522,
 588,
 588,
 588,
 588,
 612,
 612,
 612,
 666,
 666,
 666,
 666,
 666,
 672,
 672,
 672,
 744,
 864,
 864,
 864,
 1056,
 1056,
 1056,
 1056,
 1134,
 1134,
 1134,
 1134,
 1176,
 1176,
 1176,
 1176,
 1230,
 1230,
 1230]
```

```
def separate(X,a,m):
      holder = []
      for i in range(0,len(X)):
            if (i%m) == (a%m):
                  holder.append(X[i])
      output = ''.join(holder)
      return output
```

```
 Y = separate(X,0,7)
```

```
Y
```

```
    'RAGPSPTTMTVATBGCEJNGZPMULABUATJMFKXUSXHPRHNBPDXTXLWEYPNVSUKVJPGLSPC\
    EVGTORLSGXLBZSRWLJVCUVAWMOKWZQMWKVLJBHVIAFMAQAOYMOPEOYLWFPSTXAIILMMS\
    VMZJACYXOYXOTMAYMACXAJAOGIMJBWRXLYVWNIUXGHNEWNADGYUJHFVKLJRFGLZJXSKP\
    W'
```

```
```