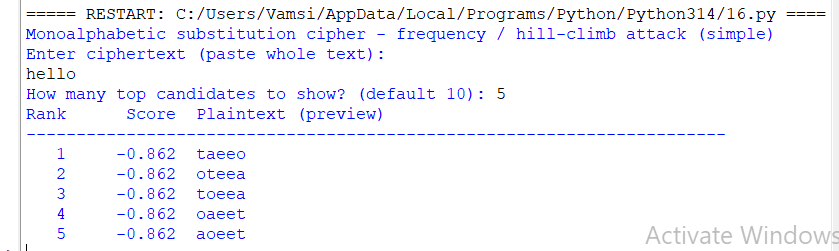
program that can perform a letter frequency attack on any monoalphabetic substitution cipher without human intervention. Your software should produce possible plaintexts in rough order of likelihood. It would be good if your user interface allowed the user to specify “give me the top 10 possible plaintexts.”

#!/usr/bin/env python3

"""

Simple automated letter-frequency / hill-climbing attack on a monoalphabetic substitution cipher.

Produces candidate plaintexts ordered by a rough likelihood score.

"""

import random

import math

from collections import Counter

# Most common English letter frequency order (for initial mapping heuristic)

EN\_ORDER = "ETAOINSHRDLCUMWFGYPBVKJXQZ"

# A short list of common English words used to score plaintexts.

COMMON\_WORDS = [

"the", "be", "to", "of", "and", "a", "in", "that", "have", "I",

"it", "for", "not", "on", "with", "he", "as", "you", "do", "at",

"this", "but", "his", "by", "from", "they", "we", "say", "her", "she",

]

def clean\_text(s):

return ''.join(ch for ch in s.upper() if ch.isalpha() or ch.isspace())

def apply\_key(ciphertext, key\_map):

"""Decrypt ciphertext using key\_map: dict mapping uppercase cipher char -> uppercase plain char."""

out = []

for ch in ciphertext:

if ch.isalpha():

up = ch.upper()

dec = key\_map.get(up, '?')

# preserve case

out.append(dec if ch.isupper() else dec.lower())

else:

out.append(ch)

return ''.join(out)

def freq\_key\_initial(ciphertext):

"""Make an initial key by mapping ciphertext letter frequencies to English frequency order."""

letters = [c for c in ciphertext.upper() if c.isalpha()]

cnt = Counter(letters)

# Sort ciphertext letters by frequency descending; for ties randomize order a bit

items = sorted(cnt.items(), key=lambda x: (-x[1], random.random()))

cipher\_order = ''.join([c for c,\_ in items])

# Fill missing letters with remaining alphabet letters in random order

remaining = [c for c in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" if c not in cipher\_order]

random.shuffle(remaining)

cipher\_order = cipher\_order + ''.join(remaining)

# map ciphertext letter -> guessed plaintext letter using EN\_ORDER

key\_map = {}

for i, c in enumerate(cipher\_order):

key\_map[c] = EN\_ORDER[i]

return key\_map

def random\_key():

letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")

random.shuffle(letters)

return {c: letters[i] for i, c in enumerate("ABCDEFGHIJKLMNOPQRSTUVWXYZ")}

def score\_plaintext(pt):

"""

Score plaintext: higher is better.

Combines:

- count of common English words found (weighted)

- negative chi-square letter-frequency distance (so closer to English gets higher)

"""

s = pt.lower()

# word match score

wordscore = 0.0

for w in COMMON\_WORDS:

# count occurrences (allow overlaps not necessary)

count = s.count(" " + w + " ") # whole word approx

# also check start and end boundaries roughly

if s.startswith(w + " "):

count += 1

if s.endswith(" " + w):

count += 1

wordscore += count

# small bonus for spaces (more plausible English has spaces)

space\_bonus = s.count(' ')

# letter frequency chi-square (lower is better) -> convert to positive contribution

en\_freq = {

'A': 8.167, 'B': 1.492, 'C': 2.782, 'D': 4.253, 'E': 12.702, 'F': 2.228,

'G': 2.015, 'H': 6.094, 'I': 6.966, 'J': 0.153, 'K': 0.772, 'L': 4.025,

'M': 2.406, 'N': 6.749, 'O': 7.507, 'P': 1.929, 'Q': 0.095, 'R': 5.987,

'S': 6.327, 'T': 9.056, 'U': 2.758, 'V': 0.978, 'W': 2.361, 'X': 0.150,

'Y': 1.974, 'Z': 0.074

}

letters = [c.upper() for c in pt if c.isalpha()]

N = len(letters)

if N == 0:

return -1e9

obs = Counter(letters)

chi2 = 0.0

for L, pct in en\_freq.items():

expected = pct / 100.0 \* N

observed = obs.get(L, 0)

exp = expected if expected > 1e-8 else 1e-8

chi2 += (observed - exp) \*\* 2 / exp

# convert chi2 to positive contribution: smaller chi2 -> larger 'fit' value

chi\_score = -0.1 \* chi2

# final weighted sum

total = (wordscore \* 3.0) + (space\_bonus \* 0.5) + chi\_score

return total

def neighbor\_swap\_key(key\_map):

"""Return a new key\_map with two plaintext letter assignments swapped (i.e., swap two mappings)."""

# represent key as list index by ciphertext letter A..Z

letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")

mapping = [key\_map[c] for c in letters]

i, j = random.sample(range(26), 2)

mapping[i], mapping[j] = mapping[j], mapping[i]

return {letters[k]: mapping[k] for k in range(26)}

def hill\_climb(ciphertext, init\_key, max\_iters=2000, improve\_threshold=0):

"""Perform hill-climbing starting from init\_key, return best (key\_map, plaintext, score)."""

best\_key = init\_key

best\_plain = apply\_key(ciphertext, best\_key)

best\_score = score\_plaintext(clean\_text(best\_plain))

current\_key = best\_key

current\_score = best\_score

for it in range(max\_iters):

cand\_key = neighbor\_swap\_key(current\_key)

cand\_plain = apply\_key(ciphertext, cand\_key)

cand\_score = score\_plaintext(clean\_text(cand\_plain))

if cand\_score > current\_score:

current\_key = cand\_key

current\_score = cand\_score

# update global best

if current\_score > best\_score:

best\_key, best\_score = current\_key, current\_score

best\_plain = apply\_key(ciphertext, best\_key)

# small random restart inside climb to escape tiny traps

if it % 500 == 0 and random.random() < 0.15:

current\_key = neighbor\_swap\_key(current\_key)

current\_score = score\_plaintext(clean\_text(apply\_key(ciphertext, current\_key)))

return best\_key, best\_plain, best\_score

def attack\_monoalphabetic(ciphertext, top\_n=10, restarts=30, iters=2000):

"""

Try multiple restarts and keep top\_n unique plaintext candidates ordered by score.

restarts: how many random starts (combination of freq heuristic and random keys).

"""

ciphertext = ciphertext # keep original case/punctuation

candidates = []

seen\_plain = set()

for r in range(restarts):

# choose initial key: heuristic freq map with some randomization half the time, else random key

if random.random() < 0.7:

init\_key = freq\_key\_initial(ciphertext)

# random small perturbation

for \_ in range(5):

init\_key = neighbor\_swap\_key(init\_key)

else:

init\_key = random\_key()

\_, best\_plain, best\_score = hill\_climb(ciphertext, init\_key, max\_iters=iters)

# normalize plaintext to lower for uniqueness

norm = clean\_text(best\_plain).strip()

if norm and norm not in seen\_plain:

seen\_plain.add(norm)

candidates.append((best\_score, best\_plain))

# sort by score descending (higher better)

candidates.sort(key=lambda x: -x[0])

return candidates[:top\_n]

def pretty\_print(cands):

print(f"{'Rank':>4} {'Score':>9} Plaintext (preview)")

print("-" \* 70)

for i, (score, plain) in enumerate(cands, start=1):

preview = plain if len(plain) <= 80 else plain[:77] + "..."

print(f"{i:>4} {score:9.3f} {preview}")

if \_\_name\_\_ == "\_\_main\_\_":

print("Monoalphabetic substitution cipher - frequency / hill-climb attack (simple)")

ct = input("Enter ciphertext (paste whole text):\n")

tn = input("How many top candidates to show? (default 10): ").strip()

try:

top\_n = int(tn) if tn else 10

except ValueError:

top\_n = 10

# reasonable defaults: more restarts -> better chance of success but slower

restarts = 40

iters = 2500

results = attack\_monoalphabetic(ct, top\_n=top\_n, restarts=restarts, iters=iters)

pretty\_print(results)