The Vigenere cipher

Rachid Hamadi, CSE, UNSW

COMP9021 Principles of Programming, Term 3, 2019

[1]:
```
from string import printable
from collections import Counter
from operator import itemgetter
from itertools import product
from re import split
```

Consider a message all written in capital letters. The *Caesar code* fixes a natural number n between 1 and 25 and shifts all letters n positions to the right, wrapping around when needed, leaving all other characters such as spaces unchanged. For instance, the message ALEA JACTA EST becomes

- with $n = 1$: BMFB KBDUB FTU
- with $n = 2$: CNGC LCEVC GUV
- ...
- with $n = 10$: KVOK TKMDK OCD
- ...
- with $n = 25$: ZKDZ IZBSZ DRS

The *Vigenere cipher* offers a generalisation as follows. Choose a word written all in uppercase letters as the key, for instance, RAW. Since RAW consists of 3 letters, write the letters that make up the message over 3 columns:
ALE
AJA
CTA
EST

As R, A and W are the 18th, 1st and 23rd letters of the alphabet, respectively, all letters in the first column are shifted 18 positions to the right, all letters in the second column are shifted 1 position to the right, and all letters in the third column are shifted 23 positions to the right, wrapping around when needed:
SMB
SKX
UUX
WTQ

This results in the encrypted message: SMBS KXUUX WTQ

Let us not distinguish between (capital) letters and other characters, but rather let the Vigenere cipher encrypt all characters in a message with a key that itself can be any sequence of characters. The `string` module defines a string consisting of all printable ASCII characters; it could be a good candidate for which characters to allow both in message and key:

[2]:
```
printable
```

[2]:
```
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:
;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

`'\r'` is the carriage return character:

[3]:
```
print('00000\r11')
```

```
11000
```

ASCII characters can be represented as `\x` followed by their ASCII code in the form of two hexadecimal digits:

[4]:
```
ord('c'), f"{ord('c'):x}", '\x63'
# f-strings cannot embed backslashes,
# so we instead use the format string method.
ord('\n'), '{:x}'.format(ord('\n')), '\x0a'
```

[4]:
```
(99, '63', 'c')
```

[4]:
```
(10, 'a', '\n')
```

`'\x0b'` and `'\x0c'` are the vertical tab and form feed characters, respectively; Jupyter ignores them. For code run from the command line, `'\x0b'` and `'\x0c'` behave the same: the next character follows the previous one but on the next line (form feed is meant to force a printer to move to the next sheet of paper). Both `'\x0b'` and `'\x0c'` actually have escaped characters as alternatives, namely, `'\v'` and `'\f'`, respectively:

[5]:
```
'\v', '\f'
```

[5]:
```
('\x0b', '\x0c')
```

It is sensible to ignore the last 3 characters in `printable` and accept all others as the possible constituents of messages and keys:

[6]:
```
characters = printable[: -3]
len(characters)
```

[6]:
```
97
```

Any character in `characters` can be used in keys and determine by how much to shift characters in messages; the values of the shifts are given by the positions in `characters` of the characters that make up the key. It would be inefficient to look for the position of a given character in `characters` every time it is needed: for instance, if a key was 7 characters long and started with `'G'`, it would be inefficient to scan `characters` left to right from the beginning, to find `'G'` at position 42 and infer that the first, eight, fiftieth… characters in the message to encrypt should be shifted 42 positions to the right, wrapping around if needed. We define a dictionary with the characters in `characters` as keys, and their positions in `characters` as values, so the latter can be obtained from the former in constant time:

[7]:
```
shifts = {characters[i]: i for i in range(len(characters))}
print(shifts)
```

```
{'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9,
'a': 10, 'b': 11, 'c': 12, 'd': 13, 'e': 14, 'f': 15, 'g': 16, 'h': 17, 'i': 18,
```

```
'j': 19, 'k': 20, 'l': 21, 'm': 22, 'n': 23, 'o': 24, 'p': 25, 'q': 26, 'r': 27,
's': 28, 't': 29, 'u': 30, 'v': 31, 'w': 32, 'x': 33, 'y': 34, 'z': 35, 'A': 36,
'B': 37, 'C': 38, 'D': 39, 'E': 40, 'F': 41, 'G': 42, 'H': 43, 'I': 44, 'J': 45,
'K': 46, 'L': 47, 'M': 48, 'N': 49, 'O': 50, 'P': 51, 'Q': 52, 'R': 53, 'S': 54,
'T': 55, 'U': 56, 'V': 57, 'W': 58, 'X': 59, 'Y': 60, 'Z': 61, '!': 62, '"': 63,
'#': 64, '$': 65, '%': 66, '&': 67, "'": 68, '(': 69, ')': 70, '*': 71, '+': 72,
',': 73, '-': 74, '.': 75, '/': 76, ':': 77, ';': 78, '<': 79, '=': 80, '>': 81,
'?': 82, '@': 83, '[': 84, '\\': 85, ']': 86, '^': 87, '_': 88, '`': 89, '{':
90, '|': 91, '}': 92, '~': 93, ' ': 94, '\t': 95, '\n': 96}
```

Let us give an example of a message and its encryption with the key `'Take it Easy!'` (we will see below how the encryption has been computed, now it is given and supposed to have been properly computed). `'Take it Easy!'` is a key of length 13, so we align the key, the message displayed over 13 columns, and the encrypted message displayed over 13 columns. As the message length is not a multiple of 13, the last lines of message and encrypted message are less than 13 characters long; formatted strings can be given a field width to display a string, padding with spaces to the right for shorter strings, ignoring the field width for longer strings:

[8]:
```
print('|'.join(f'{w:4}' for w in ('Alice', 'was', 'beginning', 'to', 'get',
                                  'very', 'tired'
                                 )
              )
     )
```

```
Alice|was |beginning|to  |get |very|tired
```

Next to key and message, we display the ASCII codes of the characters in key and message, similarly aligned. We observed that `characters` has 97 characters.

- The first character in `key`, namely, `'T'`, has position 55 in `characters`. The first character in `message`, namely `'A'`, has position 36 in `characters`; it is therefore encrypted in `encrypted_message` as the character that in `characters`, has position $(55 + 36) \bmod 97 = 91 \bmod 97 = 91$, namely, `'|'`.
- The fifth character in `key`, namely a space, has position 94 in `characters`. The fifth character in `message`, namely `'e'`, has position 14 in `characters`; it is therefore encrypted in `encrypted_message` as the character that in `characters`, has position $(94 + 14) \bmod 97 = 108 \bmod 97 = 11$, namely, `'b'`.

[9]:
```
message = ("Alice was beginning to get very tired of sitting by her "
           "sister on the bank, and of having nothing to do: once or "
           "twice she had peeped into the book her sister was reading, "
           "but it had no pictures or conversations in it, 'and what "
           "is the use of a book,' thought Alice, 'without pictures or "
           "conversations?'"
          )

encrypted_message = ('|vCqbfZ7\'7DM;,xHwkyqq#7IM|QFyFvfWf&oFv])7MwqLL'
                     'kU7D\'X+oLbpAVqSBpW\\QDBs|tDkY@pI\\\'7It|zDsWxI'
                     'v\\<DBwkyqq#7FWGQyHqbfRoBDYQ-(7MvbfK7R7RM/=oxbf'
```

```
                     'FWlBDJMX%yIy|zHoBCK!|(BhK7KqoSkFQ\\*@hprLqf(7JI'
                     '.QxIbmAFq)BG!X<BhqlFYb&CC"=<xMbfFqf(@p5+;nhKesW'
                     '|WCp"<(7OGbfRcBkpJ]<u~?|LKl)qJ"X|vCqb|q$+sVP]\\'
                     'DhDfuWr&oUv]?7wCkNHo\'kVQ];C5?'
                    )


key = 'Take it Easy!'
print(f'{key}' + '  ' + ' '.join(f'{shifts[c]:2}' for c in key))
print()
print('\n'.join(f'{message[i * 13 : (i + 1) * 13]:13}  ' +
                ' '.join(f'{shifts[c]:2}'
                              for c in message[i * 13 : (i + 1) * 13]
                        ) for i in range(len(message) // 13 + 1)
               )
     )
print()
print('\n'.join(f'{encrypted_message[i * 13 : (i + 1) * 13]:13}  ' +
                ' '.join(f'{shifts[c]:2}'
                              for c in encrypted_message[i * 13 : (i + 1) * 13]
                        ) for i in range(len(encrypted_message) // 13 + 1)
               )
     )
```

```
Take it Easy!  55 10 20 14 94 18 29 94 40 10 28 34 62

Alice was beg  36 21 18 12 14 94 32 10 28 94 11 14 16
inning to get  18 23 23 18 23 16 94 29 24 94 16 14 29
 very tired o  94 31 14 27 34 94 29 18 27 14 13 94 24
f sitting by   15 94 28 18 29 29 18 23 16 94 11 34 94
her sister on  17 14 27 94 28 18 28 29 14 27 94 24 23
 the bank, an  94 29 17 14 94 11 10 23 20 73 94 10 23
d of having n  13 94 24 15 94 17 10 31 18 23 16 94 23
othing to do:  24 29 17 18 23 16 94 29 24 94 13 24 77
 once or twic  94 24 23 12 14 94 24 27 94 29 32 18 12
e she had pee  14 94 28 17 14 94 17 10 13 94 25 14 14
ped into the   25 14 13 94 18 23 29 24 94 29 17 14 94
book her sist  11 24 24 20 94 17 14 27 94 28 18 28 29
er was readin  14 27 94 32 10 28 94 27 14 10 13 18 23
g, but it had  16 73 94 11 30 29 94 18 29 94 17 10 13
 no pictures   94 23 24 94 25 18 12 29 30 27 14 28 94
or conversati  24 27 94 12 24 23 31 14 27 28 10 29 18
ons in it, 'a  24 23 28 94 18 23 94 18 29 73 94 68 10
nd what is th  23 13 94 32 17 10 29 94 18 28 94 29 17
e use of a bo  14 94 30 28 14 94 24 15 94 10 94 11 24
ok,' thought   24 20 73 68 94 29 17 24 30 16 17 29 94
Alice, 'witho  36 21 18 12 14 73 94 68 32 18 29 17 24
```

```
ut pictures o  30 29 94 25 18 12 29 30 27 14 28 94 24
r conversatio  27 94 12 24 23 31 14 27 28 10 29 18 24
ns?'           23 28 82 68

|vCqbfZ7'7DM;  91 31 38 26 11 15 61  7 68  7 39 48 78
,xHwkyqq#7IM|  73 33 43 32 20 34 26 26 64  7 44 48 91
QFyFvfWf&oFv]  52 41 34 41 31 15 58 15 67 24 41 31 86
)7MwqLLkU7D'X  70  7 48 32 26 47 47 20 56  7 39 68 59
+oLbpAVqSBpW\  72 24 47 11 25 36 57 26 54 37 25 58 85
QDBs|tDkY@pI\  52 39 37 28 91 29 39 20 60 83 25 44 85
'7It|zDsWxIv\  68  7 44 29 91 35 39 28 58 33 44 31 85
<DBwkyqq#7FWG  79 39 37 32 20 34 26 26 64  7 41 58 42
QyHqbfRoBDYQ-  52 34 43 26 11 15 53 24 37 39 60 52 74
(7MvbfK7R7RM/  69  7 48 31 11 15 46  7 53  7 53 48 76
=oxbfFWlBDJMX  80 24 33 11 15 41 58 21 37 39 45 48 59
%yIy|zHoBCK!|  66 34 44 34 91 35 43 24 37 38 46 62 91
(BhK7KqoSkFQ\  69 37 17 46  7 46 26 24 54 20 41 52 85
*@hprLqf(7JI.  71 83 17 25 27 47 26 15 69  7 45 44 75
QxIbmAFq)BG!X  52 33 44 11 22 36 41 26 70 37 42 62 59
<BhqlFYb&CC"=  79 37 17 26 21 41 60 11 67 38 38 63 80
<xMbfFqf(@p5+  79 33 48 11 15 41 26 15 69 83 25  5 72
;nhKesW|WCp"<  78 23 17 46 14 28 58 91 58 38 25 63 79
(7OGbfRcBkpJ]  69  7 50 42 11 15 53 12 37 20 25 45 86
<u~?|LKl)qJ"X  79 30 93 82 91 47 46 21 70 26 45 63 59
|vCqb|q$+sVP]  91 31 38 26 11 91 26 65 72 28 57 51 86
\DhDfuWr&oUv]  85 39 17 39 15 30 58 27 67 24 56 31 86
?7wCkNHo'kVQ]  82  7 32 38 20 49 43 24 68 20 57 52 86
;C5?           78 38  5 82
```

Let us write a function, `encrypt()`, to encrypt a message, and a function, `decrypt()`, to decrypt an encrypted message, both with a key K provided as first argument, the message being provided as second argument. Both functions perform almost identically:

- To encrypt a message M, one goes over each character in M, determine its index i modulo the length of K, and shifts it to the right in `characters` by n with n the position in `characters` of the character in K of index i, wrapping around if needed.
- To decrypt an encrypted message E, one goes over each character in E, determine its index i modulo the length of K, and shifts it to the left in `characters` by n with n equal to the position in `characters` of the character in K of index i, wrapping around if needed.

It is therefore natural to define an auxiliary function, `encrypt_or_decrypt()` called by both `encrypt()` and `decrypt()`, being directed by either whether shifting should be "to the right" or "to the left", which corresponds to adding or subtracting positions:

[10]:
```python
def encrypt(key, text):
    return encrypt_or_decrypt(key, text, 1)

def decrypt(key, text):
    return encrypt_or_decrypt(key, text, -1)
```

```python
def encrypt_or_decrypt(key, text, mode):
    return ''.join(characters[(shifts[text[i]]
                               + shifts[key[i % len(key)]] * mode
                              ) % len(shifts)
                             ] for i in range(len(text))
                  )

encrypt(key, message)
print()
decrypt(key, encrypted_message)
print()
encrypt(key, decrypt(key, encrypted_message))
print()
decrypt(key, encrypt(key, message))
```

[10]:

```
'|vCqbfZ7\'7DM;,xHwkyqq#7IM|QFyFvfWf&oFv])7MwqLLkU7D\'X+oLbpAVqSBpW\\QDBs|tDkY@p
I\\\'7It|zDsWxIv\\<DBwkyqq#7FWGQyHqbfRoBDYQ-(7MvbfK7R7RM/=oxbfFWlBDJMX%yIy|zHoBC
K!|(BhK7KqoSkFQ\\*@hprLqf(7JI.QxIbmAFq)BG!X<BhqlFYb&CC"=<xMbfFqf(@p5+;nhKesW|WCp
"<(7OGbfRcBkpJ]<u~?|LKl)qJ"X|vCqb|q$+sVP]\\DhDfuWr&oUv]?7wCkNHo\'kVQ];C5?'
```

[10]:

```
"Alice was beginning to get very tired of sitting by her sister on the bank, and
of having nothing to do: once or twice she had peeped into the book her sister
was reading, but it had no pictures or conversations in it, 'and what is the use
of a book,' thought Alice, 'without pictures or conversations?'"
```

[10]:


```
'|vCqbfZ7\'7DM;,xHwkyqq#7IM|QFyFvfWf&oFv])7MwqLLkU7D\'X+oLbpAVqSBpW\\QDBs|tDkY@p
I\\\'7It|zDsWxIv\\<DBwkyqq#7FWGQyHqbfRoBDYQ-(7MvbfK7R7RM/=oxbfFWlBDJMX%yIy|zHoBC
K!|(BhK7KqoSkFQ\\*@hprLqf(7JI.QxIbmAFq)BG!X<BhqlFYb&CC"=<xMbfFqf(@p5+;nhKesW|WCp
"<(7OGbfRcBkpJ]<u~?|LKl)qJ"X|vCqb|q$+sVP]\\DhDfuWr&oUv]?7wCkNHo\'kVQ];C5?'
```


[10]:


```
"Alice was beginning to get very tired of sitting by her sister on the bank, and
of having nothing to do: once or twice she had peeped into the book her sister
was reading, but it had no pictures or conversations in it, 'and what is the use
of a book,' thought Alice, 'without pictures or conversations?'"
```


Let us now tackle the task of breaking the code of an encrypted message E, that is, discovering the key K that was used to encrypt a message M into E, and decrypting E into M with K. A program can only tentatively suggest candidates for K that the user would then have to validate. Also, the program might not always succeed, especially when the encrypted message is short. Let us fix an upper bound on the length of K, the first of a number of parameters we will use in our heuristics:

[11]:
```
max_key_length = 16
```

With that value, the number of possible keys is huge:

[12]:
```
sum(len(characters) ** i for i in range(1, 17))
```

[12]: 62065212901958868055012327674640

If the message is long enough, it is likely to contain many *n*-grams (i.e., *n* consecutive characters) that occur many times, and less likely but still likely enough, *n*-grams with many vertically aligned occurrences. For instance, "the", "ing", "able" could be *n*-grams (with *n* equal to 3, 3 and 4, respectively) some of whose occurrences in a message could be aligned:

```
. . . . t h e . .
. . . . . . . . .
. a b l e . . . .
. . . . . . i n g
. . . . . . . . .
. . . . . . i n g
. . . . . . . . .
. . . . . . . . .
. . . . . . . . .
. . . . t h e . .
. a b l e . . . .
. . . . t h e . .
. . . . . . . . .
. . . . . . . . .
. . . . . . i n g
```

All vertically aligned occurrences of a given *n*-gram are identically encrypted. Hence one can look for *n*-grams in the encrypted message that occur many times, and assume that they have a significant chance to correspond to *n*-grams in the original message that happen to be vertically aligned. Suppose for instance that %)O occurs many times in the encrypted message. Suppose that the second occurrence of %)O is 40 positions to the right of the first occurrence of %)O. Conjecturing that both occurrences of %)O are aligned, and assuming that the key is at most 16 characters long, there are 4 options (writing ~ for a slot for a character):

- The key is 4 characters long and the second occurrence of %)O is 10 lines below the first occurrence of %)O, e.g.:
  ```
  ~ % ) O
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ ~ ~ ~
  ~ % ) O
  ```
- The key is 5 characters long and the second occurrence of %)O is 8 lines below the first occurrence of %)O, e.g.:

~ %) O ~
~ ~ ~ ~ ~
~ ~ ~ ~ ~
~ ~ ~ ~ ~
~ ~ ~ ~ ~
~ ~ ~ ~ ~
~ ~ ~ ~ ~
~ ~ ~ ~ ~
~ %) O ~

- The key is 8 characters long and the second occurrence of %)O is 5 lines below the first occurrence of %)O, e.g.:
  ~ ~ %) O ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ %) O ~ ~ ~
- The key is 10 characters long and the second occurrence of %)O is 4 lines below the first occurrence of %)O, e.g.:
  ~ ~ ~ %) O ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
  ~ ~ ~ %) O ~ ~ ~ ~

In other words, the conjecture that two successive occurrences of an n-gram are aligned leads to the conclusion that the length of the key is a divisor at least equal to n and at most equal to the value of `max_key_length` of the distance between (the start of) both occurrences. The following function returns a generator expression for the relevant divisors of its first argument, namely, those of its factors that are at least equal to its second argument and at most equal to the value of `max_key_length`:

[13]:
```
def factors_from_successive_n_grams(num, min_value):
    return (i for i in range(min_value, max_key_length + 1) if num % i == 0)

tuple(factors_from_successive_n_grams(40, 3))
```

[13]: `(4, 5, 8, 10)`

`factors_from_successive_n_grams()` returns a generator expression; each factor will be retrieved on demand, e.g., by calls to `next()`, or thanks to a `for` statement. We would like to generate on demand not just the relevant factors of the distance between two successive occurrences of a given n-gram, but the relevant factors of the distances of all pairs of successive occurrences of all n-grams (for the chosen values of n). To this aim, we introduce another mechanism, namely, generator functions. A generator function is a function with `yield` statements in its body. A `yield` statement can, but does not have to, be followed by an expression. In case an expression follows, the value of that expression is provided when the `yield` statement is executed; otherwise, `None` is provided. In any case, execution of the generator function then stops, waiting to be resumed, possibly, on demand:

[14]:
```python
def f(n):
    print('Before yielding', n)
    yield n
    print('Before yielding', n + 1)
    yield n + 1
    print('Before yielding None')
    yield
    print('Before yielding', n + 2)
    yield n + 2
    print('After yielding', n + 2)

F = f(10)
F

next(F)
next(F)
next(F)
next(F)
next(F)
```

[14]:
```
<generator object f at 0x109befde0>
```

```
Before yielding 10
```

[14]:
```
10
```

```
Before yielding 11
```

[14]:
```
11
```

```
Before yielding None
Before yielding 12
```

[14]:
```
12
```

```
After yielding 12
```

```
␣
↪---------------------------------------------------------------------------

    StopIteration                             Traceback (most recent call␣
↪last)

    <ipython-input-14-602587f40ca4> in <module>
     17 next(F)
     18 next(F)
```

```
---> 19 next(F)

StopIteration:
```

A generator function can also contain `return` statements:

[15]:
```
def f(n):
    print('Before yielding', n)
    yield n
    print('Before yielding', n + 1)
    yield n + 1
    print('Before returning')
    return
    # Won't be printed out
    print('After returning')
    # Won't be yielded
    yield n + 2

F = f(10)
F

next(F)
next(F)
next(F)
next(F)
next(F)
```

[15]:
```
<generator object f at 0x109cc6138>
```

```
Before yielding 10
```

[15]:
```
10
```

```
Before yielding 11
```

[15]:
```
11
```

```
Before returning
```

```
␣
↪---------------------------------------------------------------------------

StopIteration                             Traceback (most recent call␣
↪last)
```

```
<ipython-input-15-5a875d3db4b3> in <module>
     16 next(F)
     17 next(F)
---> 18 next(F)
     19 next(F)
     20 next(F)


StopIteration:
```

A generator function can yield values yielded by another generator function or any other iterable:

[16]:
```python
def f():
    yield 1
    yield 2

def g():
    yield 0
    for x in f():
        yield x
    for x in range(3, 6):
        yield x
    for x in (x for x in [6, 7, 8]):
        yield x
    for x in [9, 10]:
        yield x

G = g()
for x in G:
    print(x, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9 10
```

In such a context, `yield from` statements offer a more concise alternative:

[17]:
```python
def f():
    yield 1
    yield 2

def g():
    yield 0
    yield from f()
    yield from range(3, 6)
    yield from (x for x in [6, 7, 8])
    yield from [9, 10]

list(g())
```

[17]: `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

Let us get back to the problem of discovering successive occurrences of n-grams. As part of our heuristics, let us set the sizes of n-grams to look for to the default values of 3, 4 and 5:

[18]:
```
n_gram_range = range(3, 6)
```

To look for substrings within a string, we can use either `find()` or `index()` `str`'s methods. They are exactly the same, except that `find()` returns `-1` whereas `index()` raises a `ValueError` exception when they fail to find their argument as a substring. Both methods take an optional second argument and a second optional third argument to limit the search within a narrower range than the full string to which the method is applied:

[19]:
```
'123123'.find('2')
'123123'.find('2', 2)
'123123'.find('2', 2, 4)
```

[19]: `1`

[19]: `4`

[19]: `-1`

When we find a pair of occurrences of a 5-gram, we are guaranteed to find as well 2 pairs of occurrences of 4-grams, and 3 pairs of occurrences of 3-grams, but there does not seem to be a good reason to prevent the redundancy (in particular because between two consecutive occurrences of an n-gram that extends a substring s, there can be other occurrences of s in between. The following code fragment takes as example the first 91 characters of `message` (so the first 7 lines of `message` displayed over 13 columns as above) and collects in a list the relevant factors of the distances between all successive occurrences of all 3, 4 and 5-grams in this text. We trace execution and display, in particular, the contexts in which two successive occurrences of a 3, 4 or 5-gram have been detected:

[20]:
```
text = message[: 91]

relevant_factors = []
for n in (n_gram_range):
    print(f'Looking for consecutive occurrences of {n}-grams:')
    for i in range(len(text) - 2 * n + 1):
        n_gram = text[i : i + n]
        j = text.find(n_gram, i + n)
        if j != -1:
            print(f'  Found "{n_gram}" at indexes {i} and {j}\n'
                  f'  {text[i : j + n]}\n'
                  f'  Distance: {j - i}. Adding as factors:'
                  f' {tuple(factors_from_successive_n_grams(j - i, n))}\n'
                 )
            relevant_factors.extend(factors_from_successive_n_grams(j - i, n))

print(relevant_factors)
```

```
Looking for consecutive occurrences of 3-grams:
  Found "ing" at indexes 16 and 45
```

```
  ing to get very tired of sitting
  Distance: 29. Adding as factors: ()

  Found "ng " at indexes 17 and 46
  ng to get very tired of sitting
  Distance: 29. Adding as factors: ()

  Found "d o" at indexes 36 and 78
  d of sitting by her sister on the bank, and o
  Distance: 42. Adding as factors: (3, 6, 7, 14)

  Found " of" at indexes 37 and 79
   of sitting by her sister on the bank, and of
  Distance: 42. Adding as factors: (3, 6, 7, 14)

  Found "of " at indexes 38 and 80
  of sitting by her sister on the bank, and of
  Distance: 42. Adding as factors: (3, 6, 7, 14)

  Found " si" at indexes 40 and 55
   sitting by her si
  Distance: 15. Adding as factors: (3, 5, 15)

  Found "ing" at indexes 45 and 86
  ing by her sister on the bank, and of having
  Distance: 41. Adding as factors: ()

  Found "ng " at indexes 46 and 87
  ng by her sister on the bank, and of having
  Distance: 41. Adding as factors: ()

  Found "er " at indexes 53 and 60
  er sister
  Distance: 7. Adding as factors: (7,)

Looking for consecutive occurrences of 4-grams:
  Found "ing " at indexes 16 and 45
  ing to get very tired of sitting
  Distance: 29. Adding as factors: ()

  Found "d of" at indexes 36 and 78
  d of sitting by her sister on the bank, and of
  Distance: 42. Adding as factors: (6, 7, 14)

  Found " of " at indexes 37 and 79
   of sitting by her sister on the bank, and of
  Distance: 42. Adding as factors: (6, 7, 14)
```

```
  Found "ing " at indexes 45 and 86
  ing by her sister on the bank, and of having
  Distance: 41. Adding as factors: ()

Looking for consecutive occurrences of 5-grams:
  Found "d of " at indexes 36 and 78
  d of sitting by her sister on the bank, and of
  Distance: 42. Adding as factors: (6, 7, 14)

[3, 6, 7, 14, 3, 6, 7, 14, 3, 6, 7, 14, 3, 5, 15, 7, 6, 7, 14, 6, 7, 14, 6, 7,
14]
```

It turns out that in these first seven lines, we do not get a single factor of 13: no 3-, 4- or 5-gram has two aligned successive occurrences. The search for successive occurrences of n-grams is anyway meant to be done in the encrypted message, and results are different; we used the original message for a friendlier illustration of the technique.

The following generator function adapts the previous code to, for an arbitrary text passed as argument, not collect in a list the relevant factors of the distances between all successive occurrences of all n-grams in the text (for the chosen values of n), but have a mechanism to return them on demand:

[21]:
```python
def all_collected_factors(text):
    for n in n_gram_range:
        for i in range(len(text) - 2 * n + 1):
            j = text.find(text[i : i + n], i + n)
            if j != -1:
                yield from factors_from_successive_n_grams(j - i, n)

print(list(all_collected_factors(encrypted_message)))
```

```
[4, 5, 10, 13, 4, 5, 10, 13, 4, 5, 10, 13, 4, 8, 13, 3, 6, 13, 3, 6, 13, 3, 6,
13, 3, 6, 13, 3, 6, 13, 5, 10, 13, 3, 6, 13, 3, 13, 4, 5, 10, 13, 4, 5, 10, 13,
6, 13, 6, 13, 6, 13, 6, 13, 5, 10, 13, 6, 13, 6, 13, 6, 13]
```

Arguably, the more frequent a factor is, the more likely it is to be equal to the length of the key that has been used to encrypt the message. We therefore want to rank, from most frequent to least frequent, all factors that have been collected for all consecutive pairs of n-grams. The first step is to count each factor. Thanks to the `Counter` class from the `collections` module, it is straightforward to get from a collection a (type of object that embeds a) dictionary whose keys and values are the members of the collection and the number of times they occur in the collection, respectively:

[22]:
```python
Counter([4, 5, 10, 13, 4, 5, 10, 13, 4, 5, 10, 13, 4, 8, 13])
```

[22]:
```
Counter({4: 4, 5: 3, 10: 3, 13: 4, 8: 1})
```

Factors should be ranked from the most frequent ones to the least frequent ones, but what to do when two factors have equal counts? E.g., what if both 8 and 16 occur 6,811 times? (This is the case when encrypting the text in `carroll.txt` with the key `'0123456789ABCDEF'`.) It seems probable that if the key length was 8 rather than 16, then we would have more n-grams that reveal vertical alignments over 8 columns than n-grams that reveal vertical alignments over 16 columns (actually, not having a single extra factor of 8 strongly suggests that the key length is 16 indeed).

So we decide to rank the divisors from most frequent ones to least frequent ones, and for a given count, from largest ones to smallest ones. So the key of the sorting method needs to take into account first the values of the dictionary embedded in the counter, and then the keys. We can retrieve from a dictionary (be it embedded in a counter) keys only, values only, or both keys and values, in the form of list-like objects:

[23]:
```
D = {4: 4, 5: 3, 10: 3, 13: 4, 8: 1}
D.keys()
D.values()
D.items()
```

[23]: `dict_keys([4, 5, 10, 13, 8])`

[23]: `dict_values([4, 3, 3, 4, 1])`

[23]: `dict_items([(4, 4), (5, 3), (10, 3), (13, 4), (8, 1)])`

The third way of sorting the items of a dictionary offers the solution to ranking properly our collected factors:

[24]:
```
D = {4: 4, 5: 3, 10: 3, 13: 4, 8: 1}

sorted(D.items(), reverse=True)
[x[0] for x in sorted(D.items(), reverse=True)]

sorted(D.items(), key=lambda x: x[1], reverse=True)
[x[0] for x in sorted(D.items(), key=lambda x: x[1], reverse=True)]

sorted(D.items(), key=lambda x: (x[1], x[0]), reverse=True)
[x[0] for x in sorted(D.items(), key=lambda x: (x[1], x[0]), reverse=True)]
```

[24]: `[(13, 4), (10, 3), (8, 1), (5, 3), (4, 4)]`

[24]: `[13, 10, 8, 5, 4]`

[24]: `[(4, 4), (13, 4), (5, 3), (10, 3), (8, 1)]`

[24]: `[4, 13, 5, 10, 8]`

[24]: `[(13, 4), (4, 4), (10, 3), (5, 3), (8, 1)]`

[24]: `[13, 4, 10, 5, 8]`

Rather than using a lambda expression, we can also use `itemgetter()` from the `operator` module:

[25]:
```
L = ['A', 'B', 'C']
itemgetter(0)(L)
itemgetter(0, 2)(L)
itemgetter(2, 0, 1)(L)

D = {4: 4, 5: 3, 10: 3, 13: 4, 8: 1}
sorted(D.items(), key=itemgetter(1, 0), reverse=True)
[x[0] for x in sorted(D.items(), key=itemgetter(1, 0), reverse=True)]
```

[25]: `'A'`

[25]: ('A', 'C')

[25]: ('C', 'A', 'B')

[25]: [(13, 4), (4, 4), (10, 3), (5, 3), (8, 1)]

[25]: [13, 4, 10, 5, 8]

Having with `all_collected_factors()` a mechanism to generate some key lengths from most promising to least promising, we still would like to consider all possible key lengths between 1 and the value of `max_key_length`, starting with 1 (in case the message has actually been encrypted with the simple Caesar code), then those given by `all_collected_factors()`, and finally all others, from shortest to longest. This is the purpose of the following generator function:

[26]:
```python
def key_lengths_from_most_to_least_promising(text):
    yield 1
    key_lengths =\
            [x[0] for x in sorted(Counter(all_collected_factors(text)).items(),
                                  key=itemgetter(1, 0), reverse=True
                                 )
            ]
    yield from key_lengths
    yield from\
            (i for i in range(2, max_key_length + 1) if i not in key_lengths)

list(key_lengths_from_most_to_least_promising(encrypted_message))
```

[26]: [1, 13, 6, 10, 5, 3, 4, 8, 2, 7, 9, 11, 12, 14, 15, 16]

For all $n < 13$, the n-th character of `key` encrypts the n-th column of `message`, yielding the n-th column of `encrypted_message`, which can be decrypted back to the n-th column of `message`:

[27]:
```python
for n in range(13):
    decrypt(key[n], encrypted_message[n :: 13])
```

[27]: 'Ai fh do epbeg ooneoAurn'

[27]: 'lnv et to eor,nrnd klt s'

[27]: 'inesrhohnsdo o s u,i c?'

[27]: "ciri efich kwb c ws'cpo'"

[27]: 'enyts neei aupoihe ein'

[27]: ' g tibhg nhstinna t,cv'

[27]: 'w tisaa ohte cv toh te'

[27]: "atintnvtraorritei fo'ur"

[27]: 'sorgekio d eturti uwrs'

[27]: ' e r,n t tsa rs,sagiea'

[27]: 'bgdb gdwphidhea   htst'

[27]: "ee yoa oieesiast'tbth i"

[27]:
```
'gto nnn:ce tnd iaho ooo'
```

Compare with the result obtained when decrypting the n-th column of `encrypted_message` not with the right character (the n-th character in key), but with an arbitrary character:

[28]:
```python
for n in range(13):
    decrypt('m+2I|b~!AxZ=I', encrypted_message[n :: 13])
```

[28]:
```
'(1Oq;F+hgAj@pN::z[W@tNNh'
```

[28]:
```
'9WDYusb-\t*YP{ZWz]t~y%3*-'
```

[28]:
```
"g'w4RqM+7f(Z)}(f4nD0,;\nF"
```

[28]:
```
'4VD\\hhx&^\tLP23Ao#Qv]Z35l'
```

[28]:
```
"]Jt<v=\tT+.PbY5Lj'k0\tK/["
```

[28]:
```
'{Xd3GiD(/<:Q2pZD y4Pt|g'
```

[28]:
```
'DPU3"sHZhd Y<4%W<#GOZma'
```

[28]:
```
"?Pd,w9wZ\\*VF:{P9'01p3__"
```

[28]:
```
'K`$cYN!21k,SaL\t$p#q-avz'
```

[28]:
```
'?wmYH+BG3*.T,?!ADI9u"\\['
```

[28]:
```
'h(D}veM/ok>" n&A;veN}ko'
```

[28]:
```
'q,to#xz~gf[<8m^ZW(y&]}j'
```

[28]:
```
'Uj[f|-`:CH\tbFR[;s\\."oOR'
```

The columns derived from the correct key contain more letters. But is that all? Is the distribution of letters in a given column good, in some sense? A column consisting of nothing but q's, z's and j's would not appear as natural.

"etaoinshrdlcumwfgypbvkjxqz" is all lowercase letters ordered in decreasing frequency of use in English: "e" is most common, then comes "t", then "a"...

A good distribution of letters in a given column should be relatively consistent with the ordering of letters given by the previous sequence. The notion of etaoin score will turn this idea into a precise measure.

Consider the first column, "Ai fh do epbeg ooneoAurn". Let us convert all letters to lowercase and get a count of the number of occurrences of the letters that occur in the resulting string:

[29]:
```python
column = Counter(c.lower() for c in 'Ai fh do epbeg ooneoAurn' if c.isalpha())
print(column)
```

```
Counter({'o': 4, 'e': 3, 'a': 2, 'n': 2, 'i': 1, 'f': 1, 'h': 1, 'd': 1, 'p': 1,
'b': 1, 'g': 1, 'u': 1, 'r': 1})
```

Let us order the letters by decreasing frequency in `column`. When two letters α and β share the same count, let us be "pessimistic" about `column`'s quality and rank α before β if α occurs after β in "etaoinshrdlcumwfgypbvkjxqz". So "o" comes first with 4 occurrences, then comes "e" with 3 occurrences. Both "a" and "n" have 2 occurrences, but "a" is more frequent than "n" in English (they occur at index 2 and 5 in `'etaoinshrdlcumwfgypbvkjxqz'`, respectively), so we rank "n" before “a”. All other letters occur only once in `column`, with “p” least frequent in English, so ranked next, and “i” most frequent in English, so ranked last:

[30]:
```
ranked_letters_in_column = ''.join(
       sorted(column,
              key=lambda x: (column[x], 'etaoinshrdlcumwfgypbvkjxqz'.index(x)),
              reverse=True
             )
                                 )

ranked_letters_in_column
```

[30]: `'oenabpgfudrhi'`

The previous code can be improved by defining from `'etaoinshrdlcumwfgypbvkjxqz'` a dictionary thanks to which the index in that string of a lowercase letter can be more effectively retrieved, similarly to the way the dictionary `shifts` has been defined from the string `characters`:

[31]:
```
etaoin = {'etaoinshrdlcumwfgypbvkjxqz'[i]: i for i in range(26)}

ranked_letters_in_column = ''.join(
             sorted(column, key=lambda x: (column[x], etaoin[x]), reverse=True)
                                  )

ranked_letters_in_column
```

[31]: `'oenabpgfudrhi'`

The etaoin score of `ranked_letters_in_column` is actually a function of both `ranked_letters_in_column` and a nonzero natural number l at most equal to the length of this string: for such a number l, it is equal to the number of elements common to the initial segments of `'oenabpgfudrhi'` and `'etaoinshrdlcumwfgypbvkjxqz'` of length l. Let us display for l at most equal to 10, l itself, then those initial segments of length l, then the letters both segments have in common, then the number of those letters, so the corresponding etaoin score:

[32]:
```
for i in range(1, 11):
    s_1 = ranked_letters_in_column[: i]
    s_2 = 'etaoinshrdlcumwfgypbvkjxqz'[: i]
    s = ''.join(set(s_1) & set(s_2))
    print(f'{i:2} {s_1:10} {s_2:10} {s:9} {len(s)}')
```

```
 1 o          e                    0
 2 oe         et         e         1
 3 oen        eta        e         1
 4 oena       etao       aoe       3
 5 oenab      etaoi      aoe       3
 6 oenabp     etaoin     anoe      4
 7 oenabpg    etaoins    anoe      4
 8 oenabpgf   etaoinsh   anoe      4
 9 oenabpgfu  etaoinshr  anoe      4
10 oenabpgfud etaoinshrd dnaoe     5
```

Converting strings to sets and taking their intersections is not the most efficient approach here. We can instead look whether each the l first letters in `ranked_letters_in_column` is one of the first l characters in `'etaoinshrdlcumwfgypbvkjxqz'`. To get the etaoin score of `ranked_letters_in_column` for $l = 10$, we need to see how many times the `if` statement below is executed:

[33]:
```
for i in range(10):
    s = ranked_letters_in_column[i]
    print(f'{s}', end=' ')
    if etaoin[s] < 10:
        print('occurs', end=' ')
    else:
        print('does not occur', end=' ')
    print(f"in {'etaoinshrdlcumwfgypbvkjxqz'[: 10]}")
```

```
o occurs in etaoinshrd
e occurs in etaoinshrd
n occurs in etaoinshrd
a occurs in etaoinshrd
b does not occur in etaoinshrd
p does not occur in etaoinshrd
g does not occur in etaoinshrd
f does not occur in etaoinshrd
u does not occur in etaoinshrd
d occurs in etaoinshrd
```

This is easily done with the `sum()` function:

[34]:
```
sum(1 for i in range(10) if etaoin[ranked_letters_in_column[i]] < 10)
```

[34]: 5

Putting it all together in a function:

[35]:
```
def etaoin_score(text, length):
    letter_counts = Counter(c.lower() for c in text if c.isalpha())
    ranked_letters = sorted(letter_counts,
                            key=lambda x: (letter_counts[x], etaoin[x]),
                            reverse=True
                           )
    return sum(1 for i in range(min(length, len(ranked_letters)))
                   if etaoin[ranked_letters[i]] < length
              )

etaoin_score('Ai fh do epbeg ooneoAurn', 10)
```

[35]: 5

As part of the heuristic, we have to chose a value for the second argument `length` of `etaoin_score()`, which clearly should be neither too small not too large:

[36]:
```
etaoin_length = 6
```

The 8th column of `message` displayed over 13 columns has be encrypted with "E", the 8th character of "Take it Easy!". Deciphering this column with "E" and "e" shows similar results, with at many positions, the same letter, but lowercase for one, and uppercase for the other. The etaoin score is the same for both:

[37]:
```
decrypt('E', encrypted_message[8 :: 13])
decrypt('e', encrypted_message[8 :: 13])

etaoin_score(decrypt('E', encrypted_message[8 :: 13]), etaoin_length)
etaoin_score(decrypt('e', encrypted_message[8 :: 13]), etaoin_length)
```

[37]:
```
'sorgekio d  eturti uwrs'
```

[37]:
```
'SORGEKIOnDnnETURTInUWRS'
```

[37]:
```
3
```

[37]:
```
3
```

If we assume that the original message is "usual" text, with mostly lowercase letters, then we should consider "E" to be more promising than "e". More generally, when deciphering a given column of the encrypted message with two letters, if the ataoin scores differ, then the letter with the highest score is arguably more likely to be correct; but if both ataoin scores are the same, then the letter that yields the highest proportion of lowercase letters over all letters is arguably more likely to be correct.

The following code fragment tentatively deciphers the 8th column of `encryped_message` with each of the 97 members of `characters` (1-character subkeys). It computes the 97 etaoin scores as well as the 97 proportions of lowercase letters over all letters plus 1 (to prevent division by 0). It then ranks each of the 97 1-character subkeys in decreasing value of etaoin score, and for a given etaoin score, in decreasing proportion of lower letters. Eventually, it displays the top 10 results, together with the corresponding tentative deciphering of the 8th column:

[38]:
```
scores = []
for subkey in characters:
    decrypted_column = decrypt(subkey, encrypted_message[8 :: 13])
    nb_of_lowercase_letters = 0
    nb_of_letters = 1
    letters = (c for c in decrypted_column if c.isalpha())
    for c in letters:
        nb_of_letters += 1
        if c.islower():
            nb_of_lowercase_letters += 1
    scores.append((subkey, etaoin_score(decrypted_column, etaoin_length),
                   nb_of_lowercase_letters / nb_of_letters
                  )
                 )
scores.sort(key=itemgetter(1, 2), reverse=True)

for scored_subkey in scores[: 10]:
    print(scored_subkey, decrypt(scored_subkey[0], encrypted_message[8 :: 13]))
```

```
('E', 3, 0.95) sorgekio d  eturti uwrs
```

```
('I', 3, 0.9473684210526315) okncagek{9{{apqnpe{qsno
('K', 3, 0.9411764705882353) mila8eci_7__8nolnc_oqlm
('S', 3, 0.9230769230769231) ead2064a=
==0fgdf4=gide
('e', 3, 0.16666666666666666) SORGEKIOnDnnETURTInUWRS
('J', 2, 0.9411764705882353) njmb9fdj`8``9opmod`prmn
('O', 2, 0.9285714285714286) ieh64a8e[3[[4jkhj8[kmhi
('W', 2, 0.875) a69     ~206/}//~bc9b0/ce9a
('r', 2, 0.4583333333333333) FBEtrxvBaqaarGHEGvaHJEF
('n', 2, 0.4166666666666667) JFIxvBzFeueevKLIKzeLNIJ
```

Given a tentative key length l, hoped to be the actual length of the hidden key, there are l 1-character subkeys to discover, one for each column of the message displayed over l many columns. One would like to consider, for each column, nothing but its most promising 1-character subkeys. Restricting ourselves to, for each column, its k most promising subkeys, a total of k^l full keys can then be assembled from the l 1-character subkeys. So k has to be small. Let us still make it another parameter of the overall heuristics and set it to a default of 2:

[39]:
```python
nb_of_options_for_subkey = 2
```

To easily generate all possible keys from all choices of 1-character subkeys, the `product` class from the `itertools` module is useful:

[40]:
```python
# The cartesian product {0, 1} x {'A', 'B'} x {'c', 'd'}
list(product(range(2), ['A', 'B'], (x for x in 'cd')))
# The cartesian product ({0, 1} x {'A', 'B'})^2
list(product(range(2), ['A', 'B'], repeat=2))
```

[40]:
```
[(0, 'A', 'c'),
 (0, 'A', 'd'),
 (0, 'B', 'c'),
 (0, 'B', 'd'),
 (1, 'A', 'c'),
 (1, 'A', 'd'),
 (1, 'B', 'c'),
 (1, 'B', 'd')]
```

[40]:
```
[(0, 'A', 0, 'A'),
 (0, 'A', 0, 'B'),
 (0, 'A', 1, 'A'),
 (0, 'A', 1, 'B'),
 (0, 'B', 0, 'A'),
 (0, 'B', 0, 'B'),
 (0, 'B', 1, 'A'),
 (0, 'B', 1, 'B'),
 (1, 'A', 0, 'A'),
 (1, 'A', 0, 'B'),
 (1, 'A', 1, 'A'),
 (1, 'A', 1, 'B'),
 (1, 'B', 0, 'A'),
 (1, 'B', 0, 'B'),
```

```
(1, 'B', 1, 'A'),
(1, 'B', 1, 'B')]
```

The following code fragment passes as an argument to `list()` a generator expression to generate all keys of length 3 from a triple of 2 possible options for each of three 1-character subkeys, with `'0'` or `'1'` for the first subkey, `'A'` or `'B'` for the second subkey, `'c'` or `'d'` for the third subkey:

[41]:
```
subkeys = ('0', '1'), ('A', 'B'), ('c', 'd')

list(key for key in (''.join(subkey) for subkey in product(*subkeys)))
```

[41]:
```
['0Ac', '0Ad', '0Bc', '0Bd', '1Ac', '1Ad', '1Bc', '1Bd']
```

Having assembled a complete key from 1-character subkeys, the encrypted message can be tentatively decrypted as a whole. Whereas 1-character subkeys look more or less promising depending on the distribution of letters in a tentatively decrypted column, whole keys look more or less promising depending on whether the tentatively decrypted message contains enough letters amongst all characters, and enough English words amongst all words, here defining a word as a longest sequence of consecutive letters. So any longest sequence of consecutive nonletters is a word separator; that is something not for the `split()` method of the `str` class, but for the `split()` function of the `re` module, using the syntax of regular expressions:

[42]:
```
# Using any of the characters between the square brackets as a separator
split('[+$(^?=_)]', '0+$abc(^?DEF=_')
# Using any longest sequence of the characters  between the square
# brackets as a separator
split('[+$(^?=_)]+', '0+$abc(^?DEF=_')
# Using any lowercase letter as a separator
split('[a-z]', '0+$abc(^?DEF=_')
# Using anything but a lowercase letter as a separator
split('[^a-z]', '0+$abc(^?DEF=_')
# Using any longest sequence of letters as a separator
split('[a-zA-Z]+', '0+$abc(^?DEF=_')
# Using any longest sequence of anything but letters or digits as a
# separator
split('[^a-zA-Z0-9]', '0+$abc(^?DEF=_')
```

[42]:
```
['0', '', 'abc', '', '', 'DEF', '', '']
```

[42]:
```
['0', 'abc', 'DEF', '']
```

[42]:
```
['0+$', '', '', '(^?DEF=_']
```

[42]:
```
['', '', '', 'abc', '', '', '', '', '', '', '', '']
```

[42]:
```
['0+$', '(^?', '=_']
```

[42]:
```
['0', '', 'abc', '', '', 'DEF', '', '']
```

So `'[^a-zA-Z]+'` is the regular expression we need:

[43]:
```
print(split('[^a-zA-Z]+', message))
```

```
['Alice', 'was', 'beginning', 'to', 'get', 'very', 'tired', 'of', 'sitting',
```

```
'by', 'her', 'sister', 'on', 'the', 'bank', 'and', 'of', 'having', 'nothing',
'to', 'do', 'once', 'or', 'twice', 'she', 'had', 'peeped', 'into', 'the',
'book', 'her', 'sister', 'was', 'reading', 'but', 'it', 'had', 'no', 'pictures',
'or', 'conversations', 'in', 'it', 'and', 'what', 'is', 'the', 'use', 'of', 'a',
'book', 'thought', 'Alice', 'without', 'pictures', 'or', 'conversations', '']
```

To complete the heuristics, we make use of two last parameters. First a desired fraction of letters over all letters in the tentatively deciphered message, set to a default of 70%. Second a desired fraction of English words over all words in the tentatively deciphered message, set to a default of 50%:

[44]:
```
fraction_of_letters = 0.7
fraction_of_words = .5
```

With `dictionary` denoting a list of all lowercase English words (first read from a file in practice), the following function returns `True` or `False` depending on whether the tentatively decrypted message passed as first argument looks right or not. In case the function returns `True`, the tentatively decrypted message will be displayed to the user to accept, or to reject, in which case the program will resume its search for the correct key:

[45]:
```
def looks_like_English(text, dictionary):
    if sum(1 for c in text if c.isalpha()) / len(text) < fraction_of_letters:
        return False
    possible_words = split('[^a-zA-Z]+', text)
    nb_of_words = sum(1 for w in possible_words if w in dictionary)
    return nb_of_words / len(possible_words) > fraction_of_words
```

We are ready to put everything together.

- A function `encrypt_file()` is designed to encrypt the text contained in a file, whose name is provided as second argument, the first argument being the encryption key. By default, the function displays the encrypted message to the screen, but the third argument, set by default to `None`, can be changed to the name of a file and the encrypted message will then be written to this file instead.
  - To read from a file, the second argument should be the name of an existing file. Otherwise, `open()` will raise a `FileNotFoundError` exception, which the codes catches in an `except` statement.
  - To write to a new file or overwrite the contents of an existing file, the `open()` function is given `'w'` as second argument.
  - `encrypt_file()` just calls the `encrypt()` function to perform the encryption. That function expects the message to encrypt to be given as a single string, with as many embedded `'\n'` characters in the string as there are lines in the message. To read the whole contents of a file as a single string, we use the `read()` method of the object returned by the `open()` function.
- A function `decrypt_file()` is designed to decrypt the text contained in a file, whose name is provided as second argument, the first argument being the encryption key. By default, the function displays the decrypted message to the screen, but the third argument, set by default to `None`, can be changed to the name of a file and the decrypted message will then be written to this file instead, provided that it does not exist.

  - To write to a new file, the `open()` function is given `'x'` as second argument; in case the file exists, then `open()` will not overwrite it but instead, raise a `FileExistsError` exception.
  - `decrypt_file()` just calls the `decrypt()` function to perform the decryption, also using `read()` to get the message to decrypt as a single string.

- A function `break_key_for_file()` is designed to try and break the key of an encrypted message stored in a file whose name is passed as argument to `break_key_for_file()`. Here again, `read()` is used to get the text to decrypt (from a key that first, has to be discovered) as a single string, that is passed an an argument to another function, `break_key()`.
  - `break_key()` opens a file meant to contain a list of English words, one per line. The `strip()` method is used to discard the new line character that ends each line of this file, the `lower()` method to convert each word to all lowercase.
  - `break_key()` quickly displays all keys it comes up with to tentatively decrypt the encrypted message; carriage return is used to quickly display all keys of a given length on a single line, each such key overwriting the previous one.
  - When `looks_like_English()` returns `True`, `break_key()` displays the first 200 characters of the tentatively deciphered message, and prompts the user to express whether the message has been successfully decrypted, or whether search for another key should be resumed. In case it exhausts all keys it comes up with, none of which is either proposed to or validated by the user, `break_key()` admits defeat.

[46]:
```
dictionary_file = 'dictionary.txt'

def encrypt_file(key, filename, encrypted_filename=None):
    try:
        with open(filename) as file:
            if encrypted_filename:
                with open(encrypted_filename, 'w') as encrypted_file:
                    print(encrypt(key, file.read()), end='',
                          file=encrypted_file
                         )
            else:
                return encrypt(key, file.read())
    except FileNotFoundError:
        print(f'Could not open {filename}, giving up.')

def decrypt_file(key, filename, decrypted_filename=None):
    try:
        with open(filename) as file:
            if decrypted_filename:
                try:
                    with open(decrypted_filename, 'x') as decrypted_file:
                        print(decrypt(key, file.read()), file=decrypted_file)
                except FileExistsError:
                    print(f'{decrypted_filename} exists, giving up.')
            else:
                return decrypt(key, file.read())
```

```python
    except FileNotFoundError:
        print(f'Could not open {filename}, giving up.')

def break_key_for_file(filename):
    try:
        with open(filename) as file:
            break_key(file.read())
    except FileNotFoundError:
        print(f'Could not open {filename}, giving up.')

def break_key(text):
    try:
        with open(dictionary_file) as file:
            dictionary = {w.strip().lower() for w in file}
    except FileNotFoundError:
        print(f'Could not open the file {dictionary_file}, giving up.')
        return
    for key_length in key_lengths_from_most_to_least_promising(text):
        print(f'\nNow working with keys of length {key_length}')
        subkeys = []
        for n in range(key_length):
            scores = []
            for subkey in characters:
                decrypted_column = decrypt(subkey, text[n :: key_length])
                nb_of_lowercase_letters = 0
                nb_of_letters = 1
                letters = (c for c in decrypted_column if c.isalpha())
                for c in letters:
                    nb_of_letters += 1
                    if c.islower():
                        nb_of_lowercase_letters += 1
                scores.append((subkey,
                               etaoin_score(decrypted_column, etaoin_length),
                               nb_of_lowercase_letters / nb_of_letters
                              )
                             )
            scores.sort(key=itemgetter(1, 2), reverse=True)
            subkeys.append(x[0] for x in scores[: nb_of_options_for_subkey])
        for key in (''.join(subkey) for subkey in product(*subkeys)):
            print('\r', key, end='')
            decrypted_text = decrypt(key, text)
            if looks_like_English(decrypted_text, dictionary):
                print('\nWhat about this?\n')
                print(decrypted_text[: 200], '...', sep='')
                print()
                print('Enter Y[es] if happy, otherwise press any key '
                      "and I'll keep working."
```

```
                    )
                yes_or_no = input('> ')
                if yes_or_no in {'YES', 'Yes', 'yes', 'Y', 'y'}:
                    print(f'The key is: "{key}"')
                    return
    print('Sorry, I did my best...')
```

[47]:
```
encrypt_file('Take it Easy!', 'carroll.txt', 'encrypted_carroll.txt')
break_key_for_file('encrypted_carroll.txt')
```

```
Now working with keys of length 1
 1
Now working with keys of length 13
 Take it Easy!
What about this?

Alice was beginning to get very tired of sitting by her sister on the bank, and
of having nothing to do: once or twice she had peeped into the book her sister
was reading, but it had no pictures or co…

Enter Y[es] if happy, otherwise press any key and I'll keep working.
> Y
The key is: "Take it Easy!"
```