# Python Projects

Solved by Michael Zolotarenko

## The Complex ABC Problem ✎ Code Challenges

Download the Complex ABC file package

**Introduction**

The project described here is a development of a classic ABC code challenge, providing solution aimed to add the algorithm more universality considering the problem combinatorial features.

**Description**

The classic Rosetta Code's ABC Problem is a string analysis challenge requiring a function to determine if a word can be spelled using provided blocks. However, the basic task and its solution(s) suffer from certain limitation(s). In particular, due to the features of the provided blocks, combinatorial panes of the problem are naturally ignored.

The basic task comes with a specific and ordered block set featuring mirrored blocks for any repeated letter. In other words, in the given pairs, the second occurrence of a letter in a pair {i, j} if exists is always in the mirrored pair {j, i}:

(B O), (X K), (D Q), (C P), (N A), (G T), (R E), (T G), (Q D), (F S), (J W), (H U), (V I), (A N), (O B), (E R), (F S), (L Y), (P C), (Z M).

This feature of the given set simplifies the task. If, instead, the pairs were organized differently, such as {i, j} and {k, i}, the algorithm would become more complex due to combinatorial considerations. In such cases, we would need to check each letter with its multiple occurrences, leading to increased complexity in the algorithm. For example, given any 10 random blocks, we should run the test up to P(10,10) = 10! = 3,628,800 times to determine whether the word can be formed using the given pairs.

The next example demonstrates the non-applicability of the basic solution of the problem when we use a randomized block set.

Given the following set: [{'D', 'B'}, {'K', 'O'}, {'D', 'J'}, {'A', 'S'}, {'O', 'E'}].

If we test the abstract word 'boda' against the given set, the program returns 'True', i.e. the word can be formed. However, if we test another abstract word 'doba', composed of the same set of letters, the result is 'False'.

The program version included here responds to the combinatorial limitations described above, providing a solution for any random set of blocks with randomized pairs of letters.

**Release Files ( ↓ )**

- `abc-problem-complex.py`: Contains the implementation of the solution for the ABC Problem considering combinatorial features.

- `block_sets.py`: Generates randomized block sets.

- `README.md`: Provides information and instructions for the project.

**Task** (view at rosettacode.org)

You are given a collection of ABC blocks (maybe like the ones you had when you were a kid).
There are twenty blocks with two letters on each block.
A complete alphabet is guaranteed amongst all sides of the blocks.
The sample collection of blocks:
(B O), (X K), (D Q), (C P), (N A), (G T), (R E), (T G), (Q D), (F S), (J W), (H U), (V I), (A N), (O B), (E R), (F S), (L Y), (P C), (Z M).

**Task**
Write a function that takes a string (word) and determines whether the word can be spelled with the given collection of blocks.

The rules are simple:
1. Once a letter on a block is used that block cannot be used again

2. The function should be case-insensitive
3. Show the output on this page for the following 7 words in the following example

**Example**

| **Input** | **Expected output** |
|---|---|
| >>> can_make_word("A") | True |
| >>> can_make_word("BARK") | True |
| >>> can_make_word("BOOK") | False |
| >>> can_make_word("TREAT") | True |
| >>> can_make_word("COMMON") | False |
| >>> can_make_word("SQUAD") | True |
| >>> can_make_word("CONFUSE") | True |

## Solution

**abc-problem-complex.py** (download full release from GitHub Portfolio repository)

```
#& The Complex ABC problem (*)
#* The code below provides a solution for any random set of blocks with randomized pairs of
letters.
# The following version uses custom factorial function, but it can be replaced by imported math
tools.

# Import the alphabet, a random set of blocks, and define global variables
from block_sets import random_blocks
import string

alphabet = list(string.ascii_letters)
word_letters = []

print("Hello, dear friend!")
# Get a number of blocks from user
# Function to generate a new block set
def generate_blocks():
    num_blocks = int(input("How many blocks do you want to generate (1-50)? "))
    if not 0 < num_blocks <= 50:
        raise ValueError("Please enter an integer in the range 1 to 50")

    return random_blocks(num_blocks)

# Generate the initial block set
blocks = generate_blocks()
lenb = len(blocks) # Used for calculating the number of permutations

# ABC Complex function
def abc_blocks_function(word):
    word_letters.clear()  # Reset and initialize the list for a new word

    for symbol in word:
        if symbol in alphabet:
            word_letters.append(symbol.upper())
        else:
            raise ValueError("Please, use Latin alphabet letters only!")

    #! Complex ABC combinatoric modification

    def factorial(lenb): # Calculates the number of permutations given the number of blocks/
C(n,n)=n!
        if lenb == 1: # We already excluded 0 by limiting user input range to [1-50]
            return 1
        else:
            return lenb * factorial(lenb - 1)

    for combo in range(factorial(lenb)):
        # Create a copy of the block list for each word check
        block_letters = blocks[:]

        # Iterate through the word and blocks letters to find matches
        for letter in word_letters[:]:
            found = False
            for pair in block_letters[:]:
                if letter in pair:
                    found = True
                    word_letters.remove(letter)
                    block_letters.remove(pair)
                    break

            if not word_letters:
                print(f"The word '{word}' can be formed! :)")
                return True

        print(f"The word '{word}' cannot be formed :(")
        return False

# Prompt the user to enter a word
while True:
    try:
        word = input("Enter a word + |Enter| ____ '1' + |Enter| to change blocks set ____ '0'
(zero) + |Enter| to quit: ")
        if word == '0' or word == '':
            print("Goodbye!")
            break
        elif word == '1':
            blocks = generate_blocks()
        elif len(word) > len(blocks):
            raise ValueError(f"The word is too long! With the chosen set of blocks, the max
possible word length is {len(blocks)} characters")
        else:
            abc_blocks_function(word)

    except Exception as e:
        print("An error occurred:", e)
```