```
from itertools import combinations

#
# Note range should be selectable 😎
#
notes = ["C", "D", "E", "F", "G", "A", "B"]
note_range = []

desired_octaves = int(input("select number of octave (range): "))


for i in range(desired_octaves) :
    note_range.append([note + str(i + 1) for note in notes])

avaiable_notes = sum(note_range, [])




#
#intervals defining
#
tuples_for_intervals = []
tuples_for_intervals_over_C = []

for x in range(1, desired_octaves + 1):
    for y in range(x, desired_octaves + 1):
        tuples_for_intervals.append((x, y))

tuples_for_intervals_over_C = [pair for pair in tuples_for_intervals if pair[0] != pair[1]]




#
#Definizione funzione
#
def generate_tuples(numbers, letter_tuples):
    result_tuples = []
    for letters in letter_tuples:
        temp_result = []
        for number in numbers:
            temp_result.append((f"{letters[0]}{number[0]}", f"{letters[1]}{number[1]}"))
        result_tuples.append(temp_result)
    return [item for sublist in result_tuples for item in sublist]
```

```
#Abstract definition of intervals
unison_def = [("C", "C"),("D", "D"),("E", "E"),("F", "F"),("G", "G"),("A", "A"),("B", "B")]
perfect_fifth_def1 = [("C", "G"),("D", "A"),("E", "B")]
perfect_fifth_def2 = [("F", "C"),("G", "D"),("A", "E")]
#imperfect consonances DF,EG,AC*,BD*; CE,FA,GB; EC,AF*,BG*; CA,DB,FD*,GE*
minor_third_def1 = [("D", "F"),("E", "G")]
minor_third_def2 = [("A", "C"),("B", "D")]
major_third_def1 = [("C", "E"),("F", "A"),("G", "B")]
minor_sixth_def1 = [("E", "C")]
minor_sixth_def2 = [("A", "F"),("B", "G")]
major_sixth_def1 = [("C", "A"),("D", "B")]
major_sixth_def2 = [("F", "D"),("G", "E")]
#dissonances EF,BC*; CD,DE,FG,GA,AB; FB; BF*; DC*,ED*,GF*,AG*,BA*; CB,FE*
minor_second_def1 = [("E", "F")]
minor_second_def2 = [("B", "C")]
major_second_def1 = [("C", "D"),("D", "E"),("F", "G"),("G", "A"),("A", "B")]
perfect_fourth_def1 = [("C", "F"),("D", "G"),("E", "A")]
perfect_fourth_def2 = [("G", "C"), ("A", "D"), ("B", "E")]
augmented_fourth_def1 = [("F","B")]
diminished_fifth_def2 = [("B","F")]
minor_seventh_def2 = [("D","C"),("E","D"),("G","F"),("A","G"),("B","A")]
major_seventh_def1 = [("C","B")]
major_seventh_def2 = [("F","E")]


#perfect consonances CC,DD,EE,FF,GG,AA,BB; CG,DA,EB,FC*,GD*,AE*
unison = generate_tuples(tuples_for_intervals, unison_def)
perfect_fifth = generate_tuples(tuples_for_intervals, perfect_fifth_def1) +
generate_tuples(tuples_for_intervals_over_C, perfect_fifth_def2)
#imperfect consonances DF,EG,AC*,BD*; CE,FA,GB; EC,AF*,BG*; CA,DB,FD*,GE*,FD*
minor_third = generate_tuples(tuples_for_intervals, minor_third_def1) +
generate_tuples(tuples_for_intervals_over_C, minor_third_def2)
major_third = generate_tuples(tuples_for_intervals, major_third_def1)
minor_sixth = generate_tuples(tuples_for_intervals, minor_sixth_def1) +
generate_tuples(tuples_for_intervals_over_C, minor_sixth_def2)
major_sixth = generate_tuples(tuples_for_intervals, major_sixth_def1) +
generate_tuples(tuples_for_intervals_over_C, major_sixth_def2)
#dissonances EF,BC*; CD,DE,FG,GA,AB; FB; BF*; DC*,ED*,GF*,AG*,BA*; CB,FE*
minor_second = generate_tuples(tuples_for_intervals, minor_second_def1) +
generate_tuples(tuples_for_intervals_over_C, minor_second_def2)
major_second = generate_tuples(tuples_for_intervals, major_second_def1)
perfect_fourth = generate_tuples(tuples_for_intervals, perfect_fourth_def1) +
generate_tuples(tuples_for_intervals_over_C, perfect_fourth_def2)
augmented_fourth = generate_tuples(tuples_for_intervals, augmented_fourth_def1)
diminished_fifth = generate_tuples(tuples_for_intervals_over_C, diminished_fifth_def2)
minor_seventh = generate_tuples(tuples_for_intervals_over_C, minor_seventh_def2)
major_seventh = generate_tuples(tuples_for_intervals, major_seventh_def1)

perfect_consonances = unison + perfect_fifth
imperfect_consonances = minor_third + major_third + minor_sixth + major_sixth
dissonances = minor_second + major_second + perfect_fourth + augmented_fourth + diminished_fifth
+ minor_seventh + major_seventh
```

```
#MELODY writing and melodies processing (duplication,transposition,delay...)
# "Canon simplex", "all'roverscio", "motu contrario e rectu"

print("range: " , avaiable_notes[0] , "-" , avaiable_notes[-1])
user_melody = input("input melody as 'C1D1E3': ")


melody = [(user_melody[i:i+2]) for i in range(0, len(user_melody), 2)]

#Gamut flippato per roverscio
flipped_notes = avaiable_notes[::-1]


#DERIVED MELODIES
#obsolete melody_to_test_simplex = [melody]
melody_roverscio = [flipped_notes[avaiable_notes.index(item)] for item in melody]
melody_contrario = melody[::-1]
melody_roverscio_contrario = melody_roverscio[::-1]

#AUGMENTED MELODIES
melody_x2 = [item for sublist in zip(melody, melody) for item in sublist]
melody_roverscio_x2 = [item for sublist in zip(melody_roverscio, melody_roverscio) for item in
sublist]
melody_contrario_x2 = [item for sublist in zip(melody_contrario, melody_contrario) for item in
sublist]
melody_roverscio_contrario_x2 = [item for sublist in zip(melody_roverscio_contrario,
melody_roverscio_contrario) for item in sublist]
all_augmented_melodies = [melody_x2, melody_roverscio_x2, melody_contrario_x2,
melody_roverscio_contrario_x2]


#TRANSPOSITIONS
# Function to generate transposed melodies
def generate_all_transpositions(melody_to_transpose):
    shifted_lists = []
    for shift in range(-len(avaiable_notes) + 1, len(avaiable_notes)):
        shifted_list = []
        for item in melody_to_transpose:
            index_in_A = avaiable_notes.index(item)
            shifted_index = index_in_A + shift
            if 0 <= shifted_index < len(avaiable_notes):
                shifted_list.append(avaiable_notes[shifted_index])
        if len(shifted_list) == len(melody_to_transpose):
            shifted_lists.append(shifted_list)
    return shifted_lists

#DELAYS
def generate_all_comes(melody_box):
    # Create an empty list to store the rotated versions
```

```
    comes_list = []
    # Iterate over each sublist in the original list
    for melody in melody_box:
        # Create rotations for the current sublist and append them to the rotated list
        for i in range(len(melody)):
            comes_list.append(melody[i:] + melody[:i])
    if comes_list and not comes_list[-1]:
        comes_list.pop()
    return comes_list




#Melodies box
melody_transposition_box = generate_all_transpositions(melody)
melody_roverscio_transposition_box = generate_all_transpositions(melody_roverscio)
melody_contrario_transposition_box = generate_all_transpositions(melody_contrario)
melody_roverscio_contrario_transposition_box =
generate_all_transpositions(melody_roverscio_contrario)

complete_melody_transposition_box = generate_all_comes(melody_transposition_box)
complete_melody_roverscio_transposition_box =
generate_all_comes(melody_roverscio_transposition_box)
complete_melody_contrario_transposition_box =
generate_all_comes(melody_contrario_transposition_box)
complete_melody_roverscio_contrario_transposition_box =
generate_all_comes(melody_roverscio_contrario_transposition_box)

#for augmented canon (for now only in starting position)
COMPLETE_MELODIES = melody_transposition_box + melody_roverscio_transposition_box +
melody_contrario_transposition_box + melody_roverscio_contrario_transposition_box
double_melodies = [[element for sublist in combination for element in sublist] for combination in
combinations(COMPLETE_MELODIES, 2)]
double_melodies_comes_box = generate_all_comes(double_melodies)




#Define similar motion
def is_similar_motion(matrix, column):

    def determine_motion_single_voice(matrix, voice, column):
        #   determine up or down in single voice (line)
        index1 = avaiable_notes.index(matrix[voice][column-1])
        index2 = avaiable_notes.index(matrix[voice][column])

        if index1 < index2:
            return "goes up"
        elif index1 > index2:
            return "goes down"
```

```
        else:
            return "stays there"


    motion_voice1 = determine_motion_single_voice(matrix, 0, column)
    motion_voice2 = determine_motion_single_voice(matrix, 1, column)

    if (motion_voice1 == "goes up" and motion_voice2 == "goes up") or \
       (motion_voice1 == "goes down" and motion_voice2 == "goes down"):
        return True
    else:
        return False

#UNTESTED
#Define contrary motion
def is_contrary_motion(matrix, column):
    def determine_motion_single_voice(matrix, voice, column):
        # Determine up or down in single voice (line)
        index1 = available_notes.index(matrix[voice][column-1])
        index2 = available_notes.index(matrix[voice][column])

        if index1 < index2:
            return "goes up"
        elif index1 > index2:
            return "goes down"
        else:
            return "stays there"

    motion_voice1 = determine_motion_single_voice(matrix, 0, column)
    motion_voice2 = determine_motion_single_voice(matrix, 1, column)

    # Check if one voice goes up and the other goes down, or vice versa
    if (motion_voice1 == "goes up" and motion_voice2 == "goes down") or \
       (motion_voice1 == "goes down" and motion_voice2 == "goes up"):
        return True
    else:
        return False




# ************************************************
# * HARD RULES *
# ************************************************

#I SPECIE COUNTERPOINT CONDITIONS

#(All downbeats consonant questionable)
#all perfect intervals must be approached by contrary/oblique motion
#(repeated notes in the counterpoint may not occur against repeated notes in cf)
#(counterpoint may run parallel to CF for 4 notes maximum)
#(skips must account for less than half of the melodic motions)
#(direct repetition of the whole contrapuntal combination)
```

```
#NO PARALLEL OCTAVES



#DISSONANCES
def contains_dissonance(matrix):
    for column in range(len(matrix[0])):
        column_elements = [row[column] for row in matrix]
        for tuple_sublist in dissonances:
            # Check the original sublist
            if tuple(column_elements) == tuple_sublist:
                return True
            # Rotate the sublist
            rotated_sublist = tuple_sublist[1:] + (tuple_sublist[0],)  # Rotate sublist
            # Check the rotated sublist
            if tuple(column_elements) == rotated_sublist:
                return True
    return False


# WRONG FIFTHS
def contains_wrong_fifths(matrix):
    for column in range(len(matrix[0])):
        column_elements = [row[column] for row in matrix]
        for tuple_sublist in perfect_fifth:
            # Check the original sublist
            if tuple(column_elements) == tuple_sublist:
                # Condition A is true, now check condition C
                if is_similar_motion(matrix, column):
                    return True
            # Rotate the sublist
            rotated_sublist = tuple_sublist[1:] + (tuple_sublist[0],)  # Rotate sublist
            # Check the rotated sublist
            if tuple(column_elements) == rotated_sublist:
                # Condition B is true, now check condition C
                if is_similar_motion(matrix, column):
                    return True
    return False


# WRONG OCTAVES
def contains_wrong_octaves(matrix):
    def contains_octave(column):
        column_elements = [row[column] for row in matrix]
        for tuple_sublist in unison:
            if tuple(column_elements) == tuple_sublist:
                return True
            rotated_sublist = tuple_sublist[1:] + (tuple_sublist[0],)  # Rotate sublist
            if tuple(column_elements) == rotated_sublist:
                return True
        return False
```

```
    #resto della funzione
    for column in range(len(matrix[0])):
        if contains_octave(column) and is_similar_motion(matrix, column):
            return True
        column_elements = [row[column] for row in matrix]
        previous_column_elements = [row[column-1] for row in matrix]
        if contains_octave(column) and contains_octave(column-1) and  column_elements[0][0] !=
previous_column_elements[0][0]:
            return True
    return False



# ************************************************
#EVALUATION
# ************************************************


#DISCARD FUNCTION
def is_bad_counterpoint(matrix):

    # Check if at least one HARD RULE is true (therefore, broken)
    if contains_wrong_octaves(matrix) or contains_wrong_fifths(matrix) or
contains_dissonance(matrix):
        return True
    else:
        return False


#TESTING ENVIRONMENT
#2 Voice testing environment
def generate_and_check(melody, melody_boxB, result_list):
    for item in melody_boxB:
        testing_matrix = [melody, item]
        # Check conditions for the testing matrix
        if is_bad_counterpoint(testing_matrix):
            #print("Wrong result Matrix:", testing_matrix)
            continue
        else:
            #print("Result Matrix:", testing_matrix)
            result_list.append(testing_matrix)
    return result_list


# Result boxes
results_box_canon_simplex = []
results_box_roverscio = []
results_box_contrario = []
results_box_roverscio_contrario = []
results_box_augmentation = []

# Processing
```

```
generate_and_check(melody, complete_melody_transposition_box, results_box_canon_simplex)
generate_and_check(melody, complete_melody_roverscio_transposition_box, results_box_roverscio)
generate_and_check(melody, complete_melody_contrario_transposition_box, results_box_contrario)
generate_and_check(melody, complete_melody_roverscio_contrario_transposition_box,
results_box_roverscio_contrario)
# Processing for augmented solutions
generate_and_check(melody_x2, double_melodies_comes_box, results_box_augmentation)
generate_and_check(melody_roverscio_x2, double_melodies_comes_box, results_box_augmentation)
generate_and_check(melody_contrario_x2, double_melodies_comes_box, results_box_augmentation)
generate_and_check(melody_roverscio_contrario_x2, double_melodies_comes_box,
results_box_augmentation)



#OUTPUT
print("canon simplex")
print(results_box_canon_simplex)
print("al roverscio")
print(results_box_roverscio)
print("in contrario motu")
print(results_box_contrario)
print("al roverscio/contrario")
print(results_box_roverscio_contrario)
print("per aumentazione (con raddoppio)")
print(results_box_augmentation)


print(len(COMPLETE_MELODIES), "derived melodies being tested")
print(len(double_melodies_comes_box), "augmented versions being tested")
```