Jia Cheng Li

1) Make an input.txt with words in this format:

```
AI_Cryptarithmetic_Solv
  1    SEND
  2    MORE
  3    MONEY
```

-A four letter word + four letter word = 5 letter word

2) Write the name of the input file in line 421:

```
419        print(str_3)
420
421    main('input2.txt')
422
```

-In the above example, the input file is called input2.txt.

-Replace the 'input2.txt' with the name of your input file follow by the '.txt'

3) Run the Python file

Prerequisite: Python and a code compiler

How to run using VSCode:

```
PS C:\Users\jiach\Documents\College Fall 2019\AI> python project_1.py
```

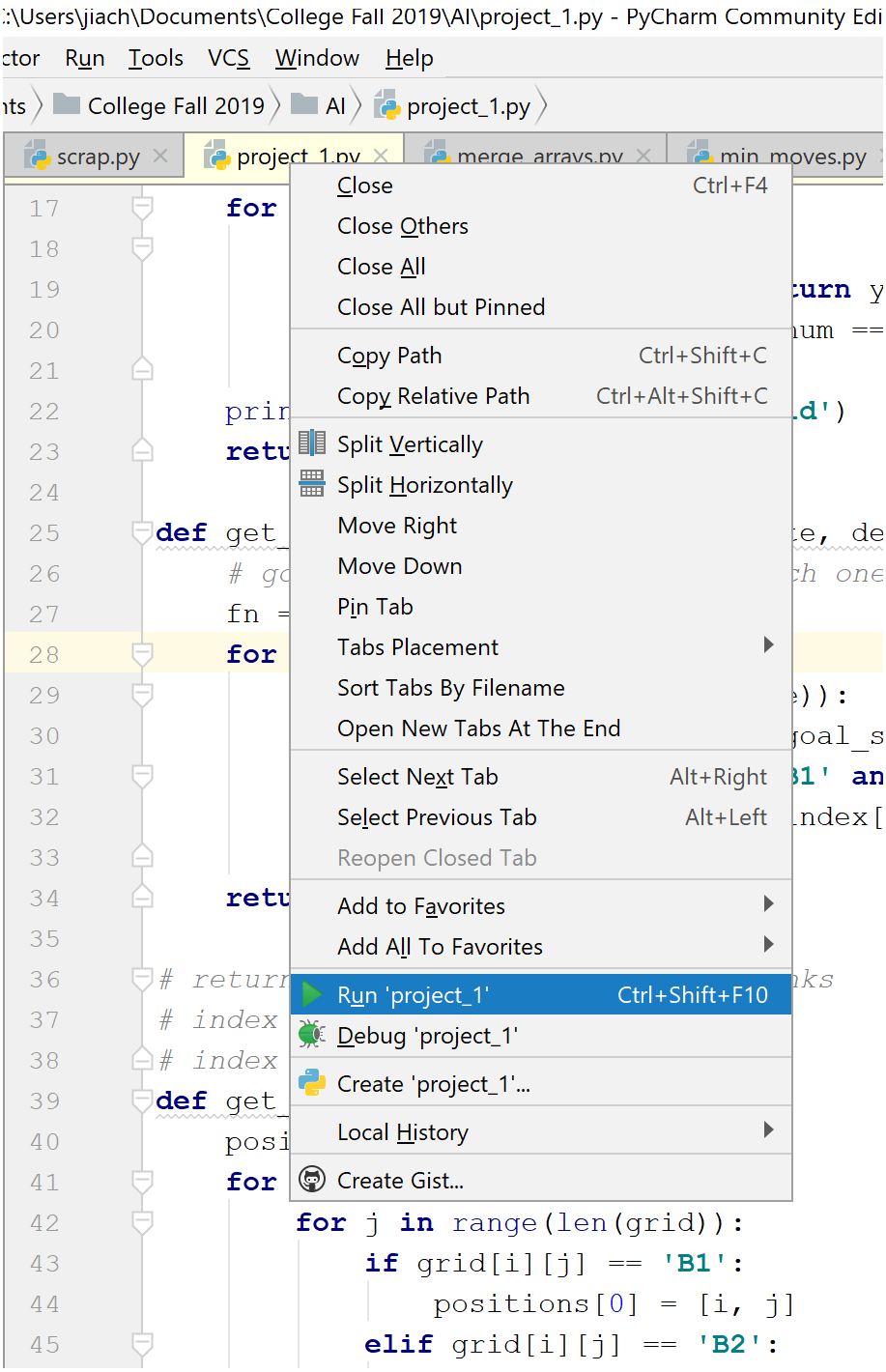
-Go to the directory containing the source code file; in my case it is …..\AI

-Type: python <name of the source code file> in terminal

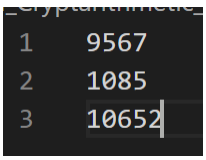
-In my case the name is project_1.py

How to run using PyCharm:

-Go to the file and press the run button in PyCharm:


:\Users\jiach\Documents\College Fall 2019\AI\project_1.py - PyCharm Community Edi
ctor Run Tools VCS Window Help
College Fall 2019
AI
project_1.py
scrap.py
project_1.py
merge_arrays.py
min_moves.py
Close Ctrl+F4
Close Others
Close All
Close All but Pinned
Copy Path Ctrl+Shift+C
Copy Relative Path Ctrl+Alt+Shift+C
Split Vertically
Split Horizontally
Move Right
Move Down
Pin Tab
Tabs Placement
Sort Tabs By Filename
Open New Tabs At The End
Select Next Tab Alt+Right
Select Previous Tab Alt+Left
Reopen Closed Tab
Add to Favorites
Add All To Favorites
Run 'project_1' Ctrl+Shift+F10
Debug 'project_1'
Create 'project_1'...
Local History
Create Gist...
for j in range(len(grid)):
if grid[i][j] == 'B1':
positions[0] = [i, j]
elif grid[i][j] == 'B2':

Output1.txt:


9567
1085
10652

Output2.txt:

```
7483
7455
14938
```

Source Code:

```
import copy

# keeps track of the positions (1-13) and domain(0-9) of a character (e.g. x1)
class Node():
    def __init__(self, char):
        self.char = char
        self.positions = []
        self.domain = []

# help keep track of the current assigned values and carry overs
class State():
    def __init__(self):
        self.values = []
        self.assigned = set()
        self.C1 = None
        self.C2 = None
        self.C3 = None
        self.C4 = None

# make a dict with the position as the key from a character_dict (char as key)
def make_position_dict(char_dict):
    postion_dict = {}
    for key, val in char_dict.items():
        for i in val.positions:
            if i not in postion_dict:
                postion_dict[i] = val.char
    return postion_dict

# define the appropriate domain for each char
def assign_domain(s1, s2, s3):
    print(s1, s2, s3)
    print(len(s1), len(s2), len(s3))
    char_dict = {}
    # make a dict for each char containing its positions
    str_lst = [s1, s2, s3]
    counter = 1
```

```
    for word in str_lst:
        for i in word:
            if i not in char_dict:
                char_dict[i] = Node(i)
            char_dict[i].positions.append(counter)
            counter += 1
    # get domains for each char
    for char, node in char_dict.items():
        # only domain for x9 is 1
        if 9 in node.positions:
            node.domain.append(1)
        else:
            start_domain = 0
            # 1 and 5 are leading terms, so domain: 1-9
            if 1 in node.positions or 5 in node.positions:
                start_domain = 1 # start at 1
            for i in range(start_domain, 10):
                node.domain.append(i)
    # make position dict
    position_dict = make_position_dict(char_dict)
    # assign edge case domains
    edge_positions = [10, 13]
    for num in edge_positions:
        # if previous 2 terms are same char then their sum gives even domain
        if position_dict[num-5] == position_dict[num-9]:
            char = position_dict[num]
            char_dict[char].domain = [0, 2, 4, 6, 8]

    return char_dict

# true if val is not already assigned in state
# check if the number works out in the addition
# if work then use this func again with modify = 1 to modify state
def can_add_val(state, val, char_dict, char, modify):
    # False if val is already used by a letter
    if val in state.values:
        return False
    positions = char_dict[char].positions
    values = copy.deepcopy(state.values)
    C1, C2, C3, C4 = state.C1, state.C2, state.C3, state.C4
    for p in positions:
        values[p-1] = val

    x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13 = \
    values[0], values[1], values[2], values[3], values[4], values[5], \
```

```
    values[6], values[7], values[8], values[9], values[10],\
    values[11], values[12]

    if x4 != -1 and x8 != -1 and x13 != -1:
        val = x4 + x8
        if val >= 10:
            C1 = 1
        else:
            C1 = 0
        if x4 + x8 != x13 + C1*10:
            return False
    if x3 != -1 and x7 != -1 and x12 != -1:
        # if C1 is not calculated then test for C1 = 0 and C1 = 1
        # return False if fail for both
        if C1 == None:
            # test when c1 == 0
            # val = x3 + x7 +0
            flag = True
            C1 = 0
            val = x3 + x7 + C1
            if val >= 10:
                C2 = 1
            else:
                C2 = 0
            if x3 + x7 + C1 != x12 + C2*10:
                flag = False
            if not flag:
                # test when c1 == 1
                C1 = 1
                val = x3 + x7 + C1
                if val >= 10:
                    C2 = 1
                else:
                    C2 = 0
                if x3 + x7 + C1 != x12 + C2*10:
                    return False
        else:
            val = x3 + x7 + C1
            if val >= 10:
                C2 = 1
            else:
                C2 = 0
            if x3 + x7 + C1 != x12 + C2*10:
                return False
    if x2 != -1 and x6 != -1 and x11 != -1:
```

```
        # if C1 is not calculated then test for C1 = 0 and C1 = 1
        # return False if fail for both
        if C2 == None:
            flag = True
            C2 = 0
            val = x2 + x6 + C2
            if val >= 10:
                C3 = 1
            else:
                C3 = 0

            if x2 + x6 + C2 != x11 + C3*10:
                flag = False
            if not flag:
                C2 = 1
                val = x2 + x6 + C2
                if val >= 10:
                    C3 = 1
                else:
                    C3 = 0
                if x2 + x6 + C2 != x11 + C3*10:
                    return False
        else:
            val = x2 + x6 + C2
            if val >= 10:
                C3 = 1
            else:
                C3 = 0
            if x2 + x6 + C2 != x11 + C3*10:
                return False
    if x1 != -1 and x5 != -1 and x10 != -1:
        # if C1 is not calculated then test for C1 = 0 and C1 = 1
        # return False if fail for both
        if C3 == None:
            flag = True
            C3 = 0
            val = x1 + x5 + C3
            if val >= 10:
                C4 = 1
            else:
                C4 = 0

            if x1 + x5 + C3 != x10 + C4*10:
                flag = False
            if not flag:
```

```
                C3 = 1
                val = x1 + x5 + C3
                if val >= 10:
                    C4 = 1
                else:
                    C4 = 0
                if x1 + x5 + C3 != x10 + C4*10:
                    return False
        else:
            val = x1 + x5 + C3
            if val >= 10:
                C4 = 1
            else:
                C4 = 0
            if x1 + x5 + C3 != x10 + C4*10:
                return False
        # C4 == x9 == 1
        if C4 == 0:
            return False

    if modify:
        state.values = values
        print('values')
        print(state.values)
        # modify values in state
        state.C1, state.C2, state.C3, state.C4 = C1, C2, C3, C4
        print('CHAR:', char)
        # add char to assigned
        state.assigned.add(char)
        print('assigned')
        print(state.assigned)

    return True


# iterate through the positions where the character is located and
# return the lowest number of empty neighbors
def get_empty_neighbors(char, state, char_dict):
    min_num = None
    # check empty neighbors for each position
    positions = char_dict[char].positions
    for i in positions: # positions from 1-13
        num_empty = 0
        # current position in state: state.values[i-1]
        if i - 1 > 0:
```

```
            if state.values[i-2] == -1:
                num_empty += 1
        if i + 1 < 14:
            if state.values[i] == -1:
                num_empty += 1
        if min_num == None:
            min_num = num_empty
        elif num_empty < min_num:
            min_num = num_empty
    return min_num

# returns the char with the lowest domain
# if tie, then go with the char with the lowest unassigned neighbors
# if tie again, then choose a random tied char
def select_var(char_dict, state):
    min_char = None
    tied_domains = set()
    for char, node in char_dict.items():
        if char in state.assigned:
            continue
        if min_char == None:
            min_char = char
        else:
            min_len = len(char_dict[min_char].domain)
            curr_len = len(char_dict[char].domain)
            # add any tied chars
            if curr_len == min_len:
                tied_domains.add(min_char)
                tied_domains.add(char)
            elif curr_len < min_len:
                min_char = char
    # domains tied- choose the key with the fewest unassigned neighbors
    # if tied- just return the current lowest one
    if len(tied_domains) > 0:
        min_char = None
        min_num_empty = None
        for i in tied_domains:
            if min_char == None:
                min_char = i
                min_num_empty = get_empty_neighbors(min_char, state, char_dict)
            else:
                curr_empty = get_empty_neighbors(i, state, char_dict)
                if curr_empty < min_num_empty:
                    min_char = i
                    min_num_empty = curr_empty
```

```
    return min_char

def copy_state(state):
    res_state = State()
    res_state.values= state.values
    res_state.C1 = state.C1
    res_state.C2 = state.C2
    res_state.C3 = state.C3
    res_state.C4 = state.C4
    return res_state

def copy_dict(dict_to_copy):
    res_dict = {}
    for key, val in dict_to_copy.items():
        if key not in res_dict:
            res_dict[key] = Node(val.char)
            # deep copy of positions
            res_positions = []
            for i in val.positions:
                res_positions.append(i)
            res_dict[key].positions = res_positions
            # deep copy of domain
            res_domain = []
            for i in val.domain:
                res_domain.append(i)
            res_dict[key].domain = res_domain
    return res_dict

def check(state):
    values = state.values
    x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13 = values[0],  \
    values[1], values[2], values[3], values[4], values[5], values[6], values[7],
\
    values[8], values[9], values[10], values[11], values[12]
    C1, C2, C3, C4 = 0, 0, 0, 0
    if x13 >= 10:
        C1 = 1
        # x13 = 10- x13
    if x12 >= 10:
        C2 = 1
        # x12 = 10- x12
    if x11 >= 10:
        C3 = 1
        # x11 = 10- x11
    if x10 >= 10:
```

```
        C4 = 1
        # x10 = 10- x10
    # constraints
    # x4 + x8 = x13 + C1*10
    # x3 + x7 + C1 = x12 + C2*10
    # x2 + x6 + C2 = x11 + C3*10
    # x1 + x5 + C3 = x10 + C4*10
    # C4 = x9
    if (x4 + x8 == x13 + C1*10) and (x3 + x7 + C1 == x12 + C2*10) \
    and (x2 + x6 + C2 == x11 + C3*10) and (x1 + x5 + C3 == x10 + C4*10) and (C4 =
= x9):
        return True
    return False


# done if positions are all assigned and values are all uniq
def done(state):
    values = state.values
    # all values have to be assigned, e.g. not = -1
    for i in values:
        if i == -1:
            return False
    # check if satisfy constraint
    return check(state)

# def remove_position(position_dict, position):
#     new_dict = {}
#     for key, val in position_dict.items():
#         if key != position:
#             new_dict[key] = val
#     return new_dict

# change modify values back
def remove_assigned(state, char, positions):
    state.assigned.remove(char)
    # reset the values to -1
    for i in positions:
        # reset any carry over to none
        if i in [4, 8, 13]:
            if i == 13: state.C1 = None
            state.C1 = None
        elif i in [3, 7, 12]:
            if i == 12: state.C2 = None
            state.C2 = None
        elif i in [2, 6, 11]:
```

```
            if i == 11: state.C3 = None
            state.C3 = None
        elif i in [1, 5, 10]:
            if i == 10: state.C4 = None
            state.C4 = None
        state.values[i-1] = -1


def backtrack(char_dict, state):
    print("-----------------------")
    values = state.values
    # check if done
    flag = True
    # check if all varaibles have a unique number
    for i in values:
        if i == -1:
            flag = False
            break
    if flag: # if all variables are filled then check constraints
        return check(state)

    # select unassigned var
    char = select_var(char_dict, state)
    print(char, char_dict[char].positions, char_dict[char].domain)
    # get domain val from var (can make var a node)
    domain_vals = char_dict[char].domain
    for i in domain_vals:
        if can_add_val(state, i, char_dict, char, 0):
            print('domain_vals', domain_vals)
            print('domain:', i)
            can_add_val(state, i, char_dict, char, 1) # apply changes to res_stat
e
            backtrack(char_dict, state)
            # check if done
            flag = True
            for i in state.values:
                if i == -1:
                    flag = False
                    break
            # if all are assigned then should be correct since
            # only add numbers that satisfy contraints
            if flag: return True
            positions = char_dict[char].positions
            remove_assigned(state, char, positions)
    print('failed')
```

```
    return False

def main(file_name):
    with open(file_name) as f:
        # get words from files
        words = f.readlines()
        # get rid of any \n
        for i in range(len(words)):
            words[i] = words[i].strip()
    # assign each word a domain of nums
    char_dict = assign_domain(words[0], words[1], words[2])
    for key, node in char_dict.items():
        print(key, node.char, node.positions, node.domain)
    state = State()
    initial_values = []
    for i in range(13):
        initial_values.append(-1)
    state.values = initial_values
    res = backtrack(char_dict, state)
    print(res)
    print(state.assigned)
    values = state.values
    str_1 = ''
    str_2 = ''
    str_3 = ''
    for i in values[0:4]: str_1 += str(i)
    for i in values[4:8]: str_2 += str(i)
    for i in values[8:13]: str_3 += str(i)
    print(str_1)
    print(str_2)
    print(str_3)

main('input2.txt')
```