

B10_csv_grid.py

```
# =====
"""CSV_GRID : display a csv file content in a text grid"""
# =====
__author__ = "Philippe Blasi"
__version__ = "1.0"
__date__ = "2023-05-02"
__usage__ = ""
User input : <filename> [filename ...]
App output : grid of the provided csv files"""
# =====
from ezCLI import *
# -----
def csv_grid(name:str) -> str:
    """return the string grid for the given csv file name"""
    matrix = read_csv(name) # read whole content of file 'name'
    return grid(matrix)
# -----
def parser(command:str) -> str:
    """parse 'command' and return content of provided files formatted as a grid"""
    names = parse(command); #inspect()
    if type(names) is str:
        return csv_grid(names)
    else:
        return '\n'.join(csv_grid(name) for name in names)
# =====
if __name__ == '__main__':
    userloop(parser, "Enter <filename> [filename ...]")
# =====
```

B11_csv_html.py

```
# =====
"""CSV_HTML : save a csv file content in a HTML file"""
# =====
__author__ = "Philippe Blasi"
__version__ = "1.0"
__date__ = "2023-05-02"
__usage__ = ""
User input : <filename> [filename ...]
App output : HTML file with the csv file content in a table"""
# =====
from ezCLI import *
# -----
def csv_html(name:str) -> str:
    """save the name csv file content in a HTML file"""
    matrix = read_csv(name) # read whole content of file 'name'
    rows, cols = len(matrix), len(matrix[0]) # get matrix size from data file
    new_name = name.split('.')[0] # get the file name without extension
    new_name = new_name+'.html' # add the html extension to the name
    # -----
    # begining of the html file
    html_begin=f"""
<!DOCTYPE html>
<html lang="fr">

    <head>
        <meta charset="utf-8" />
        <title>
            {name}
        </title>
    </head>
```

```
<body>
    <h1>Fichier {name}</h1>
    <table border="1">
        ""
        # -----
        # ending of the html file
        html_end = ""
        </table>
    </body>
</html>
"""
# -----
# add to each cell the <td> </td> mark up
table = [[f"<td>{cell}</td>" for cell in row] for row in matrix]
# convert each row from list to string and add the <tr> </tr> mark up
table = [ "<tr>\n"+'\n'.join(row)+"</tr>\n" for row in table]
# convert the table list into string
table = '\n'.join(table)
# add the beginning and the eand of the html file for finishing the treatment
result = html_begin + table + html_end
# save the resultat into the new html file
write_txt(new_name, result)
# -----
```

```
def parser(command:str) -> str:
    """parse 'command' and return content of provided files with line numbering"""
    names = parse(command); #inspect()
    if type(names) is str:
        csv_html(names)
    else:
        for name in names:
            csv_html(name)
        return f"{command} converted"
# =====
if __name__ == '__main__':
    userloop(parser, "Enter <filename> [filename ...]")
# =====
```

B1_count.py

```
# =====
"""COUNT : print the number of lines, words and chars in a set of text files"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
__usage__ = ""
User input : <filename> [filename ...]
App output : number of lines, words and chars for each provided file"""
# =====
from ezCLI import *
# -----
def count(name:str) -> str:
    """return the number of lines, words and chars stored in file 'name'"""
    text = read_txt(name) # read whole content of file 'name'
    line, word, char = len(text.split('\n')), len(text.split()), len(text)
    return f"{name} : lines = {line}, words = {word}, chars = {char}"
# -----
def parser(command:str) -> str:
    """parse 'command' and return line/word/char counters for provided files"""
    tuple_name = parse(command); #inspect()
    if type(tuple_name) is str:
        res = count(tuple_name)
    else:
        tab_res=[count(name) for name in tuple_name]
```

```

res = "\n".join(tab_res)

return res
# =====
if __name__ == '__main__':
    userloop(parser, "Enter <filename> [filename ...]")
# =====

B2_rilex.py
# =====
"""RILEX : print the lexical richness of a list of text files"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
__usage__ = """
User input : <filename> [filename ...]
App output : lexical richness of all provided files"""
# =====
from ezCLI import *
# -----
def rilex(name:str) -> str:
    """return the lexical richness of file 'name'"""
    words = read_txt(name).split() # split file content into words
    words = [word.upper() for word in words if len(word) > 3] # filter words
    histo = {} # initialize the word's histogram as an empty dictionary
    for word in words: histo[word] = histo.get(word,0) + 1
    return f"{name} : rilex = {len(histo)/max(1,len(words)):0.2}"
# -----
def parser(command:str) -> str:
    """parse 'command' and return lexical richness of all provided files"""
    names = parse(command)
    return '\n'.join(rilex(name) for name in names)
# =====
if __name__ == '__main__':
    userloop(parser, "Enter <filename> [filename ...]")
# =====

B3_numlines.py
# =====
"""NUMLINES : print a list of text files with line numbering"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
__usage__ = """
User input : <filename> [filename ...]
App output : content of all provided files with line numbering"""
# =====
from ezCLI import *
# -----
def numlines(name:str) -> str:
    """add line numbering to all lines of file 'name'"""
    text = read_txt(name) # read whole content of file 'name'
    lines = text.split('\n')
    size = len(str(len(lines))) # max number of digits for counter
    lines = [f"{p+1:>{size}} - {line}" for p,line in enumerate(lines)]
    rule = '\n' + '\u2500'*80 + '\n' # horizontal rule (used as a separator)
    return name + ' : ' + rule + '\n'.join(lines) + rule
# -----
def parser(command:str) -> str:
    """parse 'command' and return content of provided files with line numbering"""
    names = parse(command); #inspect()

```

```

return '\n'.join(numlines(name) for name in names)
# =====
if __name__ == '__main__':
    userloop(parser, "Enter <filename> [filename ...]")
# =====

B4_dice.py
# =====
"""DICE : display a graphical representation for a list of random dice rolls"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
__usage__ = """
User input : <number of dice> (must be an integer between 1 and 8)
App output : graphical representation for the provided number of dice rolls"""
# =====
from ezCLI import *
from random import randrange
# -----
def roll(n:int) -> int:
    """return a list of 'n' random dice rolls"""
    return [1 + randrange(6) for loop in range(n)]
# -----
def dice(rolls:list) -> str:
    """return a graphical representation for a list of random dice rolls"""
    # first create the dice font as a single string with 5 lines of 60 characters
    font = """
.....:.....:.....:.....:.....:
.      .:      .:      .:      .:      .:      .:      .:      .:      .:
.  .    .:      .:      .:      .:      .:      .:      .:      .:      .:
.      .:      .:      .:      .:      .:      .:      .:      .:      .:
.....:.....:.....:.....:.....:
"""
    # then convert font into a 5x6 matrix by splitting string at '\n' and at ':'
    font = [line.split(':') for line in font.strip().split('\n')]
    # for each line, extract the chars of the current digit from the font matrix
    lines = [[line[roll-1] for roll in rolls] for line in font]
    # and finally, join everything as a multi-line string
    return '\n'.join(' '.join(line) for line in lines)
# -----
def parser(command:str) -> str:
    """parse 'command' and return the graphical representation for dice rolls"""
    n = convert(command)
    assert type(n) is int and 0<n<9, "number must be an integer between 1 and 8"
    return dice(roll(n)) # roll 'n' dice and return its graphical representation
# =====
if __name__ == '__main__':
    userloop(parser, "Enter number of dice")
# =====

B5A_fiboplus.py
# =====
"""FIBOPLUS : generate the Fibonacci sequence with several options"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0" # use 'parse' function with default values
__date__ = "2022-11-12"
__usage__ = """
User input : ['n' = <n>] ['u' = <u>] ['v' = <v>] ['file' = <filename>]
            - n:int = number of terms for the sequence (default = 10)
            - u:int = value of the first term (default = 0)
            - v:int = value of the second term (default = 1)

```

```

- filename:str = name of output file (default = output on screen)
App ouput : Fibonacci sequence defined by user arguments

Note: to use all default values, simply enter <SPACE> as command"""
# =====
from ezCLI import *
# -----
def fibo(n:int, u:int, v:int) -> int:
    """compute the nth term of the Fibonacci sequence starting from (u,v)"""
    a, b = u, v
    for p in range(n):
        a, b = b, a+b
    return a
# -----
def fibos(n:int, u:int, v:int) -> str:
    """return the 'n' first terms of the Fibonacci starting from (u,v)"""
    return '\n'.join([f"fibonacci({p}) = {fibonacci(p,u,v)}" for p in range(n+1)])
# -----
def parser(command:str) -> str:
    """parse 'command' and return Fibonacci sequence of provided arguments"""
    default = 'n=10 u=0 v=1 file=' # default values for all arguments
    # parse 'command' and use default values for missing arguments
    args = parse(command, default); #inspect()
    # store all values from dictionary 'args' into variables
    n, u, v, file = (args[name] for name in ('n','u','v','file')); #inspect()
    assert type(n) is int and n >= 0, "<n> must be a positive integer"
    assert type(u) is int, "<u> must be an integer"
    assert type(v) is int, "<v> must be an integer"
    assert type(file) is str, "<filename> must be a string"
    if not file: return fibos(n, u, v) # show result on screen
    write_txt(file, fibos(n, u, v)) # write result in 'file'
    return f"Fibonacci sequence ({n} values) written in file '{file}'"
# =====
if __name__ == '__main__':
    userloop(parser)
# =====

```

B5B_fiboplus.py

```

# =====
"""FIBOPLUS : generate the Fibonacci sequence with several options"""
# -----
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "2.0" # convert function 'fibo' into generator
__date__ = "2022-11-12"
__usage__ = """
User input : ['n = ' <n>'] ['u = ' <u>'] ['v = ' <v>'] ['file = ' <filename>]
- n:int = number of terms for the sequence (default = 10)
- u:int = value of the first term (default = 0)
- v:int = value of the second term (default = 1)
- file:str = filename for output (default = output on screen)
App ouput : Fibonacci sequence defined by user arguments

Note: to use all default values, simply enter <SPACE> as command"""
# =====
from ezCLI import *
# -----
def fibo(n:int, u:int, v:int) -> int:
    """compute the nth term of the Fibonacci sequence starting from (u,v)"""
    a, b = u, v; yield a
    for p in range(n):
        a, b = b, a+b
    yield a
# -----

```

```

def fibos(n:int, u:int, v:int) -> str:
    """return the 'n' first terms of the Fibonacci starting from (u,v)"""
    return '\n'.join([f"fibonacci({p}) = {fibonacci(p,u,v)}" for p in enumerate(fibonacci(n,u,v))])
# -----
def parser(command:str) -> str:
    """parse 'command' and return Fibonacci sequence of provided arguments"""
    default = "n=10 u=0 v=1 file=" # default values for all arguments
    # parse 'command' and use default values for missing arguments
    args = parse(command, default); #inspect()
    # store all values from dictionary 'args' into variables
    n, u, v, file = (args[name] for name in ('n','u','v','file')); #inspect()
    assert type(n) is int and n >= 0, "<n> must be a positive integer"
    assert type(u) is int, "<u> must be an integer"
    assert type(v) is int, "<v> must be an integer"
    assert type(file) is str, "<filename> must be a string"
    if not file: return fibos(n, u, v)
    write_txt(file, fibos(n, u, v))
    return "Fibonacci sequence (%s values) written in file %r" % (n, file)
# =====
if __name__ == '__main__':
    userloop(parser)
# =====

```

B5_fiboplus.py

```

# =====
"""FIBOPLUS : generate the Fibonacci sequence with several options"""
# -----
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0" # use 'parse' function with default values
__date__ = "2022-11-12"
__usage__ = """
User input : ['n = ' <n>'] ['u = ' <u>'] ['v = ' <v>'] ['file = ' <filename>]
- n:int = number of terms for the sequence (default = 10)
- u:int = value of the first term (default = 0)
- v:int = value of the second term (default = 1)
- filename:str = name of output file (default = output on screen)
App ouput : Fibonacci sequence defined by user arguments

```

```

Note: to use all default values, simply enter <SPACE> as command"""
# =====
from ezCLI import *
# -----
def fibo(n:int, u:int, v:int) -> int:
    """compute the nth term of the Fibonacci sequence starting from (u,v)"""
    a, b = u, v
    for p in range(n):
        a, b = b, a+b
    return a
# -----
def fibos(n:int, u:int, v:int) -> str:
    """return the 'n' first terms of the Fibonacci starting from (u,v)"""
    return '\n'.join([f"fibonacci({p}) = {fibonacci(p,u,v)}" for p in range(n+1)])
# -----
def parser(command:str) -> str:
    """parse 'command' and return Fibonacci sequence of provided arguments"""
    default = 'n=10 u=0 v=1 file=' # default values for all arguments
    # parse 'command' and use default values for missing arguments
    args = parse(command, default); #inspect()
    # store all values from dictionary 'args' into variables
    n, u, v, file = (args[name] for name in ('n','u','v','file')); #inspect()
    assert type(n) is int and n >= 0, "%r : invalid value for 'n'" % n
    assert type(u) is int, "%r : invalid value for 'u'" % u
    assert type(v) is int, "%r : invalid value for 'v'" % v

```

```
assert type(file) is str, "%r : invalid filename" % file
if not file: return fibos(n, u, v) # show result on screen
write_txt(file, fibos(n, u, v)) # write result in 'file'
return "Fibonacci sequence (%s values) written in file %r" % (n, file)
# =====
if __name__ == '__main__':
    userloop(parser)
# =====

B6_matstat.py

# =====
"""MATSTAT : perform statistical operations on a matrices stored in CSV files"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
__usage__ = ""
User input : <filename> [filename ...]
            where filename = text file containing a set of integer or float v.....
.....alues
App output : The matrix stored in each provided CSV file is displayed with
            additional rows showing the result of statistical operations
            performed on each column: min, max, mean and deviation"""
# =====
from ezCLI import *
# -----
def matstat(name:str) -> str:
    """perform statistical operations on a matrices stored in file 'name'"""
    #try:
        # 1 - lire le fichier csv
        mat = read_csv(name)
        nb_row = len(mat)
        nb_col = len(mat[0])
        # 2 - créer la chaine de caractère contenant la grille avec le contenu du .....
        # fichier csv
        grid1 = "_" * 80 + "\n" + f"Content of '{name}'" + "\n" + grid(mat)
        # 3 - créer un tableau avec 4 ligne et le même nombre de colonne pour cont.....
        # enir le résultat
        # des calculs statistiques : min, max, moyenne (mean) et écart-type (d.....
        # eviation)
        stat = [[0 for col in range(nb_col)] for row in range(4)]
        # 4 - remplir chaque colonne de ce tableau
        # pour chaque colonne de mat
        # créer un tableau t contenant cette colonne
        # calculer les statistiques sur ce tableau pour remplir la colonne .....
        # de stat
        for col in range(nb_col):
            t = [mat[i][col] for i in range(nb_row)]
            stat[0][col] = min(t)
            stat[1][col] = max(t)
            stat[2][col] = sum(t)/len(t)
            t_carre = [val**2 for val in t]
            stat[3][col] = sum(t_carre)/len(t_carre) - stat[2][col]**2
        # 5 - créer la chaine de caractère contenant la grille avec le contenu de .....
        # tableau de statistiques
        grid2 = '\nFrom top to bottom, results for operators min, max, mean, devia.....
        # tion\n' + grid(stat)
        # 6 - renvoyer les deux chaines.
        return grid1 + grid2
    #except:
        return f"{name} : cannot read file"
# -----
def parser(command:str) -> str:
```

```
"""parse 'command' and return content of provided files with line numbering"""
filenames = parse(command); #inspect()

if type(filenames) is str:
    return matstat(filenames)
else:
    return '\n'.join([matstat(name) for name in filenames])

## res_final = ""
##
## if type(filenames) is str:
##     res_final = numlines(filenames) + "\n"
## else:
##     # pour chaque name de filenames
##     for name in filenames:
##         # appeler count sur ce name de fichier et récupérer le résultat
##         res = matstat(name)
##         # ajouter ce résultat au resultat final en une insérant un retour à .....
        .....la ligne
        res_final += res + "\n"
##
## return res_final
# =====
if __name__ == '__main__':
    userloop(parser, "Enter <filename> [filename ...]")
# =====

B7_scramble.py

# =====
"""SCRAMBLE : scramble the lines of a text file according to a password"""
# =====
__author__ = "Christophe Schlick"
__version__ = "1.0"
__date__ = "2015-09-01"
__usage__ = ""
User input : <password> <filename> [filename...]
App output : scramble the content of all 'filenames' according to 'password'
Note: use quotes if 'password' contains space characters"""
# =====
from ezCLI import *
# -----
def scramble(password, name):
    """scramble the content of file 'name' by using characters of 'password'"""
    text, n = '', len(password) # the length of 'password' is needed for cycling
    for p, char in enumerate(read_txt(name)): # enumerate all chars from file
        # cycle around all characters in password, keep its 6 least significant bits
        # and use them to scramble each file character by applying an XOR operator
        code_char, code_pass = ord(char), 63 & ord(password[p % n])
        text += chr(code_char ^ code_pass) # scramble code with XOR operator
    write_txt(name, text) # replace file content by scrambled text
    return "File %r has been scrambled" % name
# -----
def parser(command):
    """extract arguments from 'command' before calling 'scramble()'"""
    password, *names = parse(command); #inspect()
    return '\n'.join(scramble(password, name) for name in names)
# =====
if __name__ == '__main__':
    userloop(parser, "Enter <password> <filename> [filename...]")
# =====

B8A_primefacto.py

# =====
```

```
"""PRIMEFACTO : compute all prime factorizations for numbers from 2 to n"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0" # input CSV file and output INI file
__date__ = "2022-11-12"
__usage__ = ""
User input : <n> (where n:int > 1)
App output : sequence of all prime factorizations for numbers from 2 to n"""
# =====
from ezCLI import convert, read_csv, write_ini
# =====
def primefacto(n:int, primes:list) -> dict:
    """compute all prime factorizations for numbers from 2 to n"""
    factos = {} # create empty dictionary to store prime factorizations
    for p in range(2, n+1): # loop over numbers to factorize
        factos[p] = [] # insert empty factor list for number 'p'
        r = p # initialize remaining ratio before starting factorization
        for q in primes: # loop over prime numbers to find all factors for 'p'
            if q > r: break # all factors for 'p' have been found
            while r%q == 0: # 'q' is a factor for 'p'
                r = r/q; factos[p].append(q) # update 'r' and append 'q' to list
        return factos # return final dictionary with all factorizations up to 'n'
# =====
def main():
    """parse 'command' as integer 'n' before calling 'prime(n)"""
    primes = read_csv('primes.csv') # read CSV file generated by A8_prime.py
    print(f"{'-'*80}\n{__doc__}{__usage__}\n{'-'*80}") # show info (doc and usage)
    while True: # user interaction loop
        command = input("Enter value for <n> : ") # wait for user input
        if command == '': break # break interaction loop if user input is empty
        n = convert(command) # analyze user input and convert to appropriate type
        assert type(n) is int and n > 1, "<n> must be an integer and greater than 1"
        # =====
        factos = primefacto(n, primes) # compute all prime factorizations
        #print(factos) # print raw dictionary without post-processing
        # =====
        # post-process the dictionary to obtain a more intuitive display
        for p in factos: factos[p] = ' * '.join(str(q) for q in factos[p])
        for p in factos: print(f"{p} = {factos[p]}")
        # =====
        write_ini('factos.ini', factos) # write dictionary on disk as INI file
        print('See you later...')
# =====
if __name__ == "__main__":
    main()
# =====
B9A_latin.py
# =====
"""LATIN : generate a random latin square"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0" # draft implementation without command line options
__date__ = "2022-11-12"
__usage__ = ""
Input : n = size of grid"""
# =====
from random import randrange
# =====
def latin(n:int) -> list:
    """generate a matrix containing a random n by n latin square"""
    mat = [[1+(p+q)%n for q in range(n)] for p in range(n)] # initial matrix
    for loop in range(999*n): # loop 999*n times to get pretty good shuffling
```

```
p, q = randrange(n), randrange(n)
    for r in range(n): # exchange row p with row q
        mat[p][r], mat[q][r] = mat[q][r], mat[p][r]
    p, q = randrange(n), randrange(n)
    for r in range(n): # exchange col p with col q
        mat[r][p], mat[r][q] = mat[r][q], mat[r][p]
    return mat
# =====
def grid(mat:list) -> str:
    """return a string containing a graphical grid generated from matrix 'mat'"""
    # find the maximum number of digits required for the matrix elements
    digits = len(str(max(val for row in mat for val in row)))
    # create row and col separators, accounting for the maximum number of digits
    col_sep, row_sep = '|', f"\n{('+' + '-'*digits) * len(mat)}+\n"
    # convert each matrix element to a string with a fixed length
    mat = [[f"{val:{digits}}" for val in row] for row in mat]
    # finally join everything together as a multi-line string
    rows = [f"{col_sep}{col_sep.join(cols)}{col_sep}" for cols in mat]
    return f"{row_sep}{row_sep.join(rows)}{row_sep}"
# =====
def main():
    """main program of the "latin" module"""
    print(f"{'-'*80}\n{__doc__}{__usage__}\n{'-'*80}") # show info (doc and usage)
    while True:
        line = input("<> Enter value of 'n' : ")
        if line == '': break
        try:
            print(grid(latin(int(line))))
        except ValueError:
            print("Error : invalid input value")
# =====
if __name__ == '__main__':
    main()
# =====
B9B_latin.py
# =====
"""LATIN : generate a random latin square"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "2.0" # full implementation with command line options
__date__ = "2022-11-12"
__usage__ = ""
Input : [n=value] [file=filename]
        where n = size of matrix (default = 10)
        and filename = text file used for output (default = screen)
Note : enter empty line to stop interaction loop"""
# =====
from random import randrange
# =====
def setoptions(options:str, command) -> dict:
    """return dictionary containing union of default and user-defined options"""
    for option in command:
        option = option.split('=')
        if len(option) != 2 or option[1] == '' or option[0] not in options:
            raise NameError('='.join(option))
        options[option[0]] = option[1]
    return options
# =====
def latin(n:int) -> list:
    """generate a matrix containing a random n by n latin square"""
    mat = [[1+(p+q)%n for q in range(n)] for p in range(n)] # initial matrix
    for loop in range(999*n): # loop 999*n times to get pretty good shuffling
```

```

    p, q = randrange(n), randrange(n)
    for r in range(n): # exchange row p with row q
        mat[p][r], mat[q][r] = mat[q][r], mat[p][r]
    p, q = randrange(n), randrange(n)
    for r in range(n): # exchange col p with col q
        mat[r][p], mat[r][q] = mat[r][q], mat[r][p]
    return mat
# -----
def grid(mat:list) -> str:
    """return a string containing a graphical grid generated from matrix 'mat'"""
    # find the maximum number of digits required for the matrix elements
    digits = len(str(max(val for row in mat for val in row)))
    # create row and col separators, accounting for the maximum number of digits
    col_sep, row_sep = '|', f'\n{(( '+' + '-'*digits) * len(mat))}+\n"
    # convert each matrix element to a string with a fixed length
    mat = [[f"{val:{digits}}" for val in row] for row in mat]
    # finally join everything together as a multi-line string
    rows = [f"{col_sep}{col_sep.join(cols)}{col_sep}" for cols in mat]
    return f"{row_sep}{row_sep.join(rows)}{row_sep}"
# -----
def main():
    """main program of the "latin" module"""
    print(f"{'-'*80}\n{__doc__}{__usage__}\n{'-'*80}") # show info (doc and usage)
    while True:
        options = {'n':'9', 'file':''}
        line = input("<> Enter command : ")
        if line == '': return
        try:
            options = setoptions(options, line.split())
            name, n = options['file'], int(options['n'])
            table = grid(latin(n))
            if name == '':
                print(table)
            else:
                with open(name,'w') as file: file.write(table)
                print("Random latin square written in file {name}")
        except NameError as error:
            print("Invalid option : {error}")
        except ValueError as error:
            print("Invalid input value : {error}")
        except IOError as error:
            print("File access error : {error}")
# -----
if __name__ == '__main__':
    main()
# =====

```