

C1A_grid.py

```

# =====
"""GRID : create some grids with several color patterns"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
# =====
from ezTK import *
from random import randrange
# -----
def stripes(width:int=400, height:int=400, rows:int=40) -> None:
    """alternate black and white horizontal stripes"""
    win = Win(title='STRIPES') # create main window
    height = height // rows # set height for a single stripe
    colors = ('#000','#FFF') # store stripe colors in a tuple
    # -----
    for row in range(rows): # loop over the stripes
        color = colors[row % 2] # set color for new stripe (cycle on 2 colors)
        Brick(win, bg=color, width=width, height=height) # create new stripe
    # -----
    win.loop() # start interaction loop
# -----
def colorstripes(width:int=400, height:int=400, rows:int=40) -> None:
    """alternate color stripes using six colors"""
    win = Win(title='COLORSTRIPES') # create main window
    height = height // rows # set height for a single stripe
    colors = ('#F00','#0F0','#00F','#0FF','#F0F','#FF0') # store stripe colors
    # -----
    for row in range(rows): # loop over stripes
        color = colors[row % 6] # set color for new stripe (cycle on 6 colors)
        Brick(win, bg=color, width=width, height=height) # create new stripe
    # -----
    win.loop() # start interaction loop
# -----
def chessboard(width:int=400, height:int=400, cols:int=8, rows:int=8) -> None:
    """create black and white chessboard"""
    win = Win(title='CHESSBOARD', fold=cols) # set fold property to number of cols
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#000','#FFF') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        color = colors[(row+col) % 2] # set color for new cell (cycle on 2 colors)
        Brick(win, bg=color, height=height, width=width) # create new cell
    # -----
    # Alternative version using two nested loops on rows and cols
    # for row in range(rows): # loop over board rows
    #     for col in range(cols): # loop over board cols
    #         color = colors[(row+col) % 2] # set color for cell (cycle on 2 colors)
    #         Brick(win, bg=color, height=height, width=width) # create new cell
    # -----
    win.loop() # start interaction loop
# -----
def colorboard(width:int=400, height:int=400, cols:int=8, rows:int=8) -> None:
    """create six color checkerboard"""
    win = Win(title='COLORBOARD', fold=cols, op=2) # set 2-pixel outer padding
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#F00','#0F0','#00F','#0FF','#F0F','#FF0') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division

```

```

        color = colors[(row+col) % 6] # set color for new cell (cycle on 6 colors)
        Brick(win, bg=color, height=height, width=width, border=3) # create new cell
    # -----
    win.loop() # start interaction loop
# -----
def boxes(width:int=400, height:int=400, cols:int=13, rows:int=13) -> None:
    """create concentric black and white boxes"""
    win = Win(title='GRID', fold=cols) # set fold property to number of cols
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#000','#FFF') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over the cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        # for each cell, compute its distance to the topmost row, bottommost row,
        # leftmost col and rightmost col, then keep the smallest of the 4 distances
        distance = min(row, rows-row-1, col, cols-col-1)
        color = colors[distance % 2] # set color for new cell (cycle on 2 colors)
        Brick(win, bg=color, height=height, width=width) # create new cell
    # -----
    win.loop() # start interaction loop
# -----
def colorboxes(width:int=400, height:int=400, rows:int=13, cols:int=13) -> None:
    """create concentric color boxes"""
    win = Win(title='GRID', fold=cols) # set fold property to number of cols
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#F00','#0F0','#00F','#0FF','#F0F','#FF0') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        # for each cell, compute its distance to the topmost row, bottommost row,
        # leftmost col and rightmost col, then keep the smallest of the 4 distances
        distance = min(row, rows-row-1, col, cols-col-1)
        color = colors[distance % 6] # set color for new cell (cycle on 6 colors)
        Brick(win, bg=color, height=height, width=width) # create new cell
    # -----
    win.loop() # start interaction loop
# -----
def ballboard(width:int=600, height:int=600, cols:int=12, rows:int=12) -> None:
    """create board containing one random color ball for each cell"""
    win = Win(title='BALLBOARD', fold=cols, bg='#000') # set win background as b.....
    .....lack
    width, height = width // cols, height // rows # set size for a single cell
    images = tuple(Image(f"Z{c}.gif") for c in 'RGBCMY') # store images for cells
    # -----
    for loop in range(rows*cols): # loop over the board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        image = images[randrange(6)] # set random image for new cell
        Brick(win, image=image, height=height, width=width) # create new cell
    # -----
    win.loop() # start interaction loop
# =====
if __name__ == "__main__":
    stripes()
    stripes(800, 600, 200) # do not use default values
    colorstripes()
    chessboard()
    colorboard()
    boxes()
    colorboxes()
    ballboard()
# =====

```

C1B_grid.py

```

# =====
"""GRID : create some grids with several color patterns"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "2.0" # use multi-state widgets
__date__ = "2022-11-12"
# =====
from ezTK import *
from random import randrange
# -----
def stripes(width:int=400, height:int=400, rows:int=40) -> None:
    """alternate black and white horizontal stripes"""
    win = Win(title='STRIPES') # create main window
    height = height // rows # set height for a single stripe
    colors = ('#000','#FFF') # store stripe colors in a tuple
    # -----
    for row in range(rows): # loop over the stripes
        state = row % 2 # set state for new stripe (cycle on 2 states)
        Brick(win, bg=colors, state=state, width=width, height=height)
    # -----
    win.loop() # start interaction loop
# -----
def colorstripes(width:int=400, height:int=400, rows:int=40) -> None:
    """alternate color stripes using six colors"""
    win = Win(title='COLORSTRIPES') # create main window
    height = height // rows # set height for a single stripe
    colors = ('#F00','#0F0','#00F','#0FF','#F0F','#FF0') # store stripe colors
    # -----
    for row in range(rows): # loop over the stripes
        state = row % 6 # set state for new stripe (cycle on 6 states)
        Brick(win, bg=colors, state=state, width=width, height=height)
    # -----
    win.loop() # start interaction loop
# -----
def chessboard(width:int=400, height:int=400, cols:int=8, rows:int=8) -> None:
    """create black and white chessboard"""
    win = Win(title='CHESSBOARD', fold=cols) # set fold property to number of cols
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#000','#FFF') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over the board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        state = (row+col) % 2 # set state for new cell (cycle on 2 states)
        Brick(win, bg=colors, state=state, height=height, width=width)
    # -----
    # Alternative version using nested loops on rows and cols
    for row in range(rows): # loop over board rows
        for col in range(cols): # loop over board cols
            state = (row+col) % 2 # set state for new cell (cycle on 2 states)
            Brick(win, bg=colors, state=state, height=height, width=width)
    # -----
    win.loop() # start interaction loop
# -----
def colorboard(width:int=400, height:int=400, cols:int=8, rows:int=8) -> None:
    """create six color checkerboard"""
    win = Win(title='COLORBOARD', fold=cols, op=2) # set 2-pixel outer padding
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#F00','#0F0','#00F','#0FF','#F0F','#FF0') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over the board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        state = (row+col) % 6 # set state for new cell (cycle on 6 states)
        Brick(win, bg=colors, state=state, height=height, width=width, border=3)

```

```

# -----
win.loop() # start interaction loop
# -----
def boxes(width:int=400, height:int=400, cols:int=13, rows:int=13) -> None:
    """create concentric black and white boxes"""
    win = Win(title='BOXES', fold=cols) # set fold property to number of cols
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#000','#FFF') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over the board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        # for each cell, compute its distance to the topmost row, bottommost row,
        # leftmost col and rightmost col, then keep the smallest of the 4 distances
        distance = min(row, rows-row-1, col, cols-col-1)
        state = distance % 2 # set state for new cell (cycle on 2 states)
        Brick(win, bg=colors, state=state, height=height, width=width)
    # -----
    win.loop() # start interaction loop
# -----
def colorboxes(width:int=400, height:int=400, rows:int=13, cols:int=13) -> None:
    """create concentric color boxes"""
    win = Win(title='COLORBOXES', fold=cols) # fold is controlled by number of c.....
    .....ols
    width, height = width // cols, height // rows # set size for a single cell
    colors = ('#F00','#0F0','#00F','#0FF','#F0F','#FF0') # store cell colors
    # -----
    for loop in range(rows*cols): # loop over the board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        # for each cell, compute its distance to the topmost row, bottommost row,
        # leftmost col and rightmost col, then keep the smallest of the 4 distances
        distance = min(row, rows-row-1, col, cols-col-1)
        state = distance % 6 # set state for new cell (cycle on 6 states)
        Brick(win, bg=colors, state=state, height=height, width=width)
    # -----
    win.loop() # start interaction loop
# -----
def ballboard(width:int=600, height:int=600, cols:int=12, rows:int=12) -> None:
    """create board containing one random color ball for each cell"""
    win = Win(title='BALLBOARD', fold=cols, bg='#000') # set win background as b.....
    .....lack
    width, height = width // cols, height // rows # set size for a single cell
    images = tuple(Image(f"Z{c}.gif") for c in 'RGBCMY') # store images for cells
    # -----
    for loop in range(rows*cols): # loop over the board cells
        row, col = loop // cols, loop % cols # get cell coords by Euclidian division
        state = randrange(6) # set random state for new cell
        Brick(win, image=images, state=state, height=height, width=width)
    # -----
    win.loop() # start interaction loop
# =====
if __name__ == "__main__":
    stripes()
    stripes(800, 600, 200) # do not use default values
    colorstripes()
    chessboard()
    colorboard()
    boxes()
    colorboxes()
    ballboard()
# =====
C2A_counter.py
# =====

```

```

"""COUNTER : a user-controlled digital counter"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
# =====
from ezTK import *
# -----
def main():
    """create the main window and pack the widgets"""
    global win; font1, font2 = 'Courier 16 bold', 'Arial 96 bold'
    win = Win(title='COUNTER', font=font1, op=5) # create main window
    # -----
    frame = Frame(win) # create frame for the 7 buttons
    Button(frame, text='|<', width=5, command=lambda: on_but(0))
    Button(frame, text='<<', width=5, command=lambda: on_but(1))
    Button(frame, text='<', width=5, command=lambda: on_but(2))
    Button(frame, text='0', width=5, command=lambda: on_but(3))
    Button(frame, text='>', width=5, command=lambda: on_but(4))
    Button(frame, text='>>', width=5, command=lambda: on_but(5))
    Button(frame, text='>|', width=5, command=lambda: on_but(6))
    Label(win, text=0, font=font2, border=2) # create counter widget
    # -----
    win.counter = win[1] # friendly name for the counter widget
    win.loop() # start interaction loop
# -----
def on_but(index:int) -> None:
    """generic callback function for all seven buttons"""
    value = win.counter['text'] # get current counter value
    if index == 0: value = -1000 # action for button: |<
    elif index == 1: value -= 10 # action for button: <<
    elif index == 2: value -= 1 # action for button: <
    elif index == 3: value = 0 # action for button: 0
    elif index == 4: value += 1 # action for button: >
    elif index == 5: value += 10 # action for button: >>
    elif index == 6: value = 1000 # action for button: >|
    win.counter['text'] = min(1000, max(-1000, value)) # clamp counter value
# =====
if __name__ == '__main__':
    main()
# =====

```

C2B_counter.py

```

# =====
"""COUNTER : a user-controlled digital counter"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
# =====
from ezTK import *
# -----
def main():
    """create the main window and pack the widgets"""
    global win; font1, font2 = 'Courier 16 bold', 'Arial 96 bold'
    win = Win(title='COUNTER', font=font1, op=5) # create main window
    # -----
    frame = Frame(win, font=font1) # create a frame for the 7 buttons widgets
    text = '|< << < 0 > >> >|'.split() # create a list for the 7 button strings
    for n in range(7): # loop to create the 7 control buttons
        Button(frame, text=text[n], width=5, command=lambda n=n: on_but(n))
    Label(win, text=0, font=font2, border=2) # create counter widget
    # -----

```

```

win.counter = win[1] # friendly name for the counter widget
win.loop() # start interaction loop
# -----
def on_but(index:int) -> None:
    """generic callback function for all seven buttons"""
    slow, fast, minval, maxval = 1, 10, -1000, 1000 # set counter parameters
    value = win.counter['text'] # get current counter value
    # create a tuple containing all 7 new counter values, one for each button
    values = (minval, value-fast, value-slow, 0, value+slow, value+fast, maxval)
    value = values[index] # select counter value using the provided button index
    win.counter['text'] = min(maxval, max(minval, value)) # clamp counter value
# =====
if __name__ == '__main__':
    main()
# =====

```

C3_chrono.py

```

# =====
"""CHRONO : a user-controlled digital stopwatch"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
# =====
from ezTK import *
# -----
def main():
    """create the main window and pack the widgets"""
    global win
    win = Win(title='CHRONO', op=5) # create main window
    font1, font2 = 'Arial 16', 'Times 120 bold' # define fonts for widgets
    # -----
    frame = Frame(win, grow=False, font=font1) # create frame for buttons
    Button(frame, text=('START', 'STOP'), width=9, command=on_start) # multi state
    Button(frame, text='RESET', width=9, command=on_reset) # single state
    # -----
    colors = ('#F00', '#0F0', '#00F', '#0FF', '#F0F', '#FF0') # store colors for chrono
    Label(win, font=font2, width=5, fg=colors, border=2) # create chrono widget
    # -----
    win.start, win.chrono = frame[0], win[1] # friendly names for widgets
    on_reset() # invoke callback for button 'RESET'
    win.loop() # start interaction loop
# -----
def on_reset() -> None:
    """callback function for the 'RESET' button"""
    win.chrono['text'] = win.chrono.state = 0 # reset value and state for chrono
# -----
def on_start() -> None:
    """callback function for the 'START/STOP' button"""
    win.start.state = 1-win.start.state # switch button state (state <--> 1-state)
    if win.start.state == 1: tick() # call 'tick' function when button state is 1
# -----
def tick() -> None:
    """increment counter and make recursive function call after 10ms"""
    win.chrono['text'] += 1 # increment counter value
    win.chrono.state = win.chrono['text']//50 # change chrono state every 50 steps
    if win.start.state == 1: win.after(10, tick) # call next tick after 10ms
# =====
if __name__ == '__main__':
    main()
# =====

```

C4A_colors.py

```
# =====
"""COLORS : edit a color with RGB scales"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0"
__date__ = "2022-11-12"
# =====
from ezTK import *
# -----
def main():
    """create the main window and pack the widgets"""
    global win
    win = Win(title='COLORS', op=5) # create main window
    # -----
    frame = Frame(win, grow=False) # create frame for top line
    Label(frame, width=9, border=1) # use fixed width=9 to avoid layout jittering
    for rgb in ('#F00', '#0F0', '#00F'): # loop over the 3 primary colors R,G,B
        Scale(frame, scale=(0,255), show=False, troughcolor=rgb, command=on_scale)
    Brick(win, height=200) # create brick widget to show selected color
    # -----
    win.brick, win.label = win[1], frame[0] # friendly names for all
    win.R, win.G, win.B = frame[1], frame[2], frame[3] # widgets used in callbacks
    on_scale(); win.loop() # set initial color as '#00000' and start user loop
    # -----
def on_scale() -> None:
    """callback function for all three RGB scales"""
    # get current value for the three RGB scales
    x, r, g, b = '0123456789ABCDEF', win.R.state, win.G.state, win.B.state
    # convert decimal integer values into hexadecimal color string (= '#RRGGBB')
    color = '#' + x[r//16] + x[r%16] + x[g//16] + x[g%16] + x[b//16] + x[b%16]
    win.brick['bg'] = win.label['text'] = color # set new color on brick and label
    # =====
if __name__ == '__main__':
    main()
# =====
```

C5A_leapfrog.py

```
# =====
"""LEAP FROG : implement the LeapFrog puzzle"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0" # play game with mouse (left click on board cells)
__date__ = "2022-11-12"
# -----
from ezTK import *
# -----
def main(frog:int=3):
    """create the main window and pack the widgets"""
    global win
    win = Win(title='LEAP FROG', op=3, click=on_click) # create main window
    images = tuple(Image(f"frog{c}.gif") for c in '012') # read images
    # -----
    board = Frame(win) # create frame to store board cells
    for loop in range(2*frog+1): # loop over all board cells
        Brick(board, image=images, border=2) # create cell with default state = 0
    Button(win, grow=False, text='RESET', command=on_reset) # create reset button
    # -----
    win.frog, win.board = frog, board # store frog and board as win properties
    on_reset(); win.loop() # set initial configuration and start event loop
    # -----
def on_reset() -> None:
    """callback function for the 'RESET' button"""
    frog, board = win.frog, win.board # create local shortcuts
```

```
# define initial cell states : 0 = empty, 1 = blue frog, 2 = green frog
states = [1]*frog + [0] + [2]*frog # create list of initial cell states
for n in range(2*frog+1): # loop over all board cells
    board[n].state = states[n] # set initial state for current cell
# -----
def on_click(widget:object, code:str, mods:str) -> None:
    """callback function for all mouse click events"""
    if widget.master != win.board: return # wrong click (= not on a board cell)
    move(widget.index) # apply move for frog located at selected cell
    # -----
def move(n:int) -> None:
    """move frog located at index 'n', after checking if its move is valid"""
    frog, board = win.frog, win.board # create local shortcuts
    if board[n].state == 1: # blue frog selected
        if n < 2*frog and board[n+1].state == 0: # blue frog can move right
            board[n].state, board[n+1].state = 0, 1 # apply blue frog move
        elif n < 2*frog-1 and board[n+2].state == 0: # blue frog can jump right
            board[n].state, board[n+2].state = 0, 1 # apply blue frog jump
        elif board[n].state == 2: # green frog selected
            if n > 0 and board[n-1].state == 0: # green frog can move left
                board[n].state, board[n-1].state = 0, 2 # apply green frog move
            elif n > 1 and board[n-2].state == 0: # green frog can jump left
                board[n].state, board[n-2].state = 0, 2 # apply green frog jump
    # =====
if __name__ == '__main__':
    main() # create window with default game parameters: frog=3
    #main(7) # try with 7 frogs per color
    # =====
```

C4B_colors.py

```
# =====
"""COLORS : edit a color with RGB scales"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "2.0" # add 'RANDOM' and 'PSYCHE' buttons
__date__ = "2022-11-12"
# -----
from ezTK import *
from random import randrange as rr # import 'randrange' and rename as 'rr'
# -----
def main():
    """create the main window and pack the widgets"""
    global win
    win = Win(title='COLORS', op=5) # create main window
    # -----
    frame = Frame(win, grow=False) # create frame for top line
    Label(frame, width=9, border=1) # use fixed width to avoid layout jittering
    for rgb in ('#F00', '#0F0', '#00F'): # loop over the 3 primary colors R,G,B
        Scale(frame, scale=(0,255,1), show=False, troughcolor=rgb, command=on_scale)
    Button(frame, text='RANDOM', command=on_random) # create random button
    Button(frame, text='PSYCHE', command=on_psyche) # create psyche button
    Brick(win, height=200) # create brick widget to show selected color
    # -----
    win.brick, win.label, win.switch = win[1], frame[0], frame[5]
    win.R, win.G, win.B = frame[1], frame[2], frame[3]
    on_scale(); win.loop() # set initial color as '#000000' and start user loop
    # -----
def on_scale() -> None:
    """callback function for all three RGB scales"""
    # get current value for the three RGB scales
    x, r, g, b = '0123456789ABCDEF', win.R.state, win.G.state, win.B.state
    # convert decimal integer values into hexadecimal color string (= '#RRGGBB')
    color = '#' + x[r//16] + x[r%16] + x[g//16] + x[g%16] + x[b//16] + x[b%16]
```

```

win.brick['bg'] = win.label['text'] = color # set new color on brick and label
# -----
def on_random() -> None:
    """callback function for the RANDOM button"""
    # use rr(256) to select random value between 0 and 255 for R,G,B scales
    win.R.state, win.G.state, win.B.state = rr(256), rr(256), rr(256)
# -----
def on_psychic() -> None:
    """callback function for the PSYCHE button"""
    on_random(); win.after(100, on_psychic) # new random color every 100 ms
# -----
if __name__ == '__main__':
    main()
# -----

```

C5B_leapfrog.py

```

# -----
"""LEAP FROG : implement the LeapFrog puzzle"""
# -----
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "2.0" # play game with keyboard (left and right arrow)
__date__ = "2022-11-12"
# -----
from ezTK import *
# -----
def main(frog:int=3):
    """create the main window and pack the widgets"""
    global win
    win = Win(title='LEAP FROG', op=3, key=on_key) # create main window
    images = tuple(Image(f"frog{c}.gif") for c in '012') # read images
    # -----
    board = Frame(win) # create frame to store board cells
    for loop in range(2*frog+1): # loop over all board cells
        Brick(board, image=images, border=2) # create cell with default state = 0
    Button(win, grow=False, text='RESET', command=on_reset) # create reset button
    # -----
    win.frog, win.board = frog, board # store frog and board as win properties
    on_reset(); win.loop() # set initial configuration and start event loop
# -----
def on_reset() -> None:
    """callback function for the 'RESET' button"""
    frog, board = win.frog, win.board # create local shortcuts
    # define initial cell states : 0 = empty, 1 = blue frog, 2 = green frog
    states = [1]*frog + [0] + [2]*frog # create list of initial cell states
    for n in range(2*frog+1): # loop over all board cells
        board[n].state = states[n] # set initial state for current board cell
# -----
def on_key(widget:object, code:str, mods:str) -> None:
    """callback function for all keyboard events"""
    if code not in ('Right','Left'): return # wrong key (= not a directional key)
    move(code) # apply move for frog in selected direction
# -----
def move(direction:str) -> None:
    """find empty cell and move its neighboring frog in provided direction"""
    frog, board = win.frog, win.board # create local shortcuts
    # first version : use while loop to find index of empty board cell
    n = 0
    while board[n].state != 0: n += 1 # loop while current cell is not empty
    # second version : use for loop to find index of empty board cell
    #for n in range(2*frog+1): # loop over all board cells
    # if board[n].state == 0: break # break loop when empty cell has been found
    # third version : use the 'index' method to directly find index of empty cell
    #states = [cell.state for cell in board] # get state for each board cell

```

```

#n = states.index(0) # call the 'index' method on the list of states
# -----
if direction == 'Right': # try to move or jump right
    if n > 0 and board[n-1].state == 1: # blue frog can move right
        board[n-1].state, board[n].state = 0, 1 # apply blue frog move
    elif n > 1 and board[n-2].state == 1: # blue frog can jump right
        board[n-2].state, board[n].state = 0, 1 # apply blue frog jump
elif direction == 'Left': # try to move or jump left
    if n < 2*frog and board[n+1].state == 2: # green frog can move left
        board[n+1].state, board[n].state = 0, 2 # apply green frog move
    elif n < 2*frog-1 and board[n+2].state == 2: # green frog can jump left
        board[n+2].state, board[n].state = 0, 2 # apply green frog jump
# -----
if __name__ == '__main__':
    main() # create window with default game parameters: frog=3
    #main(7) # try with 7 frogs per color
# -----

```

C6B_slide.py

```

# -----
"""SLIDE : a color-based slide puzzle game"""
# -----
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "2.0" # play game with keyboard
__date__ = "2022-11-12"
# -----
from ezTK import *
from random import shuffle, randrange as rr
# -----
def main():
    """create the main window and pack the widgets"""
    global win
    win = Win(title='SLIDE', op=5, key=on_key) # create main window
    colors = ('FFFF', '#F00', '#0F0', '#00F', '#FF0') # colors for board cells
    # -----
    frame = Frame(win, grow=False)
    Button(frame, text='SHUFFLE', width=9, command=on_shuffle)
    Button(frame, text='RESET', width=9, command=on_reset)
    win.board = Frame(win, fold=5, op=0, border=1)
    for n in range(25): # create all board cells as multi-state Brick widgets
        Brick(win.board, bg=colors, width=64, height=64, border=1)
    # -----
    on_shuffle(); win.loop() # shuffle board and start game
# -----
def on_shuffle() -> None:
    """callback function for the 'SHUFFLE' button"""
    # create list of 25 cells, shuffle it and call 'on_reset' to show new grid
    win.cells = 6*[1,2,3,4] + [0]; shuffle(win.cells); on_reset()
# -----
def on_reset() -> None:
    """callback function for the 'RESET' button"""
    # convert list of 25 cells into a 5x5 grid and show it on game board
    win.grid = [[0 for col in range(5)] for row in range(5)]
    for i in range(25):
        row,col = divmod(i,5) # get cell coords by Euclidian division
        win.grid[row][col] = win.cells[i]

    # position for the white square in win.col_white and win.row_white
    if win.grid[row][col] == 0:
        win.row_white = row
        win.col_white = col

    show()

```

```

# -----
def on_key(widget:object, code:str, mods:str) -> None:
    """callback function for all keyboard events"""
    if code not in ('Right','Left','Up','Down'): return # wrong key
    move(code) # find position of empty cell and move its neighboring cell
# -----
def move(direction:str) -> None:
    """find empty cell and move its neighboring cell in provided direction"""
    row, col = win.row_white, win.col_white
    if direction == 'Right' and col > 0: r, c = row, col-1 # get left-side cell
    elif direction == 'Left' and col < 4: r, c = row, col+1 # get right-side cell
    elif direction == 'Down' and row > 0: r, c = row-1, col # get upper-side cell
    elif direction == 'Up' and row < 4: r, c = row+1, col # get lower-side cell
    else: return # move is not valid
    win.grid[row][col], win.grid[r][c] = win.grid[r][c], 0 # apply cell move
    win.row_white, win.col_white = r, c
    if check(): win.grid = []; victory() # start animation if victory
    else: show() # show new board configuration after cell displacement
# -----
def show():
    """show current game board by setting state defined for each grid cell"""
    for n in range(25): # loop over grid cells and change state of board cells
        row, col = n//5, n%5; win.board[row][col].state = win.grid[row][col]
# -----
def check() -> bool:
    """check victory by counting number of peer cells with same state"""
    for loop in range(25): # loop over all board cells
        if loop == 12: continue # skip central cell (should be empty, anyway)
        row, col = loop//5, loop%5 # get cell coords by Euclidian division
        peers = [win.grid[row+r][col+c] for r,c in ((1,0),(-1,0),(0,1),(0,-1))
            if 0 <= row+r <= 4 and 0 <= col+c <= 4] # get states for all peer cells
        if peers.count(win.grid[row][col]) not in (2,3): return False # no victory
    return True # all cells have 2 or 3 peers with same state ==> victory
# -----
def victory() -> None:
    """play victory animation"""
    win.board[rr(5)][rr(5)].state = rr(1,5) # change color of random grid cell
    if not win.grid: win.after(10, victory) # launch 'victory' every 10 ms
# =====
if __name__ == '__main__':
    main()
# =====

```

C6A_slide.py

```

# =====
"""SLIDE : a color-based slide puzzle game"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0" # play game with mouse
__date__ = "2022-11-12"
# =====
from ezTK import *
from random import shuffle, randrange as rr
# -----
def main():
    """create the main window and pack the widgets"""
    global win
    win = Win(title='SLIDE', op=5, click=on_click) # create main window
    colors = ('#FFF','#F00','#0F0','#07F','#FF0') # colors for board cells
    # -----
    frame = Frame(win, grow=False)
    Button(frame, text='SHUFFLE', width=9, command=on_shuffle)
    Button(frame, text='RESET', width=9, command=on_reset)

```

```

win.board = Frame(win, fold=5, op=0, border=1)
for n in range(25): # create all board cells as multi-state Brick widgets
    Brick(win.board, bg=colors, width=64, height=64, border=1)
# -----
on_shuffle(); win.loop() # shuffle board and start game
# -----
def on_shuffle() -> None:
    """callback function for the 'SHUFFLE' button"""
    # create list of 25 cells, shuffle it and call 'on_reset' to show new grid
    win.cells = 6*[1,2,3,4] + [0]; shuffle(win.cells); on_reset()
# -----
def on_reset() -> None:
    """callback function for the 'RESET' button"""
    # convert list of 25 cells into a 5x5 grid and show it on game board
    win.grid = [win.cells[n:n+5] for n in range(0,25,5)]; show()
# -----
def on_click(widget:object, code:str, mods:str) -> None:
    """callback function for all mouse click events"""
    if widget.master != win.board: return # wrong click (= not on a board cell)
    row, col = widget.index; move(row, col) # apply move on selected cell
# -----
def move(row:int, col:int) -> None:
    """move cell located at (row,col) to the position of the empty cell"""
    if col > 0 and win.grid[row][col-1] == 0: r,c = row,col-1 # left-side cell
    elif col < 4 and win.grid[row][col+1] == 0: r,c = row,col+1 # right-side cell
    elif row > 0 and win.grid[row-1][col] == 0: r,c = row-1,col # upper-side cell
    elif row < 4 and win.grid[row+1][col] == 0: r,c = row+1,col # lower-side cell
    else: return # move is not valid (no empty cell in neighboring cells)
    win.grid[row][col], win.grid[r][c] = 0, win.grid[r][col] # apply cell move
    if check(): win.grid = []; victory() # start animation if victory
    else: show() # show new board configuration after cell displacement
# -----
def show() -> None:
    """show current game board by setting state defined for each grid cell"""
    for n in range(25): # loop over grid cells and change state of board cells
        row, col = n//5, n%5; win.board[row][col].state = win.grid[row][col]
# -----
def check() -> bool:
    """check victory by counting number of peer cells with same state"""
    for loop in range(25): # loop over all board cells
        if loop == 12: continue # skip central cell (should be empty, anyway)
        row, col = loop//5, loop%5 # get cell coords by Euclidian division
        peers = [win.grid[row+r][col+c] for r,c in ((1,0),(-1,0),(0,1),(0,-1))
            if 0 <= row+r <= 4 and 0 <= col+c <= 4] # get states for all peer cells
        if peers.count(win.grid[row][col]) not in (2,3): return False # no victory
    return True # all cells have 2 or 3 peers with same state ==> victory
# -----
def victory() -> None:
    """play victory animation"""
    win.board[rr(5)][rr(5)].state = rr(1,5) # change color of random grid cell
    if not win.grid: win.after(10, victory) # launch 'victory' every 10 ms
# =====
if __name__ == '__main__':
    main()
# =====

```

C8A_maze.py

```

# =====
"""MAZE : animate walker in a maze"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "1.0" # use rectangles for walker animation
__date__ = "2022-11-12"

```

```

# =====
from ezTK import *
from ezCLI import read_csv
# -----
def main(index:int=0):
    """create the main window and pack the widgets"""
    global win
    maze, walk = read_files(index) # read content of data files for given index
    rows, cols = len(maze), len(maze[0]) # get maze size from data file
    # -----
    win = Win(title=f"MAZE : {cols}x{rows}", op=4, grow=False) # put size on title
    width, height = win.wininfo_screenwidth()-64, win.wininfo_screenheight()-64
    step = min(width/cols, height/rows) # compute step according to screen size
    win.canvas = Canvas(win, width=cols*step, height=rows*step)
    # -----
    colors = {'A': '#555', 'B': '#FFF'} # define one color for each maze symbol
    for row in range(rows): # loop over maze cells and draw one square per cell,
        for col in range(cols): # selecting color according to maze symbol (A or B)
            x, y, color = step*col, step*row, colors[maze[row][col]]
            win.canvas.create_rectangle(x, y, x+step, y+step, fill=color)
    # -----
    win.walk, win.step = walk, step; win.after(1000, tick); win.loop()
# -----
def read_files(index:int) -> (list,list):
    """return content of data files (maze file + walk file) for provided index"""
    mazefile, walkfile = f"maze{index}.csv", f"walk{index}.csv" # built filenames
    maze = read_csv(mazefile) # read maze file as a list of char lists
    walk = read_csv(walkfile) # read walk file as a list of integer pairs
    return maze, walk
# -----
def tick() -> None:
    """update walker position on canvas"""
    col, row = win.walk.pop(0) # get new walker position and remove it from list
    x, y = win.step*col, win.step*row # convert walker position to canvas coords
    # show new walker position by drawing a green (= #5F5) square on the canvas
    win.canvas.create_rectangle(x, y, x+win.step, y+win.step, fill='#5F5')
    if win.walk != []: win.after(50, tick) # schedule next step, if list not empty
# =====
if __name__ == '__main__':
    main(0) # show maze grid and animate walker for the provided maze index
# =====

```

C8B_maze.py

```

# =====
"""MAZE : animate walker in a maze"""
# =====
__author__ = "Christophe Schlick modified by Philippe Blasi"
__version__ = "2.0" # use rounded lines for walker animation
__date__ = "2022-11-12"
# =====
from ezTK import *
from ezCLI import read_csv
# -----
def main(index:int=0):
    """create the main window and pack the widgets"""
    global win
    maze, walk = read_files(index) # read content of data files for given index
    rows, cols = len(maze), len(maze[0]) # get maze size from data file
    # -----
    win = Win(title=f"MAZE : {cols}x{rows}", op=4, grow=False) # put size on title
    width, height = win.wininfo_screenwidth()-64, win.wininfo_screenheight()-64
    step = min(width/cols, height/rows) # compute step according to screen size
    win.canvas = Canvas(win, width=cols*step, height=rows*step)

```

```

# -----
colors = {'A': '#000', 'B': '#FFF'} # define one color for each maze symbol
for row in range(rows): # loop over maze cells and draw one square per cell,
    for col in range(cols): # selecting color according to maze symbol (A or B)
        x, y, color = col*step, row*step, colors[maze[row][col]]
        win.canvas.create_rectangle(x, y, x+step, y+step, fill=color, width=0)
# -----
x, y = walk[0]; win.xy = (x+0.5)*step, (y+0.5)*step # initial walker position
# -----
win.walk, win.step = walk, step; win.after(1000, tick); win.loop()
# -----
def read_files(index:int) -> (list,list):
    """return content of data files (maze file + walk file) for provided index"""
    mazefile, walkfile = f"maze{index}.csv", f"walk{index}.csv" # built filenames
    maze = read_csv(mazefile) # read maze file as a list of char lists
    walk = read_csv(walkfile) # read walk file as a list of integer pairs
    return maze, walk
# -----
def tick() -> None:
    """update walker position on canvas"""
    step = win.step; xa, ya = win.xy # get current walker position
    width, color = max(3, step//2), '#5F5' # set width and color for walker lines
    col, row = win.walk.pop(0) # get new walker position and remove it from list
    xb, yb = (col+0.5)*step, (row+0.5)*step # convert position into canvas coords
    win.xy = xb, yb # store new walker position and show it on canvas
    win.canvas.create_line(xa, ya, xb, yb, width=width, fill=color, caps='round')
    if win.walk != []: win.after(50, tick) # schedule next step, if list not empty
# =====
if __name__ == '__main__':
    main(0) # show maze grid and animate walker for the provided maze index
# =====

```

C10_csv.py

```

# =====
"""CSV : display a csv file content in a tclTk window, allow modifications and
saving"""
# =====
__author__ = "Philippe Blasi"
__version__ = "1.0"
__date__ = "2023-05-02"
# =====
from ezTK import *
from ezCLI import *
# -----
def main():
    """create the main window and pack the widgets"""
    global win
    win = Win(title='CSV FILE', grow=False, click=on_click)
    win.row, win.col = 0, 0
    frame=Frame(win, fold=2)
    Button(frame, text='OPEN FILE', width=12, command=on_open)
    Button(frame, text='SAVE FILE', width=12, command=on_save)
    win.label = Label(frame, text=f"Cell({win.row},{win.col})", width=10)
    win.entry = Entry(frame, width=10, command=on_entry)
    # -----
    Frame(win) # create an empty frame to store the size and name of the file
    Frame(win) # create an empty frame to store the grid
    # -----
    win.matrix = None
    win.grid = None
    win.loop()
# -----
def on_open() -> None:

```

```

"""callback for the "OPEN FILE" button"""
name = Dialog('open',title='OPEN FILE')
grid_tcl(name)
# -----
def on_save() -> None:
    """callback for the "SAVE FILE" button"""
    name = Dialog('save',title='SAVE FILE')

    if win.matrix != None:
        for row in range(win.rows): # loop over grid cells and copy each
            for col in range(win.cols): # value into the matrix cell in row and col
                win.matrix[row][col] = parse(win.grid[row][col]['text'])

    write_csv(name,win.matrix)
# -----
def on_click(widget:object, code:str, mods:str) -> None:
    """callback function for all mouse click events"""
    if widget.master != win.grid or win.matrix == None: return # wrong click (= .....
    .....not on a board cell)
    win.row, win.col = widget.index
    win.label['text'] = f"Cell({win.row},{win.col})"
    win.entry.state = win.grid[win.row][win.col]['text']
# -----
def on_entry() -> None:
    """callback function for the cell entry"""
    if win.grid[win.row][win.col]['text'] != win.entry.state:
        win.grid[win.row][win.col]['fg'] = '#FFF'
        win.grid[win.row][win.col]['bg'] = '#F00'
        win.grid[win.row][win.col]['text'] = win.entry.state
# -----
def grid_tcl(name:str) -> None:
    """create the grid within the main window and pack the widgets"""
    matrix = read_csv(name) # read content of data file name
    rows, cols = len(matrix), len(matrix[0]) # get matrix size from data file
    widths = [len(str(matrix[row][col])) for col in range(cols) for row in range.....
    .....(rows)]
    width = max(widths)+2 # get the maximum width for the cells of matrix
    # -----
    del win[2] # delete the frame containing the current grid
    del win[1] # delete the frame storing the size and name of the file
    # -----
    frame = Frame(win,op=3) # create a new frame to store the size and name
    short_name=name.split('/')[-1] # create the local name without the full path
    Label(frame, text=f'File name : {short_name}', grow=False)
    Label(frame, text=f'Number of rows : {rows}', grow=False)
    Label(frame, text=f'Number of cols : {cols}', grow=False)
    # -----
    grid = Frame(win, fold=cols) # create a new frame to store the new grid
    # -----
    for row in range(rows): # loop over matrix cells and create a Label for each
        for col in range(cols): # value of the matrix cell in row and col
            Label(grid,text=matrix[row][col],border=1,width=width)
    # -----
    # save in win for later use
    win.matrix = matrix; win.rows=rows; win.cols=cols; win.grid=grid
# =====
if __name__ == '__main__':
    main()
# =====

```