Interlude: Games

This interlude presents a short "game" that plays with the idea of logging. In this game, you use the log created by your typing to play the game. All play happens through logs. This idea is perhaps more "surface-level clever" than deep or trenchant, but it does force the player to focus on what is being logged, why those logs are happening, and their constituent parts.

As with all interludes, we start by importing every LIBRARY module that we will need throughout the program.

```
from random import choice, shuffle
from IPython.display import display, clear_output
from time import sleep
```

In this game, you are playing tic-tac-toe with logs. Perhaps the trickiest part of programming tic-tac-toe is figuring out whether the game has been won already. A human can look at a board and see if there are enough Xs or Os in a row, but a computer needs to "walk" through the board row by row, column by column, or, most tricky,

diagonal by diagonal. If this were a guaranteed 3×3 tic-tac-toe, we could just list all possible winning/losing/tying boards and check the current board against them, but we are making a variably sized board, so we cannot do that.

```
BOARD_SIZE = 4 # 1 is the smallest board, the biggest is limited by your screen
start_board = "."*(BOARD_SIZE*BOARD_SIZE) # the board is actually just a string of periods

# these are helper functions that translate between the x & y and the internal representation
# remember, the internal representation is just a simple string
xy2i = lambda x,y: x + (y * BOARD_SIZE)
i2xy = lambda i: (int(i % BOARD_SIZE), int(i / BOARD_SIZE))

# drawing the board, labeling the axes
def a_draw_board(a_board):
    # the "reversed" just makes it look more like cartesian coordinates
    # like (0,0) in bottom left corner
    for j in reversed(range(BOARD_SIZE)):
        print(str(j),end="|")
        for i in range(BOARD_SIZE):
            print(a_board[xy2i(i,j)], end="")
        print()
1 print("-+" + "-"*BOARD_SIZE)
    print(" |" + "".join(list(map(str,range(BOARD_SIZE)))))

# if i flip the whole board to its horizontal mirror to check for winners,
# it means i can do less work overall. tic-tac-toe works in a mirror.
def ab_hflip(a_board):
    output = ""
    for j in range(BOARD_SIZE):
        for i in range(BOARD_SIZE,0,-1):
            output += a_board[xy2i(i-1,j)]
    return output
```

```
def a_board_won(a_board):
    # horiz in a row [[(0,0),(0,1),(0,2)], . . .]
    horiz = [a_board[xy2i(0,j):xy2i(BOARD_SIZE,j)] for j in range(BOARD_SIZE)]
    # vert in a row [[(0,0),(1,0),(2,0)], . . .]
    vert = [“”.join([a_board[xy2i(i,j)] for j in range(BOARD_SIZE)]) for i in
    range(BOARD_SIZE)]
    # diag in a row [[(0,0),(1,1),(2,2)],[(2,0),(1,1),(0,2)]]
    diag = [“”.join(t_board[xy2i(i,i)] for i in range(BOARD_SIZE)) for t_board in
    [a_board,ab_hflip(a_board)]]
    all_combos = horiz + vert + diag # it has to be one of them!
    for combo in all_combos:
        if combo[0]!= “.” and combo == len(combo) * combo[0]:
            return combo[0]
    return False

def a_board_complete(a_board): # no moves exist if there are no empty
(i.e., “.”) spaces.
    if not a_board_won(a_board):
        return not any([c == “.” for c in a_board])
    return True

def a_play(a_board,x,y,player):
    a_board[xy2i(x,y)] = player
    return a_board
```

We need to test our implementation of tic-tac-toe before we implement the “game” element of the, uh, game. The following cell simulates tic-tac-toe, playing both sides until a win or a draw.

```
def valid_xy(x,y): # is the move on the board?
    return x >= 0 and y >= 0 and x < BOARD_SIZE and y < BOARD_SIZE

def insert_piece(a_board, index, piece): # this adds a piece to the middle of the
board
    return a_board[:index] + piece + a_board[index+1:]
# for our own sanity, we need to test out our tic-tac-toe
```

```
# implementation. this function simulates tic-tac-toe.
def a_sim_ttt():
    sim_board = start_board # start with an empty board
    player = "x" # ttt starts with x
    while not a_board_complete(sim_board): # end when the board is complete or
    won
        clear_output(wait = True) # to look nice, we want to animate the simulation
        valid_plays = [i for i in range(len(sim_board)) if "." == sim_board[i]]
        # where can we play?
        shuffle(valid_plays) # randomize all possible spots; do not pick, say, the
        top left
        sim_board = insert_piece(sim_board,valid_plays[0],player) # pick that
        random spot
        if "x" == player: # switch player here now that a piece is inserted
            player = "o"
        else:
            player = "x"
        a_draw_board(sim_board) # draw the board
        sleep(0.25) # wait for a quarter second so we can see the animation

a_sim_ttt()
```

```
>>>
3|x.xo
2|xox.
1|ooxx
0|oox.
-+--
 |0123
```

This is our game. In this game, we type something, it is logged, and that last log entry is parsed to generate an (x,y) coordinate and either an x or an o play. It's . . . hard. That said, the code contains all the answers to how to solve it. The rhetorical value is debatable, but the pedagogical value is in thinking through how to generate logs, how to modify code, and the joy of solving this little conceptual puzzle.

```
# you generate log entry
# every log entry maps to a "xo."
# then you place that piece
# the game is to understand map of log entry to x/o/.
# so you can win as x or o

def secret_piece(msg): # do not look
    return "xo."[ord(user_msg[-1]) % 3]

def secret_x(msg): # look away
    return (ord(user_msg[0]) % BOARD_SIZE)

def secret_y(msg): # nothing here
    return (ord(user_msg[1]) % BOARD_SIZE)

LOG = []
quit = False # are we done?
game_board = start_board
user_name = input("What's your name?") # Just to be nice
while not quit:
    winner = a_board_won(game_board)
    user_msg = input("Type here (q to quit):")
    if user_msg[0] == 'q': # we have to let people quit somehow
        quit = True
    elif winner: # or end the game
        LOG.append(f"WIN: {winner} wins!")
        quit = True
    elif len(user_msg) < 3 or len(user_msg) > 128:
        LOG.append(f"ERROR: Please type between 3 and 100 letters and/or
        numbers.")
    else:
        piece = secret_piece(user_msg) # this code must not do anything
        sx = secret_x(user_msg)
        sy = secret_y(user_msg)
        LOG.append(f"LOG: {user_name} typed {user_msg} which generates a
        {piece} at ({sx}, {sy})")
        # we have now added to the log
        if valid_xy(sx, sy): # and we will add the piece from the log
            game_board = insert_piece(game_board,xy2i(sx,sy),piece)
```

```
        else:
            LOG.append(f"ERROR: {piece} cannot be placed at ({sx},{sy})")
    clear_output(wait = True) # make it look kind of animated
    a_draw_board(game_board) # draw the board
    for line in LOG:
        print(line) # print the log (cleared every turn, so we need to reprint)
    sleep(0.25)
>>>
3|. . . .
2|. . . .
1|. . . .
0|. . . .
-+--
 |0123
```

The Left Hand of Data

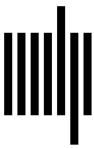
Designing Education Data for Justice

By: Matthew Berland, Antero Garcia

Citation:

The Left Hand of Data: Designing Education Data for Justice
By: Matthew Berland, Antero Garcia
DOI: 10.7551/mitpress/14381.001.0001
ISBN (electronic): 9780262377645
Publisher: The MIT Press
Published: 2024

The open access edition of this book was made possible by generous funding and support from MIT Press Direct to Open



The MIT Press

The MIT Press would like to thank the anonymous peer reviewers who provided comments on drafts of this book. The generous work of academic experts is essential for establishing the authority and quality of our publications. We acknowledge with gratitude the contributions of these otherwise uncredited readers.

This book was set in Stone Serif and Stone Sans by Westchester Publishing Services.

Library of Congress Cataloging-in-Publication Data

Names: Berland, Matthew, author. | Garcia, Antero, author.
Title: The left hand of data : designing education data for justice / Matthew Berland and Antero Garcia.
Description: Cambridge, Massachusetts : The MIT Press, [2024] | Includes bibliographical references and index.
Identifiers: LCCN 2023030088 (print) | LCCN 2023030089 (ebook) | ISBN 9780262547529 (paperback) | ISBN 9780262377652 (epub) | ISBN 9780262377645 (pdf)
Subjects: LCSH: Education—Data processing. | Education—Research. | Educational evaluation.
Classification: LCC LB1028.43 .B45 2024 (print) | LCC LB1028.43 (ebook) | DDC 370.285—dc23/eng/20230718
LC record available at https://lccn.loc.gov/2023030088
LC ebook record available at https://lccn.loc.gov/2023030089