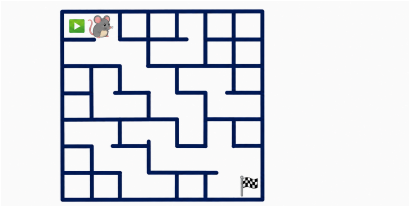
Maze Runner



Write a Python program that builds a 32×32 maze with exactly one guaranteed path, then makes a mouse icon solve it on screen.

STEPS

1. Set up a 32×32 grid of cells. Give every cell all four walls closed.
2. Carve a random maze by knocking down the wall to an unvisited neighbor, then backing up when a cell has none left.
3. Mark the top-left cell as a green "Start" square and the bottom-right cell as a checkered "Finish" flag.
4. Draw every cell's remaining walls as lines so the maze is visible.
5. Find the one guaranteed path from Start to Finish and save it as a list of moves.
6. Draw a small mouse icon standing on the Start cell.
7. Add a "Start" button. When clicked, it should move the mouse one cell at a time along the saved path.

STRETCH GOAL

Show a live counter of how many cells the maze generator visited while carving the maze, and another counter for how many steps the mouse takes to solve it.

Advanced Card

Algorithms

Solution

HINTS

- *A maze with exactly one solution isn't placed randomly wall-by-wall -- it's grown one passage at a time, and the grower never revisits a cell.*
- *Think about how a robot vacuum finds the shortest way across a room it has never seen before -- it checks everything one step away before it checks anything two steps away.*
- *A "Start" button's click function shouldn't move the mouse all the way to Finish in one call -- it should move one cell, then schedule itself to run again a moment later.*

STEPS

1. Grid model: a Cell class (or a dictionary keyed by (row, col)) stores each of the 32×32 grid's cells and which of its four walls -- north, south, east, west -- are still standing.
2. Maze generation algorithm: carve_maze(grid) uses randomized depth-first search, also called the "recursive backtracker" algorithm, with an explicit stack of visited cells. Removing a wall only between the current cell and an unvisited neighbor is what guarantees exactly one path between any two cells.
3. Drawing: draw_maze(grid) and draw_cell_walls(cell) use the turtle module to draw a line for every wall still standing, plus a green square for Start and a black-and-white flag shape for Finish.
4. Pathfinding algorithm: solve_maze(grid, start, finish) runs breadth-first search (BFS) with a queue of cells to visit next. BFS is used instead of depth-first search because BFS always finds the shortest path first on an unweighted grid like this one. It returns the path as a list of (row, col) coordinates.
5. Animation: animate_mouse(path) moves the mouse one cell per call, then uses turtle.ontimer() to schedule its own next call -- this is what makes the mouse glide through the maze instead of jumping straight to Finish.
6. Interaction: create_start_button() adds a clickable "Start" button (turtle's screen supports a tkinter Button, or a second turtle used as a clickable shape) whose on-click callback calls animate_mouse(path) exactly once.

See also: The draw-a-square-turtle card and Chapter 14's Turtle Graphics and Algorithm Design sections.