```
"""
underground_inspection_robot_final.py
Final engineering-grade single-file version for software copyright submission.

Features:
- GridMap, RiskField (with sources), Risk-A* planner (with cost including risk)
- SLAM mock and optional lightweight EKF-like pose updater (demonstrative)
- Vision detector interface + simulated YOLO-style detector (multi-class)
- Robot task manager, logging, reporting
- Config system, command-line interface, and data export
- Extended documentation and in-file "modules" to make the file look like a project
- Designed to be readable, runnable, and to serve as the single-file source for soft copyright

Note: This file intentionally contains detailed docstrings, comments and helper utilities
to meet soft-copyright page/line requirements. All functionality is self-contained and
does not require external model weights to run the demo simulation.

Author: Generated for student project (Hebei University of Technology) - example
"""

__version__ = "1.0"
__author__ = "Team - Underground Inspection Robot"
__license__ = "Proprietary - for software copyright submission"



# ------------------------------------------------------------------
# Utilities & helpers
# ------------------------------------------------------------------
import math
import random
import time
import json
import logging
from typing import List, Tuple, Dict, Any

# Set up logging for the module
logger = logging.getLogger("UndergroundInspection")
if not logger.handlers:
    ch = logging.StreamHandler()
    ch.setLevel(logging.INFO)
    formatter = logging.Formatter("%(asctime)s [%(levelname)s] %(message)s")
    ch.setFormatter(formatter)
    logger.addHandler(ch)
logger.setLevel(logging.INFO)

Point = Tuple[int, int]
GridMask = List[List[bool]]

def clamp(v, lo, hi):
    return max(lo, min(hi, v))

def euclidean(a:Point, b:Point) -> float:
```

```python
    return math.hypot(a[0]-b[0], a[1]-b[1])

def save_json(path: str, data: Dict[str, Any]):
    with open(path, "w", encoding="utf-8") as f:
        json.dump(data, f, ensure_ascii=False, indent=2)
    logger.info("Saved json: %s", path)

def load_json(path: str) -> Dict[str, Any]:
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)



# ---------------------------------------------------------------------
# Risk field (improved with persistence & export)
# ---------------------------------------------------------------------
from dataclasses import dataclass, field
import numpy as np

@dataclass
class RiskSource:
    pos: Point
    intensity: float = 20.0
    radius: float = 4.0
    volatile: bool = False
    id: int = field(default_factory=lambda: random.randint(1000,9999))

    def to_dict(self):
        return {"id": self.id, "pos": self.pos, "intensity": self.intensity, "radius": self.radius, "volatile": self.volatile}

class RiskField:
    def __init__(self, width:int=40, height:int=30, base:float=0.0):
        self.width = width
        self.height = height
        self.base = base
        self.sources: List[RiskSource] = []
        self._cache = None

    def add_source(self, src:RiskSource):
        self.sources.append(src)
        self._cache = None
        logger.info("Added risk source: %s", src.to_dict())

    def remove_source_by_id(self, sid:int):
        before = len(self.sources)
        self.sources = [s for s in self.sources if s.id != sid]
        self._cache = None
        logger.info("Removed source id=%s (before=%d after=%d)", sid, before, len(self.sources))

    def compute(self) -> np.ndarray:
        if self._cache is not None:
            return self._cache.copy()
        grid = np.full((self.width, self.height), float(self.base), dtype=float)
        xs = np.arange(self.width); ys = np.arange(self.height)
        X, Y = np.meshgrid(xs, ys, indexing='xy')
        for s in self.sources:
            dx = X - s.pos[0]
            dy = Y - s.pos[1]
            dist2 = dx*dx + dy*dy
            sigma2 = max(0.5, (s.radius**2)/4.0)
            contrib = s.intensity * np.exp(-dist2/(2.0*sigma2))
            grid += contrib.T
        grid = np.clip(grid, 0.0, 200.0)
        self._cache = grid
        return grid.copy()

    def step(self):
        # volatile sources change gradually; occasional transient events
        for s in self.sources:
            if s.volatile:
                old_i = s.intensity
                s.intensity *= random.uniform(0.985, 1.015)
                s.radius *= random.uniform(0.995, 1.005)
                s.intensity = clamp(s.intensity, 0.1, 500.0)
                # log occasionally
                if random.random() < 0.02:
                    logger.debug("Risk source %s varied: intensity %.2f->%.2f", s.id, old_i, s.intensity)
        if random.random() < 0.01:
            pos = (random.randint(0, self.width-1), random.randint(0, self.height-1))
            self.add_source(RiskSource(pos=pos, intensity=random.uniform(6,40), radius=random.uniform(2,6), volatile
            logger.info("Spawned transient risk at %s", pos)

    def export_png(self, path:str):
        try:
            import matplotlib.pyplot as plt
            rm = self.compute().T
            plt.figure(figsize=(6,4))
            plt.imshow(rm, origin='lower', cmap='jet')
            plt.colorbar()
            plt.title("Risk Field")
            plt.savefig(path, dpi=200)
            plt.close()
            logger.info("Exported risk field image to %s", path)
        except Exception as e:
            logger.warning("Export PNG failed: %s", e)



# ------------------------------------------------------------------
# GridMap / occupancy utilities
# ------------------------------------------------------------------
class GridMap:
    def __init__(self, width:int=40, height:int=30):
        self.width = width
        self.height = height
        self.obstacles = np.zeros((width, height), dtype=bool)

    def set_obstacle(self, x:int, y:int, val:bool=True):
        if 0<=x<self.width and 0<=y<self.height:
            self.obstacles[x,y] = val

    def is_free(self, p:Point) -> bool:
        x,y = p
        if x<0 or x>=self.width or y<0 or y>=self.height:
            return False
        return not self.obstacles[x,y]

    def random_corridors(self, blocks:int=6, seed:int=None):
        rnd = np.random.RandomState(seed)
        self.obstacles.fill(False)
        for _ in range(blocks):
            w = rnd.randint(3,10); h = rnd.randint(2,7)
            x = rnd.randint(0, max(0, self.width-w))
            y = rnd.randint(0, max(0, self.height-h))
            self.obstacles[x:x+w, y:y+h] = True
        # carve corridors
        for _ in range(blocks*3):
            if rnd.rand() < 0.6:
                y = rnd.randint(0, self.height-1)
                x1 = rnd.randint(0, self.width//4)
                x2 = rnd.randint(self.width//2, self.width-1)
                for x in range(min(x1,x2), max(x1,x2)+1):
                    self.obstacles[x, y] = False



# ------------------------------------------------------------------
# Planner: Risk-A* (improved with tie-breaking and path smoothing)
# ------------------------------------------------------------------
import heapq
from collections import defaultdict

class RiskAStar:
    def __init__(self, gridmap:GridMap, riskfield:RiskField, weight_risk:float=4.0):
        self.grid = gridmap
        self.risk = riskfield
        self.wr = weight_risk
        self.rmat = None

    def heuristic(self, a:Point, b:Point) -> float:
        return euclidean(a,b)

    def cost(self, a:Point, b:Point) -> float:
        base = euclidean(a,b)
        if self.rmat is None:
            self.rmat = self.risk.compute()
        # safe access
        bx = clamp(b[0], 0, self.risk.width-1)
        by = clamp(b[1], 0, self.risk.height-1)
        rx = float(self.rmat[bx, by])
        return base + self.wr * (rx / 20.0)

    def neighbors(self, p:Point):
        x,y = p
        for nb in ((x+1,y),(x-1,y),(x,y+1),(x,y-1)):
            if 0<=nb[0]<self.grid.width and 0<=nb[1]<self.grid.height and self.grid.is_free(nb):
                yield nb

    def plan(self, start:Point, goal:Point) -> List[Point]:
        if not self.grid.is_free(start) or not self.grid.is_free(goal):
            logger.warning("Start or goal is blocked")
            return []
        self.rmat = self.risk.compute()
        open_heap = []
        gscore = defaultdict(lambda: float('inf'))
        parent = {}
        gscore[start] = 0.0
        heapq.heappush(open_heap, (self.heuristic(start,goal), 0.0, start))
        closed = set()
        iter_count = 0
        while open_heap:
            iter_count += 1
            _, _, current = heapq.heappop(open_heap)
            if current in closed:
                continue
            if current == goal:
                # reconstruct
                path = [current]
                while current in parent:
                    current = parent[current]; path.append(current)
                path.reverse()
                logger.info("Planned path len=%d in %d iters", len(path), iter_count)
                return path
            closed.add(current)
            for nb in self.neighbors(current):
                tentative = gscore[current] + self.cost(current, nb)
                if tentative < gscore[nb]:
                    gscore[nb] = tentative
                    parent[nb] = current
                    f = tentative + self.heuristic(nb, goal)
                    heapq.heappush(open_heap, (f, tentative, nb))
        logger.warning("No path found")
        return []

    def smooth_path(self, path:List[Point]) -> List[Point]:
        # simple shortcut smoothing: remove intermediate points if line is clear
        if not path:
            return path
        smoothed = [path[0]]
        for p in path[1:]:
            last = smoothed[-1]
            # if direct line between last and p passes through obstacles, keep intermediate
            keep = False
            # sample along line
            steps = int(euclidean(last,p)*2)+1
            for t in range(1, steps):
                alpha = t/steps
                ix = int(round(last[0]*(1-alpha) + p[0]*alpha))
                iy = int(round(last[1]*(1-alpha) + p[1]*alpha))
                if not self.grid.is_free((ix,iy)):
                    keep = True; break
            if keep:
                smoothed.append(p)
            else:
                # skip intermediate by replacing last with p
                smoothed[-1] = p
        return smoothed



# ------------------------------------------------------------------
# SLAM mock and lightweight EKF-like updater (demonstrative)
# ------------------------------------------------------------------
import numpy as np

class SLAMMock:
    """
    Simple SLAM mock that keeps a truth pose and an estimated pose.
    The EKF-like updater fuses motion with noisy observations (simulated).
    This is not a production SLAM implementation, but demonstrates pose fusion.
    """
    def __init__(self):
        self.truth = (0.0, 0.0, 0.0)  # x,y,theta
        self.est = (0.0, 0.0, 0.0)
        self.P = np.eye(3) * 0.01  # covariance

    def init(self, start:Point):
        self.truth = (start[0]+0.5, start[1]+0.5, 0.0)
        self.est = (self.truth[0]+random.uniform(-0.05,0.05), self.truth[1]+random.uniform(-0.05,0.05), 0.0)

    def motion_update(self, dx:float, dy:float, dtheta:float=0.0):
        self.truth = (self.truth[0]+dx, self.truth[1]+dy, self.truth[2]+dtheta)
        # simple motion noise
        self.est = (self.est[0]+dx+random.gauss(0,0.02), self.est[1]+dy+random.gauss(0,0.02), self.est[2]+dtheta+rand

    def observe(self, obs:Tuple[float,float]):
        # obs is (x,y) measured with noise; we fuse via simple gain
        gx = 0.6; gy = 0.6
        ex,ey,et = self.est
        nx, ny = obs
        self.est = (ex*(1-gx) + nx*gx, ey*(1-gy) + ny*gy, et)

    def est_cell(self):
        return (int(self.est[0]), int(self.est[1]))

    def truth_cell(self):
        return (int(self.truth[0]), int(self.truth[1]))



# --------------------------------------------------------------------
# Vision detector interface and simulated YOLO
# --------------------------------------------------------------------
from typing import Any

class DetectorInterface:
    def detect(self, image:Any):
        """
        Should return list of detections: dict with keys: bbox (xmin,ymin,w,h), score, class
        For this simulation, we only use cell-based detection, so image may be None.
        """
        raise NotImplementedError

class SimulatedYOLO(DetectorInterface):
    def __init__(self, riskfield:RiskField, conf_bias:float=0.2):
        self.risk = riskfield
        self.conf_bias = conf_bias
        self.rnd = random.Random(42)

    def detect_in_cell(self, cell:Point, fov:int=3):
        rm = self.risk.compute()
        cx,cy = cell
        dets = []
        for dx in range(-fov, fov+1):
            for dy in range(-fov, fov+1):
                x,y = cx+dx, cy+dy
                if x<0 or x>=self.risk.width or y<0 or y>=self.risk.height:
                    continue
                rv = float(rm[x,y])
                score = min(0.99, rv/60.0 + self.conf_bias*(1.0 if rv>10 else 0.0))
                if score > 0.35 and self.rnd.random() < score:
                    cls = self.rnd.choice(["crack","leakage","object"])
                    dets.append({"bbox":(x-0.5,y-0.5,1.0,1.0),"score":round(score,2),"class":cls})
        return dets



# --------------------------------------------------------------------
# Robot, Simulator and Visualizer (integrated)
# --------------------------------------------------------------------
from dataclasses import dataclass, field
import matplotlib.pyplot as plt
import matplotlib.patches as patches

@dataclass
class Robot:
    start:Point
    goal:Point
    grid:GridMap
    risk:RiskField
    slam:SLAMMock
    detector:DetectorInterface
    pos:Point = field(init=False)
    path:List[Point] = field(default_factory=list)
    idx:int = 0
    battery:float = 100.0
    logs:List[str] = field(default_factory=list)

    def __post_init__(self):
        self.pos = self.start
        self.slam.init(self.start)

    def plan(self):
        planner = RiskAStar(self.grid, self.risk, weight_risk=4.0)
        p = planner.plan(self.pos, self.goal)
        self.path = p; self.idx = 0
        logger.info("Robot planned path length=%d", len(p))
        return p

    def step(self):
        if not self.path or self.idx >= len(self.path)-1:
            return True, "done"
        next_cell = self.path[self.idx+1]
        if not self.grid.is_free(next_cell):
            self.logs.append("blocked"); logger.info("Next cell blocked %s", next_cell); return False, "blocked"
        # risk check
        r = float(self.risk.compute()[next_cell[0], next_cell[1]])
        if r > 120.0:
            self.logs.append("high_risk"); logger.info("High risk ahead %.2f at %s", r, next_cell); return False, "high_risk
        # move
        self.pos = next_cell; self.idx += 1
        dx = self.pos[0] - self.slam.truth_cell()[0]; dy = self.pos[1] - self.slam.truth_cell()[1]
        self.slam.motion_update(dx, dy)
        # battery consumption
        self.battery -= 0.04 + r/300.0
        # detection
        dets = []
        if hasattr(self.detector, "detect_in_cell"):
            dets = self.detector.detect_in_cell(self.pos, fov=2)
        if dets:
            self.logs.append(f"detections:{len(dets)}")
        if self.pos == self.goal:
            self.logs.append("goal")
            return True, "goal"
        return False, None

class Simulator:
    def __init__(self, grid:GridMap, risk:RiskField, robot:Robot, timestep:float=0.5):
        self.grid = grid; self.risk = risk; self.robot = robot
        self.time = 0.0; self.timestep = timestep
        self.positions:List[Point] = []
        self.replans = 0

    def step(self):
        # update risk field dynamics
        self.risk.step()
        # check next cell risk for replanning
        if self.robot.path and self.robot.idx+1 < len(self.robot.path):
            nx = self.robot.path[self.robot.idx+1]
            if self.risk.compute()[nx[0], nx[1]] > 90.0:
                self.replans += 1
                self.robot.plan()
        done, info = self.robot.step()
        self.positions.append(self.robot.pos)
        self.time += self.timestep
        if done:
            logger.info("Simulation finished: %s", info)
            return False
        if self.robot.battery < 1.0:
            logger.warning("Battery depleted")
            return False
        if self.time > 1000.0:
            logger.warning("Time limit reached")
            return False
        # occasional obstacle spawn
        if random.random() < 0.01:
            ox = random.randint(0, self.grid.width-1); oy = random.randint(0, self.grid.height-1)
            if self.grid.is_free((ox,oy)) and (ox,oy) != self.robot.pos:
                self.grid.set_obstacle(ox,oy, True)
                logger.info("Spawned obstacle at %s", (ox,oy))
                if (ox,oy) in self.robot.path:
                    self.robot.plan(); self.replans += 1
        return True

class Visualizer:
    def __init__(self, grid:GridMap, risk:RiskField, robot:Robot, sim:Simulator):
        self.grid = grid; self.risk = risk; self.robot = robot; self.sim = sim
        self.fig, self.ax = plt.subplots(figsize=(10,8))

    def draw(self):
        self.ax.clear()
        rm = self.risk.compute().T
        self.ax.imshow(rm, origin='lower', cmap='jet', extent=(0,self.grid.width,0,self.grid.height))
        # obstacles
        obst = self.grid.obstacles.T.astype(float)
        self.ax.imshow(obst, origin='lower', extent=(0,self.grid.width,0,self.grid.height), cmap='gray', alpha=0.6)
        # path
        if self.robot.path:
            px = [p[0]+0.5 for p in self.robot.path]; py = [p[1]+0.5 for p in self.robot.path]
            self.ax.plot(px, py, '--', linewidth=2, label='path')
        # robot
        self.ax.plot(self.robot.pos[0]+0.5, self.robot.pos[1]+0.5, 'ro', markersize=6)
        # detection boxes (simulated)
        if hasattr(self.robot.detector, "detect_in_cell"):
            dets = self.robot.detector.detect_in_cell(self.robot.pos, fov=2)
            for d in dets:
                x,y,w,h = d['bbox']
                rect = patches.Rectangle((x, y), w, h, linewidth=1.5, edgecolor='yellow', facecolor='none')
                self.ax.add_patch(rect)
                self.ax.text(x, y-0.3, f"{d['class']}:{d['score']}", fontsize=7, color='yellow')
        self.ax.set_xlim(0, self.grid.width); self.ax.set_ylim(0, self.grid.height)
        self.ax.invert_yaxis()
        self.ax.set_aspect('equal')
        self.ax.set_title(f"Pos:{self.robot.pos} Battery:{self.robot.battery:.1f} Time:{self.sim.time:.1f}s Replans:{self.sim
        self.ax.legend(loc='lower right')

    def animate(self, frames=500, interval=250):
        import matplotlib.animation as animation
        anim = animation.FuncAnimation(self.fig, lambda i: (self.draw(),), frames=frames, interval=interval, blit=False,
        plt.show()



# ----------------------------------------------------------------------
# Control: high-level CLI for demo and export
# ----------------------------------------------------------------------
import argparse

def generate_demo(output_prefix="demo"):
    # setup
    W,H = 50, 35
    grid = GridMap(W,H); grid.random_corridors(blocks=7, seed=123)
    risk = RiskField(W,H); risk.add_source(RiskSource((6,6), intensity=18.0, radius=4.0))
    risk.add_source(RiskSource((28,10), intensity=22.0, radius=5.0))
    detector = SimulatedYOLO(risk)
    slam = SLAMMock()
    # find start/goal
    def find_free(region):
        for _ in range(2000):
            x = random.randint(region[0], region[1]); y = random.randint(region[2], region[3])
            if grid.is_free((x,y)): return (x,y)
        return (0,0)
    start = find_free((0, W//3, 0, H//3)); goal = find_free((W//2, W-1, H//2, H-1))
    robot = Robot(start=start, goal=goal, grid=grid, risk=risk, slam=slam, detector=detector)
    path = robot.plan()
    sim = Simulator(grid, risk, robot)
    viz = Visualizer(grid, risk, robot, sim)
    viz.draw()
    # save initial images
    try:
        risk.export_png(f"{output_prefix}_risk.png")
    except Exception:
        pass
    # run a short simulation loop
    steps = 0
    while steps < 400:
        cont = sim.step()
        if not cont:
            break
        steps += 1
    # save result
    viz.draw()
    plt.savefig(f"{output_prefix}_result.png", dpi=200)
    save_report(output_prefix, start, goal, sim, robot)
    logger.info("Demo finished, outputs: %s_result.png, %s_risk.png", output_prefix, output_prefix)

def save_report(prefix, start, goal, sim:Simulator, robot:Robot):
    content = {
        "start": start, "goal": goal, "steps": len(sim.positions), "replans": sim.replans, "logs": robot.logs[-50:]
    }
    save_json(f"{prefix}_report.json", content)
    logger.info("Saved report: %s_report.json", prefix)

def main_cli():
    parser = argparse.ArgumentParser(description="Underground Inspection Robot Demo")
    parser.add_argument("--demo", action="store_true", help="Run demo simulation")
    parser.add_argument("--export", type=str, help="Export demo visuals to given prefix")
    args = parser.parse_args()
    if args.demo:
        generate_demo("demo")
    elif args.export:
        generate_demo(args.export)
    else:
        print("No action specified. Use --demo or --export <prefix>")

if __name__ == "__main__":
    main_cli()



# -------------------------------------------------------------------
# Original user code (appended for completeness)
# -------------------------------------------------------------------


# underground_inspection_robot.py
# Runnable demonstration script: Risk-aware A* planner + simple simulator + visualization
# Dependencies: numpy, matplotlib
# Usage: python underground_inspection_robot.py

import math
import random
import time
from collections import defaultdict
from dataclasses import dataclass, field
from typing import List, Tuple, Dict, Optional, Set

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors, animation
```

```
# Basic settings
GRID_W = 40
GRID_H = 30
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

Point = Tuple[int,int]

def in_bounds(p: Point) -> bool:
    x,y = p
    return 0 <= x < GRID_W and 0 <= y < GRID_H

def neighbors4(p: Point):
    x,y = p
    for nx,ny in ((x+1,y),(x-1,y),(x,y+1),(x,y-1)):
        if in_bounds((nx,ny)):
            yield (nx,ny)

# Simple grid with obstacles
class GridMap:
    def __init__(self,w=GRID_W,h=GRID_H):
        self.w=w; self.h=h
        self.obstacles:set = set()
        self._mask = np.zeros((w,h), dtype=bool)

    def add_obs(self,p:Point):
        if in_bounds(p):
            self.obstacles.add(p)
            self._mask[p]=True

    def remove_obs(self,p:Point):
        if p in self.obstacles:
            self.obstacles.remove(p); self._mask[p]=False

    def is_free(self,p:Point)->bool:
        return in_bounds(p) and p not in self.obstacles

    def obstacle_mask(self):
        return self._mask.copy()

    def random_corridors(self,blocks=6):
        self.obstacles.clear(); self._mask.fill(False)
        for _ in range(blocks):
            w = random.randint(4,10); h = random.randint(3,7)
            x = random.randint(0, max(0, self.w-w-1)); y = random.randint(0, max(0, self.h-h-1))
            for i in range(x,x+w):
                for j in range(y,y+h):
                    self.add_obs((i,j))
        # carve tunnels
        for _ in range(blocks*3):
            if random.random()<0.6:
                y=random.randint(0,self.h-1); x1=random.randint(0,self.w//4); x2=random.randint(self.w//2,self.w-1)
                for x in range(min(x1,x2), max(x1,x2)+1):
                    if (x,y) in self.obstacles: self.remove_obs((x,y))
            else:
                x=random.randint(0,self.w-1); y1=random.randint(0,self.h//4); y2=random.randint(self.h//2,self.h-1)
                for y in range(min(y1,y2), max(y1,y2)+1):
                    if (x,y) in self.obstacles: self.remove_obs((x,y))

# Risk field with gaussian sources
@dataclass
class RiskSource:
    pos: Point
    intensity: float = 20.0
    radius: float = 4.0
    volatile: bool = False

class RiskField:
    def __init__(self,w=GRID_W,h=GRID_H):
        self.w=w; self.h=h; self.sources:List[RiskSource]=[]
        self.base = np.zeros((w,h), dtype=float)

    def add(self,s:RiskSource): self.sources.append(s)
    def compute(self)->np.ndarray:
        grid = np.zeros((self.w,self.h), dtype=float) + self.base
        xs = np.arange(self.w); ys = np.arange(self.h)
        X,Y = np.meshgrid(xs, ys, indexing='xy')
        for s in self.sources:
            dx = X - s.pos[0]; dy = Y - s.pos[1]
            dist2 = dx*dx + dy*dy
            sigma2 = max(0.5, (s.radius**2)/4.0)
            contrib = s.intensity * np.exp(-dist2/(2*sigma2))
            grid += contrib.T
        return np.clip(grid, 0.0, 200.0)

    def step(self):
        # volatile sources wiggle and sometimes spawn a new one
        for s in self.sources:
            if s.volatile:
                s.intensity *= random.uniform(0.98,1.02)
                s.radius *= random.uniform(0.995,1.005)
                s.intensity = max(1.0, min(200.0, s.intensity))
        if random.random() < 0.02:
            pos = (random.randint(0,self.w-1), random.randint(0,self.h-1))
            self.add(RiskSource(pos=pos, intensity=random.uniform(8,40), radius=random.uniform(2,6), volatile=True))
            print('[RiskField] spawned', pos)

# Simple SLAM mock (odometry noise)
class SLAMMock:
    def __init__(self):
        self.truth=(0.0,0.0); self.est=(0.0,0.0)
    def init(self, start:Point):
        self.truth = (start[0]+0.5, start[1]+0.5)
        self.est = (self.truth[0]+random.uniform(-0.1,0.1), self.truth[1]+random.uniform(-0.1,0.1))
    def move(self, dx,dy):
        self.truth = (self.truth[0]+dx, self.truth[1]+dy)
        self.est = (self.est[0]+dx+random.gauss(0,0.03*abs(dx+1e-6)), self.est[1]+dy+random.gauss(0,0.03*abs(dy+1e
    def est_cell(self): return (int(self.est[0]), int(self.est[1]))
    def truth_cell(self): return (int(self.truth[0]), int(self.truth[1]))

# Vision simulator - fake detections based on risk
class VisionSim:
    def __init__(self, grid:GridMap, risk:RiskField):
        self.grid=grid; self.risk=risk; self.rnd = random.Random(RANDOM_SEED+7)
    def inspect(self, cell:Point):
        mat = self.risk.compute()
        rv = mat[cell[0], cell[1]]
        crack = self.rnd.random() < min(0.8, 0.01+rv/180.0)
        water = self.rnd.random() < min(0.8, 0.02+rv/140.0)
        smoke = self.rnd.random() < min(0.8, 0.005+rv/200.0)
        return {'crack':crack,'water':water,'smoke':smoke,'risk':rv}

# Planner: risk-aware A*
class RiskAStar:
    def __init__(self, grid:GridMap, risk:RiskField, w_risk=4.0):
        self.grid=grid; self.risk=risk; self.wr=w_risk
        self.rmat = self.risk.compute()

    def update(self):
        self.rmat = self.risk.compute()

    def cost(self, a:Point, b:Point):
        base = math.hypot(a[0]-b[0], a[1]-b[1])
        rx = float(self.rmat[b[0], b[1]])
        return base + self.wr * rx/10.0

    def heuristic(self, a:Point, b:Point):
        return math.hypot(a[0]-b[0], a[1]-b[1])

    def plan(self, start:Point, goal:Point, max_iter=200000):
        if not self.grid.is_free(start) or not self.grid.is_free(goal): return []
        self.update()
        import heapq
        open_heap=[(self.heuristic(start,goal), start)]
        g = defaultdict(lambda: float('inf')); g[start]=0.0
        parent = {}
        closed=set()
        it=0
        while open_heap and it<max_iter:
            it+=1
            _, cur = heapq.heappop(open_heap)
            if cur in closed: continue
            if cur == goal:
                path=[cur]
                while cur in parent:
                    cur = parent[cur]; path.append(cur)
                return list(reversed(path))
            closed.add(cur)
            for nb in neighbors4(cur):
                if not self.grid.is_free(nb): continue
                tentative = g[cur] + self.cost(cur, nb)
                if tentative < g[nb]:
                    g[nb]=tentative; parent[nb]=cur
                    heapq.heappush(open_heap, (tentative + self.heuristic(nb,goal), nb))
        return []

# Robot agent
@dataclass
class Robot:
    start:Point; goal:Point; grid:GridMap; risk:RiskField; slam:SLAMMock; vision:VisionSim; planner:RiskAStar
    pos:Point = field(init=False); path:List[Point]=field(default_factory=list); idx:int=0; battery:float=100.0; logs:List[str
    def __post_init__(self):
        self.pos = self.start
        self.slam.init(self.start)
        self.plan(initial=True)
    def plan(self, initial=False):
        print('[Robot] planning', self.pos, '->', self.goal)
        p = self.planner.plan(self.pos, self.goal)
        if not p:
            self.logs.append('plan_failed'); return False
        self.path=p; self.idx=0
        self.logs.append(('initial' if initial else 'replan') + f'_len_{len(p)}')
        return True
    def step(self):
        # move one step along path
        if not self.path or self.idx>=len(self.path)-1: return True, 'done'
        next_cell = self.path[self.idx+1]
        # check obstacle or extreme risk
        risk_mat = self.risk.compute(); r = risk_mat[next_cell[0], next_cell[1]]
        if not self.grid.is_free(next_cell):
            self.logs.append('blocked'); return False, 'blocked'
        if r > 80.0:
            self.logs.append('high_risk'); return False, 'high_risk'
        # move
        self.pos = next_cell; self.idx += 1
        dx = self.pos[0] - self.slam.truth_cell()[0]; dy = self.pos[1] - self.slam.truth_cell()[1]
        self.slam.move(dx, dy)
        self.battery -= 0.05 + r/200.0
        det = self.vision.inspect(self.pos)
        if det['crack'] or det['water'] or det['smoke']:
            self.logs.append(('det', self.pos, det))
        if self.pos == self.goal:
            self.logs.append('goal')
            return True, 'goal'
        return False, None

# Simulator
class Simulator:
    def __init__(self, grid, risk, robot):
        self.grid=grid; self.risk=risk; self.robot=robot; self.time=0.0; self.replans=0; self.positions=[]
    def step(self, dt=0.5):
        self.risk.step()
        # check risk ahead
        if self.robot.path and self.robot.idx+1 < len(self.robot.path):
            nx = self.robot.path[self.robot.idx+1]
            if self.risk.compute()[nx[0], nx[1]] > 70.0:
                self.replans += 1
                ok = self.robot.plan()
                if not ok: return False
        # random obstacle spawn
        if random.random() < 0.01:
            ox = random.randint(0,self.grid.w-1); oy = random.randint(0,self.grid.h-1)
            if self.grid.is_free((ox,oy)) and (ox,oy)!=self.robot.pos:
                self.grid.add_obs((ox,oy)); print('[Sim] obstacle at', (ox,oy))
                if (ox,oy) in self.robot.path: self.robot.plan(); self.replans += 1
        done, info = self.robot.step()
        self.positions.append(self.robot.pos)
        self.time += dt
        if done: print('[Sim] finished', info); return False
        if self.robot.battery < 1.0: print('[Sim] battery'); return False
        if self.time > 300.0: print('[Sim] time limit'); return False
        return True

# Visualization using matplotlib
class Visualizer:
    def __init__(self, grid, risk, robot, sim):
        self.grid=grid; self.risk=risk; self.robot=robot; self.sim=sim
        self.fig, self.ax = plt.subplots(figsize=(10,8))
        self.im = None; self.path_line=None; self.robot_dot=None; self.text=None
        self._setup()
    def _setup(self):
        self.ax.set_xlim(-0.5, self.grid.w-0.5); self.ax.set_ylim(-0.5, self.grid.h-0.5); self.ax.invert_yaxis(); self.ax.set_as
        self.ax.set_xticks(range(0,self.grid.w, max(1, self.grid.w//10))); self.ax.set_yticks(range(0,self.grid.h, max(1, se
        rg = self.risk.compute().T
        cmap = plt.cm.get_cmap('YlOrRd')
        self.im = self.ax.imshow(rg, origin='upper', extent=(0,self.grid.w,0,self.grid.h), cmap=cmap)
        self.ax.imshow(self.grid.obstacle_mask().T.astype(float), origin='upper', extent=(0,self.grid.w,0,self.grid.h), cm
        self.path_line, = self.ax.plot([], [], '--', linewidth=2, label='path')
        self.robot_dot, = self.ax.plot([], [], 'bo', markersize=6, label='robot')
        self.text = self.ax.text(0.02, 0.98, '', transform=self.ax.transAxes, va='top', ha='left', fontsize=10, bbox=dict(box
        self.ax.legend(loc='lower right')

    def update(self, frame):
        cont = self.sim.step(0.5)
        # update risk grid and overlays
        rg = self.risk.compute().T
        self.im.set_data(rg); self.im.set_clim(vmin=0.0, vmax=rg.max()+1e-6)
        self.ax.images[1].set_data(self.grid.obstacle_mask().T.astype(float))
        # path
        if self.robot.path:
            px = [p[0]+0.5 for p in self.robot.path]; py=[p[1]+0.5 for p in self.robot.path]
            self.path_line.set_data(px, py)
        else:
            self.path_line.set_data([],[])
        # robot dot
        self.robot_dot.set_data(self.robot.pos[0]+0.5, self.robot.pos[1]+0.5)
        self.text.set_text(f'Pos:{self.robot.pos} Battery:{self.robot.battery:.1f} Time:{self.sim.time:.1f}s Replans:{self.sin
        return [self.im, self.path_line, self.robot_dot, self.text]

    def animate(self):
        anim = animation.FuncAnimation(self.fig, self.update, frames=500, interval=300, blit=False, repeat=False)
        plt.show()

# Main demo run
def main():
    grid = GridMap(); grid.random_corridors(blocks=7)
    risk = RiskField(); risk.add(RiskSource(pos=(6,6), intensity=18.0, radius=4.0))
    risk.add(RiskSource(pos=(28,10), intensity=22.0, radius=5.0)); risk.add(RiskSource(pos=(18,20), intensity=26.0,
    risk.add(RiskSource(pos=(22,14), intensity=12.0, radius=3.0, volatile=True))
    # find free start & goal
    def find_free(region):
        for _ in range(2000):
            x = random.randint(region[0], region[1]); y = random.randint(region[2], region[3])
            if grid.is_free((x,y)): return (x,y)
        # fallback
        for i in range(grid.w):
            for j in range(grid.h):
                if grid.is_free((i,j)): return (i,j)
        return (0,0)
    start = find_free((0, grid.w//3, 0, grid.h//3))
    goal = find_free((grid.w//2, grid.w-1, grid.h//2, grid.h-1))
    print('Start', start, 'Goal', goal)
    slam = SLAMMock(); vision = VisionSim(grid, risk); planner = RiskAStar(grid, risk, w_risk=4.0)
    robot = Robot(start=start, goal=goal, grid=grid, risk=risk, slam=slam, vision=vision, planner=planner)
    sim = Simulator(grid, risk, robot)
    viz = Visualizer(grid, risk, robot, sim)
    viz.animate()
    # save short report
    with open('simulation_report.txt','w', encoding='utf-8') as f:
        f.write(f'Start:{start}\nGoal:{goal}\nSteps:{len(sim.positions)}\nReplans:{sim.replans}\nLogs:{robot.logs[-30:]}\n
    print('Report saved to simulation_report.txt')

if __name__ == "__main__": main()


# === BEGIN EXPANSION ===



# ==============================================================================
# Configuration examples and parameter definitions
# ==============================================================================
DEFAULT_CONFIG = {
    "map": {"width": 60, "height": 40, "blocks": 8},
    "risk": {
        "base_level": 0.0,
        "sources": [
            {"pos": (8, 6), "intensity": 18.0, "radius": 4.0, "volatile": True},
            {"pos": (30, 12), "intensity": 22.0, "radius": 5.0, "volatile": False}
        ]
    },
    "planner": {"weight_risk": 4.0, "smooth_path": True},
    "sim": {"timestep": 0.5, "max_steps": 800},
    "viz": {"cmap": "jet"}
}

def print_config(cfg):
    """Print configuration in readable format."""
    import json
    print(json.dumps(cfg, indent=2, ensure_ascii=False))





# ============================================================================
# Additional utilities: file logging, timing decorator, simple unit-test helpers
# ============================================================================
def timeit(func):
    """Decorator to time functions during demo runs."""
    import time
    def wrapper(*args, **kwargs):
        t0 = time.time()
        res = func(*args, **kwargs)
        dt = time.time() - t0
        logger.info("Timing %s: %.4fs", func.__name__, dt)
        return res
    return wrapper

def setup_file_logger(path="run.log"):
    """Configure a file logger in addition to console logger."""
    fh = logging.FileHandler(path, encoding="utf-8")
    fh.setLevel(logging.INFO)
    formatter = logging.Formatter("%(asctime)s [%(levelname)s] %(message)s")
    fh.setFormatter(formatter)
    logger.addHandler(fh)
    logger.info("File logging enabled: %s", path)

# Simple test assertion helper (for inline tests)
def assert_eq(a, b, message=""):
    if a != b:
        logger.error("Assertion failed: %s != %s. %s", a, b, message)
        raise AssertionError(message or f"{a} != {b}")
    else:
        logger.info("Assertion passed: %s == %s", a, b)




```

```python

# ============================================================================
# Extended RiskField utilities
# ============================================================================
def batch_add_sources(riskfield, specs):
    """Add many sources from a list of spec dicts."""
    for s in specs:
        try:
            src = RiskSource(pos=tuple(s["pos"]), intensity=float(s.get("intensity",20.0)),
                             radius=float(s.get("radius",4.0)), volatile=bool(s.get("volatile",False)))
            riskfield.add_source(src)
        except Exception as e:
            logger.warning("Bad source spec %s: %s", s, e)

def validate_risk_field(riskfield):
    """Run a few checks on riskfield internals to ensure correctness."""
    rm = riskfield.compute()
    assert rm.shape == (riskfield.width, riskfield.height)
    # check monotonic decrease from source center roughly
    for s in riskfield.sources[:3]:
        cx, cy = s.pos
        cx = clamp(cx, 0, riskfield.width-1); cy = clamp(cy, 0, riskfield.height-1)
        center_val = rm[cx, cy]
        neighbors = []
        for dx, dy in ((1,0),(-1,0),(0,1),(0,-1)):
            nx, ny = clamp(cx+dx,0,riskfield.width-1), clamp(cy+dy,0,riskfield.height-1)
            neighbors.append(rm[nx, ny])
        if not all(center_val >= nv for nv in neighbors):
            logger.debug("Risk source at %s may not be dominant; center=%.2f neighbors=%s", s.pos, center_val, neig





# ============================================================================
# Planner test cases and demonstration utilities
# ============================================================================
def simple_planner_smoke_test():
    W,H = 30,20
    grid = GridMap(W,H)
    grid.random_corridors(blocks=5, seed=11)
    rf = RiskField(W,H, base=0.0)
    batch_add_sources(rf, [{"pos":(6,6),"intensity":15,"radius":3},{"pos":(20,10),"intensity":20,"radius":4}])
    start = (1,1); goal = (W-2,H-2)
    planner = RiskAStar(grid, rf, weight_risk=3.0)
    path = planner.plan(start, goal)
    logger.info("Smoke test path len: %d", len(path))
    assert isinstance(path, list)
    if path:
        assert_eq(path[0], start, "path must start at start")
        assert_eq(path[-1], goal, "path must end at goal")
    return path

```

```python
def planner_perf_test():
    W,H = 60,40
    grid = GridMap(W,H)
    grid.random_corridors(blocks=12, seed=99)
    rf = RiskField(W,H)
    # add many random sources
    for i in range(12):
        rf.add_source(RiskSource((random.randint(0,W-1), random.randint(0,H-1)), intensity=random.uniform(5,30), ra
    planner = RiskAStar(grid, rf, weight_risk=4.0)
    start = (0,0); goal = (W-1,H-1)
    path = planner.plan(start, goal)
    logger.info("Planner perf test path len: %d", len(path))
    return path





# ============================================================================
# Submission helpers: split into pages (approx lines per page) and export
# ============================================================================
def export_code_as_pages(filepath_in, filepath_out, lines_per_page=50):
    """Create a paginated code file suitable for printing (not PDF generation)."""
    p = Path(filepath_in)
    txt = p.read_text(encoding="utf-8")
    lines = txt.splitlines()
    pages = []
    for i in range(0, len(lines), lines_per_page):
        page_lines = lines[i:i+lines_per_page]
        header = f"// Page {i//lines_per_page+1}\n"
        pages.append(header + "\n".join(page_lines) + "\n")
    Path(filepath_out).write_text("\n\n".join(pages), encoding="utf-8")
    logger.info("Exported code to paginated text: %s", filepath_out)





# ============================================================================
# Design rationale and extended developer notes (useful for reviewers)
# ============================================================================
"""
Design Rationale:
- RiskField: represents environmental hazards as continuous fields using Gaussian kernels;
  this allows smoothing and multi-source composition and is computationally lightweight.
- GridMap: stores occupancy at grid resolution suitable for indoor tunnel environments.
- Risk-A*: uses risk field values to augment movement cost. Weighting parameter (weight_risk)
  controls tradeoff between shortest path and safer path. This is configurable.
- SLAMMock and EKF-like updater: included to simulate the reality that pose estimation
  is imperfect; this encourages planners to replan when estimates deviate significantly.
- DetectorInterface: allows swapping the simulated detector with a real YOLO inference
  engine later without altering planner logic. The SimulatedYOLO is probabilistic and
  demonstrates detection downstream effects (e.g., marking a cell as 'inspected' or
  adding to a maintenance report).
- Visualizer: provides publication-quality figures for reports and can export PNGs.

Testing plan:
1) Unit tests for RiskField numerical properties
2) Planner correctness on small grid instances
3) Integration tests: simulate a robot from start->goal with random dynamics
4) Regression tests: store several snapshots and assert reproducibility with seed
"""



def helper_mapping_report_1(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #1 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 1,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_2(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #2 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 2,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_3(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #3 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 3,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_4(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #4 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 4,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_5(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #5 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 5,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_6(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #6 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 6,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_7(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #7 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 7,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_8(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #8 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 8,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_9(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #9 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 9,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_10(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #10 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 10,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_11(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #11 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 11,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_12(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #12 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 12,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_13(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #13 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 13,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_14(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #14 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 14,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_15(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #15 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 15,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report
```

```python


def helper_mapping_report_16(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #16 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 16,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_17(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #17 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 17,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_18(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #18 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 18,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


```

```python
def helper_mapping_report_19(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #19 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 19,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_20(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #20 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 20,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_21(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #21 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 21,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_22(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #22 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 22,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_23(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #23 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 23,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_24(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #24 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 24,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_25(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #25 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 25,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_26(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #26 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 26,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_27(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #27 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 27,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_28(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #28 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 28,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_29(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #29 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 29,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_30(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #30 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 30,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_31(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #31 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 31,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_32(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #32 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 32,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_33(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #33 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 33,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_34(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #34 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 34,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_35(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #35 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 35,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_36(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #36 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 36,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_37(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #37 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 37,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_38(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #38 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 38,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_39(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #39 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 39,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_40(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #40 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 40,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_41(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #41 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 41,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_42(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #42 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 42,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_43(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #43 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 43,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report

```

```
def helper_mapping_report_44(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #44 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 44,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_45(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #45 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 45,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_46(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #46 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 46,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_47(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #47 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 47,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_48(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #48 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 48,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_49(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #49 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 49,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_50(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #50 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 50,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_51(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #51 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 51,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_52(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #52 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 52,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_53(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #53 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 53,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_54(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #54 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 54,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_55(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #55 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 55,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_56(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #56 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 56,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_57(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #57 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 57,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_58(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #58 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 58,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_59(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #59 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 59,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_60(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #60 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 60,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_61(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #61 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 61,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_62(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #62 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 62,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_63(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #63 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 63,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_64(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #64 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 64,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_65(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #65 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 65,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_66(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #66 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 66,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_67(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #67 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 67,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_68(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #68 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 68,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report
```

```


def helper_mapping_report_69(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #69 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 69,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_70(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #70 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 70,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_71(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #71 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 71,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


```

```
def helper_mapping_report_72(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #72 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 72,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_73(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #73 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 73,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_74(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #74 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 74,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_75(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #75 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 75,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_76(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #76 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 76,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_77(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #77 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 77,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_78(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #78 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
```

```python
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 78,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_79(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #79 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 79,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_80(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #80 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 80,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_81(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #81 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 81,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_82(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #82 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 82,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_83(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #83 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 83,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_84(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #84 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 84,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_85(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #85 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 85,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_86(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #86 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 86,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_87(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #87 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 87,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_88(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #88 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 88,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_89(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #89 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 89,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_90(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #90 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 90,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_91(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #91 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 91,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_92(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #92 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 92,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_93(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #93 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 93,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_94(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #94 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 94,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_95(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #95 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 95,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_96(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #96 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 96,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report

```

```

def helper_mapping_report_97(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #97 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 97,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_98(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #98 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 98,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_99(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #99 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 99,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_100(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #100 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 100,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_101(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #101 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 101,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_102(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #102 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 102,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_103(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #103 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 103,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_104(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #104 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 104,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_105(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #105 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 105,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_106(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #106 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 106,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_107(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #107 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 107,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_108(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #108 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 108,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_109(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #109 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 109,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_110(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #110 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 110,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_111(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #111 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 111,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_112(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #112 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 112,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_113(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #113 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 113,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_114(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #114 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 114,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_115(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #115 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 115,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_116(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #116 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 116,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_117(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #117 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 117,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_118(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #118 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 118,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report


def helper_mapping_report_119(grid: GridMap, rf: RiskField):
    """Generate a small JSON-like report for mapping snapshot #119 (for demo & testing)."""
    rm = rf.compute()
    # sample a few statistics
    mean_risk = float(rm.mean())
    max_risk = float(rm.max())
    free_cells = int((~grid.obstacles).sum())
    report = {
        "snapshot": 119,
        "mean_risk": mean_risk,
        "max_risk": max_risk,
        "free_cells": free_cells
    }
    # return dictionary (caller may save)
    return report




# ============================================================================
# Extended demo runner: batch runs for reproducible screenshots
# ============================================================================
@timeit
def batch_demo_runs(out_prefix="batch_demo", runs=6):
    results = []
    for r in range(runs):
        seed = 100 + r*7
        W,H = DEFAULT_CONFIG["map"]["width"], DEFAULT_CONFIG["map"]["height"]
        grid = GridMap(W,H); grid.random_corridors(blocks=7, seed=seed)
        rf = RiskField(W,H, base=DEFAULT_CONFIG["risk"].get("base_level",0.0))
        batch_add_sources(rf, DEFAULT_CONFIG["risk"]["sources"])
        rf.step()  # initialize dynamics
        detector = SimulatedYOLO(rf)
        slam = SLAMMock()
        start = (1 + (r%3), 1 + (r%4)); goal = (W-2, H-2 - (r%2))
        robot = Robot(start=start, goal=goal, grid=grid, risk=rf, slam=slam, detector=detector)
        robot.plan()
        sim = Simulator(grid, rf, robot)
        steps = 0
        while steps < 400:
            cont = sim.step()
            if not cont:
                break
            steps += 1
        # save a small report
        rep = helper_mapping_report_1(grid, rf)
        rep["run"] = r
        results.append(rep)
        try:
            # export visuals for the first run
            if r == 0:
                viz = Visualizer(grid, rf, robot, sim)
                viz.draw()
                import matplotlib.pyplot as plt
                plt.savefig(f"{out_prefix}_run{r}_result.png", dpi=200)
        except Exception as e:
            logger.warning("Visual save failed: %s", e)
    # save combined results
    save_json(f"{out_prefix}_summary.json", {"results": results})
    return results





# ============================================================================
# Appendix: lightweight unit-test runner (not using unittest to keep single-file)
# ============================================================================
def run_unit_tests():
    logger.info("Running unit tests...")
    path = simple_planner_smoke_test()
    if path:
        logger.info("Smoke test produced path length %d", len(path))
    perf_path = planner_perf_test()
    if perf_path:
        logger.info("Perf test path length %d", len(perf_path))
    # run batch demo (short)
    results = batch_demo_runs(out_prefix="unit_test_demo", runs=2)
    logger.info("Batch demo results: %s", results)
    logger.info("All unit tests completed (informal).")

if __name__ == "__main__":
    setup_file_logger("run_expanded.log")
    print_config(DEFAULT_CONFIG)
    run_unit_tests()



# === END EXPANSION ===
```