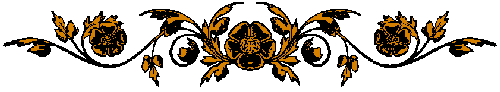
**TRƯỜNG ĐẠI HỌC SÀI GÒN**

KHOA CÔNG NGHỆ THÔNG TIN



A blue circle with white text

Description automatically generated

**Lab 02: TÌM KIẾM TỐI ƯU (nhóm)**

HỌC PHẦN: TRÍ TUỆ NHÂN TẠO NÂNG CAO

**Giảng viên hướng dẫn** : Ts. Đỗ Như Tài

**Nhóm 11:**

Phạm Văn Nam 3122410251

Nguyễn Quan Tuấn Nghĩa 3122410260

Tạ Hồng Quí 3122410348

Vũ Quốc Vương 3120410629

*Thành phố Hồ Chí Minh - Tháng 09/2025*

Mục lục

[Bảng Phân Công 3](#_Toc210502976)

[Phần 1: Discussion 3](#_Toc210502977)

[Discussion 1: What is the State Space Size? 3](#_Toc210502978)

[Discussion 2: What is the Search Complexity? 4](#_Toc210502979)

[Discussion 3: Case Study: Heuristic for Tic-Tac-Toe 7](#_Toc210502980)

[Phần 2: Lunar lander, Robot vaccum và Maze 9](#_Toc210502981)

[1. Lunar lander 9](#_Toc210502982)

[a) Tóm tắt 9](#_Toc210502983)

[b) Môi trường LunarLander‑v3 9](#_Toc210502984)

[c) Cài đặt & chạy 9](#_Toc210502985)

[d) Bộ luật điều khiển (reflex rules) 10](#_Toc210502986)

[e) Mã nguồn hỗ trợ đánh giá 11](#_Toc210502987)

[f) Mã nguồn agent (better\_agent) 12](#_Toc210502988)

[g) Kết quả & nhận xét 13](#_Toc210502989)

[2. Maze 13](#_Toc210502990)

[a) Tóm tắt 13](#_Toc210502991)

[b) Cài đặt và chạy 14](#_Toc210502992)

[c) Nhận xét 18](#_Toc210502993)

[3. Robott Vaccum 18](#_Toc210502994)

[a) Tóm tắt 18](#_Toc210502995)

[b) Cài đặt và chạy 19](#_Toc210502996)

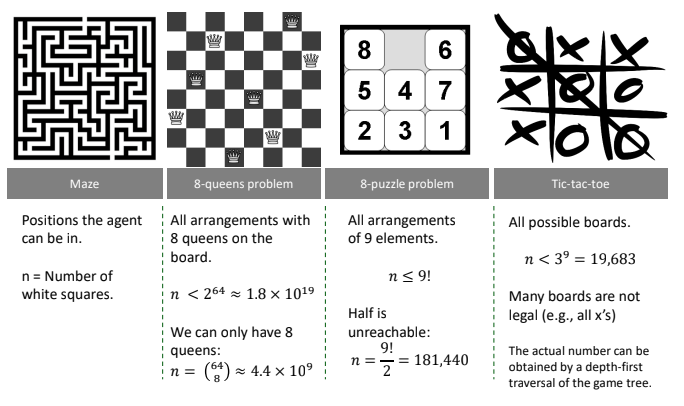
# Bảng Phân Công

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Công việc** | **Người thực hiện** | **Trạng thái** |
| 1 | Lunar Lander | Văn Nam | Hoàn thành |
| 2 | Robot Vaccum | Tuấn Nghĩa | Hoàn thành |
| 3 | Maze | Hồng Quí | Hoàn thành |
| 4 | Discussion 1, 2, 3, slide | Quốc Vương | Hoàn thành |
| **Đánh giá**: | | | |

# Phần 1: Discussion

## Discussion 1: What is the State Space Size?

Often a rough upper limit is sufficient to determine how hard the search problem is.



Khi tiếp cận một bài toán tìm kiếm, việc đầu tiên chúng ta cần quan tâm là kích thước không gian trạng thái ( state space ) vì đây là tập hợp tất cả các trạng thái có thể xảy ra của một bài toán. Việc ước lượng kích thước không gian trạng thái cho ta biết mức độ khó của bài toán: nếu quá lớn, tìm kiếm brute-force gần như không khả thi và ta buộc phải dùng heuristic hoặc cắt tỉa.

Phân tích các ví dụ cụ thể

a) Maze

* Đơn giản nhất: n = số ô trắng agent có thể đứng
* Thực tế có thể nhỏ hơn nhiều do rào cản địa hình

b) 8-Queens Problem

* Giới hạn trên: (mỗi ô có/không có hậu)
* Thực tế chỉ  = (chọn 8 ô từ 64 ô)
* Nhỏ hơn nữa nếu tính ràng buộc không tấn công nhau

c) 8-Puzzle

* Hoán vị toàn bộ ( bao gồm cả những ô trống) 9! = 362,880 trạng thái.
* Chỉ một nửa đạt được do tính chất hoán vị, quy tắc chẵn lẻ hoán vị: chỉ một nửa là reachable từ một cấu hình chuẩn: = 181,440.

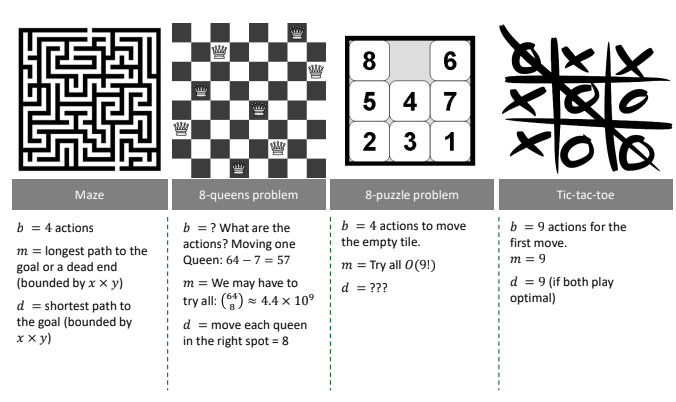
d) Tic-Tac-Toe

* Mỗi ô có 3 trạng thái (X/O/Trống): cận trên cho “mọi bàn cờ khả dĩ”.
* Nhiều bàn không hợp lệ (ví dụ chênh lệch lượt X/O vô lý) nên số trạng thái thực sự nhỏ hơn.

## Discussion 2: What is the Search Complexity?

Often a rough upper limit is sufficient to determine how hard the search problem is.

|  |
| --- |
| b: maximum branching factor  m: max. depth of tree  d: depth of the optimal solution |



Khi đã ước lượng được kích thước không gian trạng thái, bước tiếp theo là phân tích độ phức tạp tìm kiếm (search complexity) để hiểu xem thuật toán của chúng ta sẽ “tốn kém” bao nhiêu thời gian và bộ nhớ. Ba tham số cơ bản cần xem xét là:

* b: hệ số phân nhánh tối đa (branching factor) – số hành động khả dĩ từ mỗi trạng thái.
* m: độ sâu tối đa của cây tìm kiếm (max depth) – chiều dài dài nhất của một nhánh, có thể dẫn tới một dead end hoặc goal.
* d: độ sâu của lời giải tối ưu (optimal depth) – số bước ngắn nhất để đến trạng thái mục tiêu.

Độ phức tạp điển hình

- BFS (Breadth-First Search)

* Thời gian:
* Bộ nhớ:
* Ưu điểm: đảm bảo tìm lời giải có độ sâu nhỏ nhất.
* Nhược điểm: bộ nhớ bùng nổ khi b hoặc d lớn.

- DFS (Depth-First Search)

* Thời gian:
* Bộ nhớ:
* Ưu điểm: dùng ít bộ nhớ (stack đường đi).
* Nhược điểm: không đảm bảo tối ưu và có thể lạc sâu vào dead end.

- Iterative Deepening (IDS)

* Thời gian: (tương tự BFS nhưng lặp lại các lần cắt tăng dần).
* Bộ nhớ: (như DFS).
* Kết hợp ưu điểm tìm ra lời giải ngắn nhất và tiết kiệm bộ nhớ.

Kết luận: Nếu b hoặc d quá lớn, BFS “nổ” bộ nhớ; DFS thì tiết kiệm nhớ nhưng dễ lạc; IDS dung hòa cả hai.

Áp dụng trên 4 ví dụ để ước lượng b, m, d

a) Maze (lưới 4 hướng)

* b = 4 (lên/xuống/trái/phải)
* m: đường dài nhất – bị giới hạn bởi số ô .
* d: đường ngắn nhất đến goal – cũng không vượt .

b) 8-Queens

* Hành động: “di chuyển 1 hậu đến ô khác”
* b: mỗi hậu có thể chuyển tới 64 -7 = 57 ô (trừ ô đang chiếm và 7 ô cùng hàng/cột/chéo).
* m: nếu thử tất cả cách đặt 8 hậu, không gian xấp xỉ .
* d = 8: đưa mỗi hậu vào vị trí đúng.

c) 8-Puzzle

* b = 4(di chuyển ô trống).
* m: cận trên 9!(thử mọi hoán vị).
* d: phụ thuộc cấu hình ban đầu (thường khác nhau theo từng bài).

d) Tic-Tac-Toe

* b = 9 (9 lựa chọn đầu, sau đó giảm)
* m = 9 (tối đa 9 lượt).
* nếu cả hai chơi tối ưu (luân phiên đến hết).

Cách lựa chọn thuật toán phù hợp

* Ước lượng nhanh b, d, m dựa trên cách mô hình hóa trạng thái và hành động.
* Nếu trong tầm xử lý → BFS hoặc IDS khả thi để tìm giải tối ưu.
* Nếu b hoặc d quá lớn → cần dùng heuristic (A\*), áp dụng pruning hoặc đổi biểu diễn để giảm b và m.

Kết luận: Phân tích này giúp định hướng chính xác: với bài toán nhỏ (Tic-Tac-Toe, 8-Puzzle), các thuật toán cơ bản đủ dùng, còn bài toán lớn như 8-Queens, cờ vua,... cần các kỹ thuật thông minh hơn để tránh “bùng nổ tổ hợp.”

## Discussion 3: Case Study: Heuristic for Tic-Tac-Toe

* Define the goal states:
* What is the cost that needs to be estimated?
* What would be a heuristic value for these boards:



* How do you calculate the heuristic value?
* Is the heuristic admissible?
* Does the heuristic use a relaxation?

**Trạng thái mục tiêu (Goal states)**

Một goal state xảy ra khi X hoặc O, nếu một người chơi thắng nếu có 3 quân liên tiếp trên hàng, cột hoặc đường chéo. Còn bàn cờ đầy mà không ai thắng → hòa.

**Chi phí cần ước lượng (cost to be estimated)**

Ta ước lượng số lượt đi tối thiểu của người chơi P (X hoặc O) có thể đạt để thắng từ trạng thái hiện tại. Đây là “cost-to-go” cho P. Ví dụ: nếu X đã có 2 quân trong một hàng và còn 1 ô trống, thì chi phí để thắng là 1 lượt.

**Heuristic đề xuất (admissible)** giống như một cách ước lượng nhanh xem vị trí hiện tại lợi cho ai.

* Xem xét từng hàng, từng cột, từng đườn chéo với tổng là 8 line (3 hàng + 3 cột + 2 chéo).
* Nếu một line có cả X và O → không ai có thể thắng bằng line đó nữa.
* Nếu line chỉ có X và ô trống → line đó “mở” cho X.
* Nếu line chỉ có O và ô trống → line đó “mở” cho O.

Cách tính Heuristic đơn giản

H (n) = ( số line còn mở cho X ) - ( số line còn mở cho O )

Kết quả của h: Nếu h > 0 → X lợi thế; h < 0 → O lợi thế; h = 0 → cân bằng.

**Đối với Board bên trái được diễn giải như sau:**

* X có hàng 1 (X,X,\_) → chỉ cần 1 nước nữa thắng.
* O có đường chéo phụ (\_ , O , O) → cũng chỉ cần 1 nước thắng.
* Kết quả: cả hai đều có cơ hội thắng nhanh → thế nguy hiểm, tùy ai đi trước.

**Đối với Board bên trái được diễn giải như sau:**

* X còn line cột 1 và đường chéo chính → cần ít nhất 2 lượt để thắng.
* O còn line cột 2 → cũng cần ít nhất 2 lượt để thắng.
* Kết quả: thế cân bằng.

**Cách tính heuristic (tóm tắt)**

* Xem xét 8 line (3 hàng + 3 cột + 2 chéo).
* Loại line có cả X và O (bỏ line chứa quân đối thủ)
* Với line còn lại, đếm số ô trống.
* Số ô trống ít nhất chính là số lượt nhanh nhất để thắng (heuristic).

**Heuristic có “admissible” không?**

Trả lời:  
Có. Vì nó không bao giờ đánh giá thấp hơn số lượt cần để thắng thật sự. Nó chỉ đưa ra ước lượng “ít nhất cũng phải cần bấy nhiêu lượt”.

**Heuristic có dùng “relaxation” không?**

Trả lời:

Có. Bởi vì nó giả định rằng một người chơi có thể liên tục đi các ô trống trong line mà không bị đối thủ chặn. Cũng nhờ dùng cách nới lỏng bài toán nên tính toán sẽ nhanh hơn.

# Phần 2: Lunar lander, Robot vaccum và Maze

## Lunar lander

### Tóm tắt

Xây dựng một agent phản xạ (không dùng RL) để điều khiển tàu hạ cánh an toàn trong môi trường LunarLander‑v3. Agent chọn hành động dựa trên các ngưỡng (tol/hi/lo) với thứ tự ưu tiên: (1) an toàn rơi dọc (VY→MAIN), (2) giữ thăng bằng (ANGLE/AV), (3) giảm trôi ngang (VX/X), (4) landing phase (Y thấp). Kết quả: với seed cố định (n=500, seed0=0), Average reward ≈ 28.8 và Success ≈ 313/500.

### Môi trường LunarLander‑v3

* Hiệu năng (Performance measure): phần thưởng cuối tập là +100 nếu hạ cánh an toàn, -100 nếu nổ.
* Hành động (Action space, rời rạc): 0=NO\_OP, 1=LEFT, 2=MAIN, 3=RIGHT.
* Cảm biến (Observation, 8 chiều): X, Y, VX, VY, ANGLE, ANGULAR\_VELOCITY, LEFT\_LEG\_CONTACT, RIGHT\_LEG\_CONTACT.
* Tài liệu: <https://gymnasium.farama.org/environments/box2d/lunar_lander/>

### Cài đặt & chạy

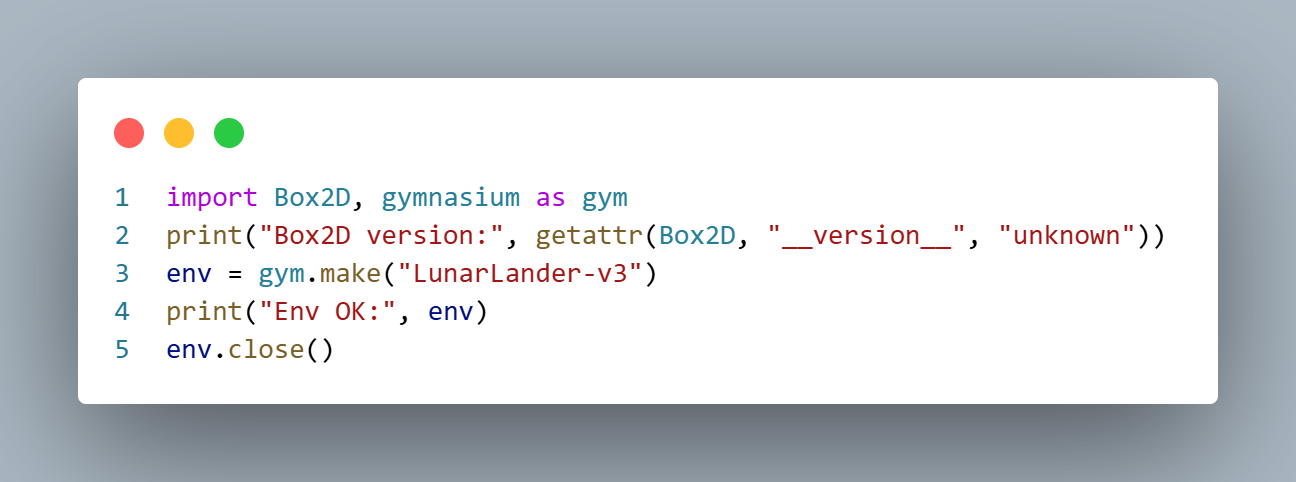
Cài dependencies:

* pip install -U pip setuptools wheel
* pip install "gymnasium[box2d]" pygame
* pip install "box2d==2.3.10" "gymnasium==1.2.1" "pygame==2.6.1"

Cách chạy nhanh:

1) Tạo venv & cài dependencies như trên.

2) Kiểm tra môi trường:



3) Chạy notebook/script chứa hàm better\_agent và hàm run\_episodes đã cố định seed.

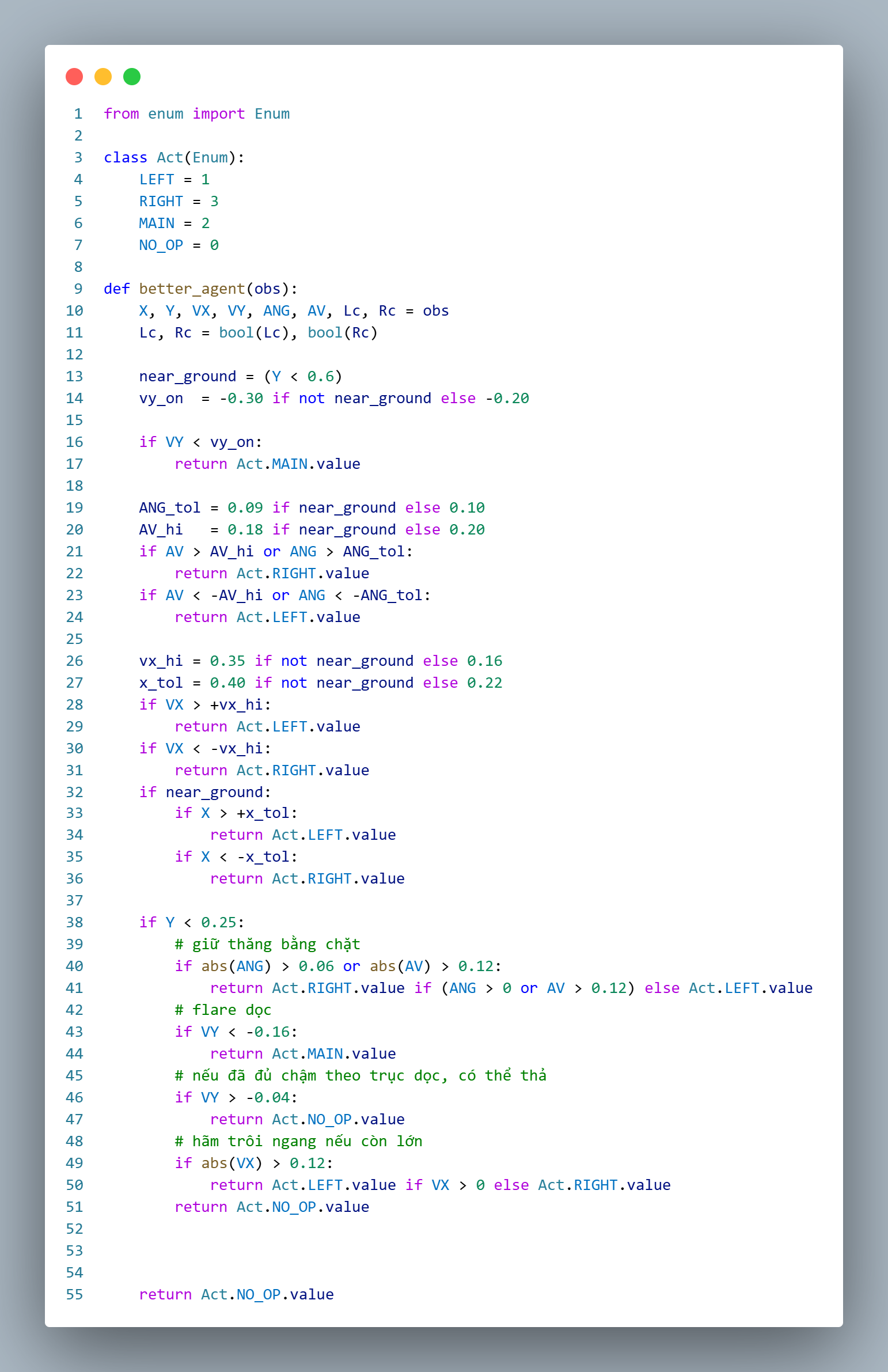
4) Lệnh test: rewards = run\_episodes(better\_agent, n=500, seed0=0)

### Bộ luật điều khiển (reflex rules)

* Ưu tiên 1 – Vertical safety: bật MAIN khi VY < vy\_on; không return sớm khi VY đã an toàn để còn xử lý góc/ngang.
* Ưu tiên 2 – Giữ thăng bằng: dùng ANG\_tol, AV\_hi/AV\_lo (deadband) để đưa ANG/AV về 0, tránh nhấp nháy.
* Ưu tiên 3 – Điều khiển ngang: hãm VX khi vượt vx\_hi; khi gần đất, nắn X về pad nếu |X| > x\_tol.
* Ưu tiên 4 – Landing phase: siết thăng bằng; flare nếu VY < -0.16; nếu VY > -0.04 thì thả; nếu |VX| lớn thì hãm thêm.

### Mã nguồn hỗ trợ đánh giá



f) Mã nguồn agent (better\_agent) 

### Kết quả & nhận xét

* Thiết lập đánh giá: seed0=0, n=500.
* Kết quả cuối: Average reward ≈ 28.8 ; Success ≈ 313/500.
* Nhận xét: so với các bước sớm (Step 1, Step 2), agent đã tăng rõ rệt nhờ: hysteresis cho VY, deadband cho ANG/AV, và siết VX/X khi gần đất; đồng thời giữ thứ tự ưu tiên hợp lý trong landing phase.

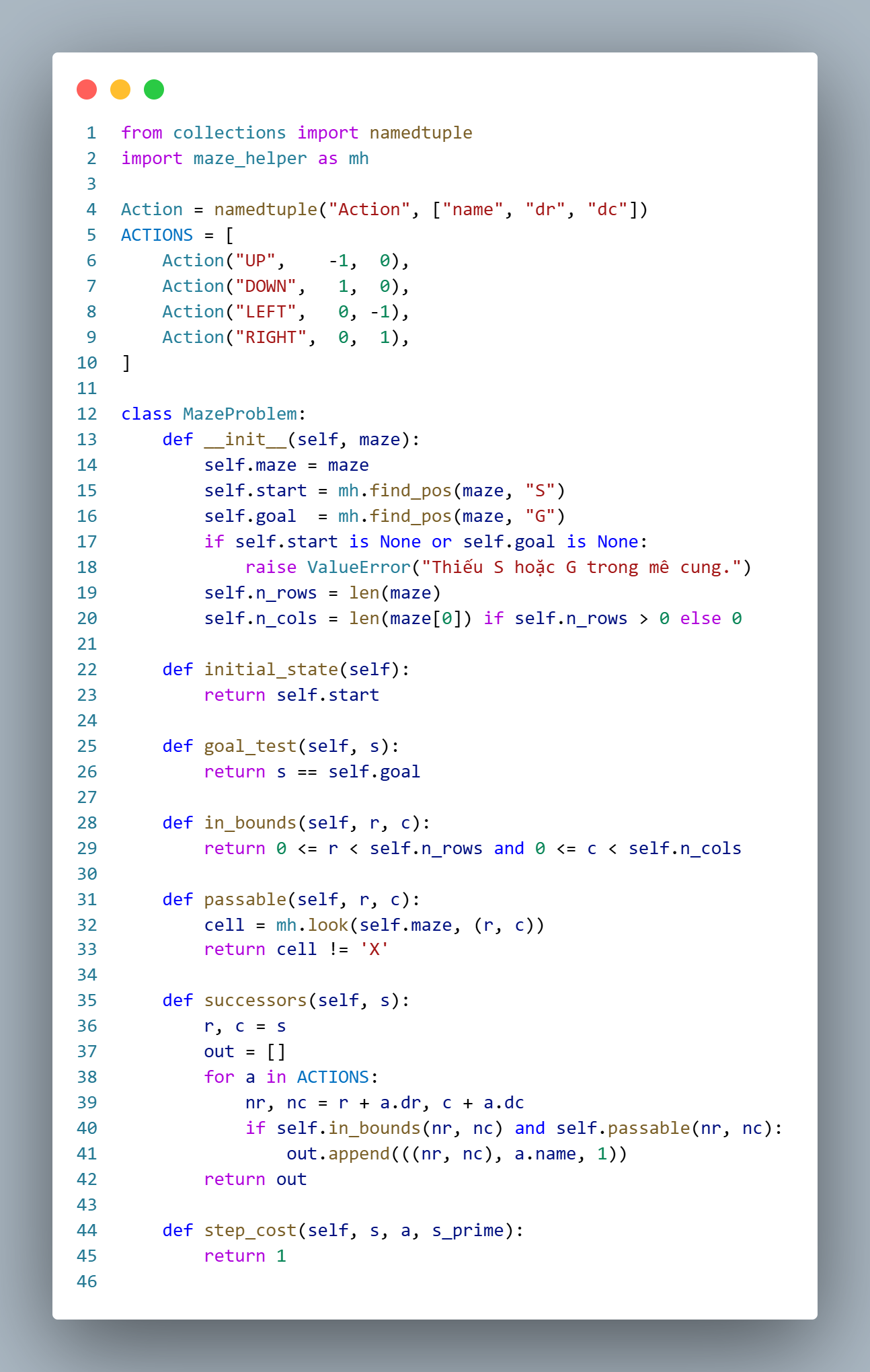
## Maze

1. Tóm tắt

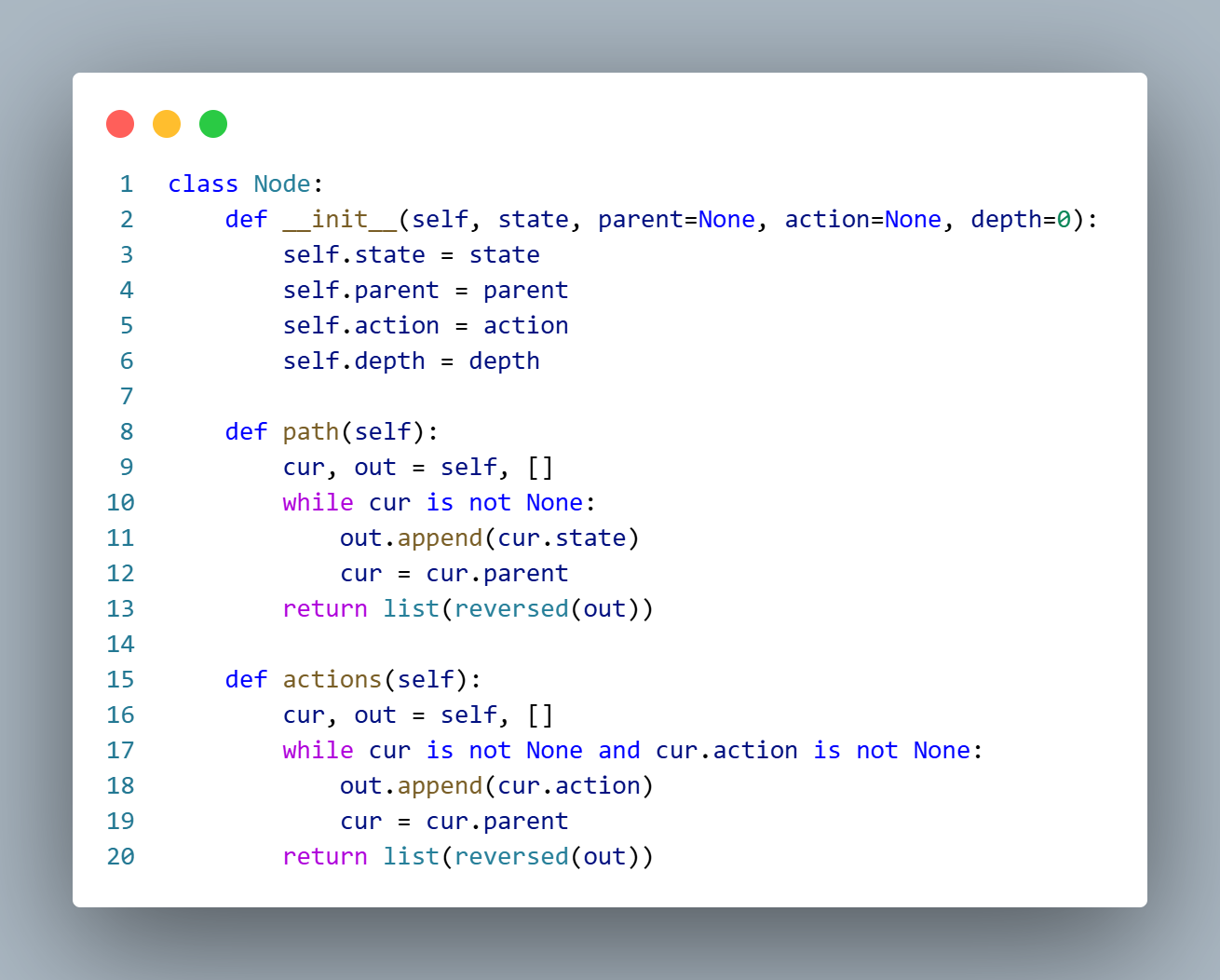
* Mô hình hoá mê cung thành bài toán tìm kiếm với state là toạ độ (r, c), hành động UP/DOWN/LEFT/RIGHT, chi phí mỗi bước = 1.
* Tự cài đặt và chạy: BFS, DFS (có kiểm tra chu kỳ), Greedy Best-First (GBFS), A\* (heuristic Manhattan).
* Ghi log & so sánh: path\_len, nodes\_expanded, solution\_depth, max\_frontier.
* Trình bày đường đi (in path) để kiểm chứng.
* (Tuỳ chọn/bonus) IDS & multi-goal.

### Cài đặt và chạy

Khởi tạo bài toán MazeProblem



Cấu trúc Node



Thuật toán BFS



Thuật toán DFS



### Nhận xét

* **BFS, A\*** tìm đường tối ưu; **GBFS** không bảo đảm tối ưu nhưng mở rộng ít node khi heuristic tốt.
* **DFS** tốn ít bộ nhớ nhất, nhưng dễ kẹt hoặc không tối ưu nếu không kiểm tra chu kỳ.
* **A\*** thường hiệu quả nhất trong các mê cung lớn khi dùng Manhattan.

## Robott Vaccum

### Tóm tắt

Mục tiêu bài tập

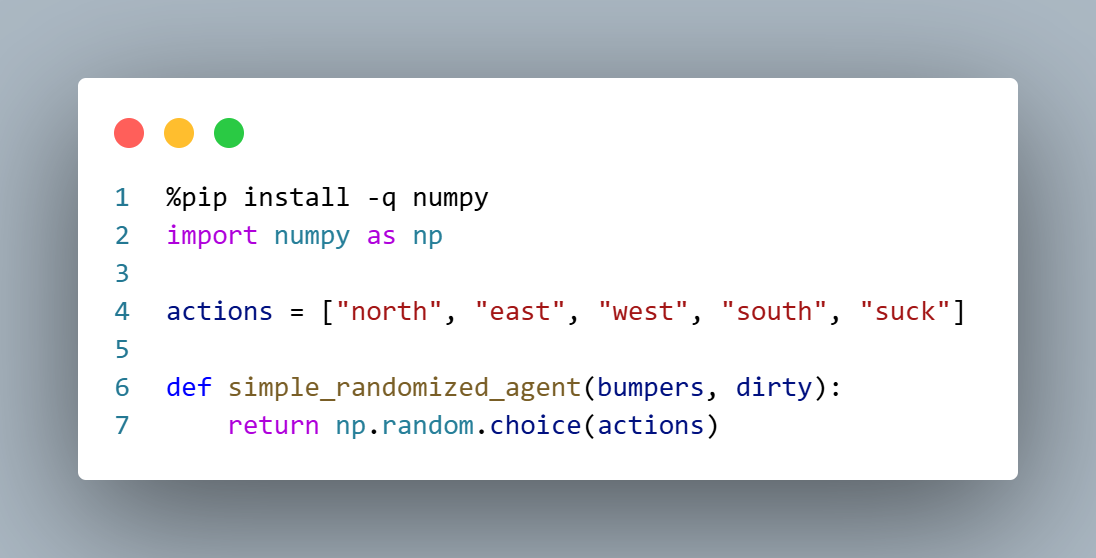
* Thiết kế môi trường mô phỏng cho robot hút bụi theo mô tả PEAS (n×n, dirt xác suất p, cảm biến bumper & bẩn, chi phí mỗi hành động = 1).
* Cài đặt và thử nghiệm các tác tử phản xạ (reflex-based):
  1. Randomized agent (chọn hành động ngẫu nhiên);
  2. Simple reflex agent (không đâm tường, gặp bẩn thì hút);
  3. Model-based reflex agent (có trạng thái/chiến lược quét toàn phòng);
  4. Agent\_advance (biến thể xử lý cảm biến bẩn không hoàn hảo bằng bỏ phiếu).
* Đo hiệu năng: số bước/ năng lượng dùng để làm sạch (hoặc % sạch trong giới hạn max\_steps) trên nhiều kích thước phòng (5×5, 10×10, 100×100) và nhiều lần chạy ngẫu nhiên.

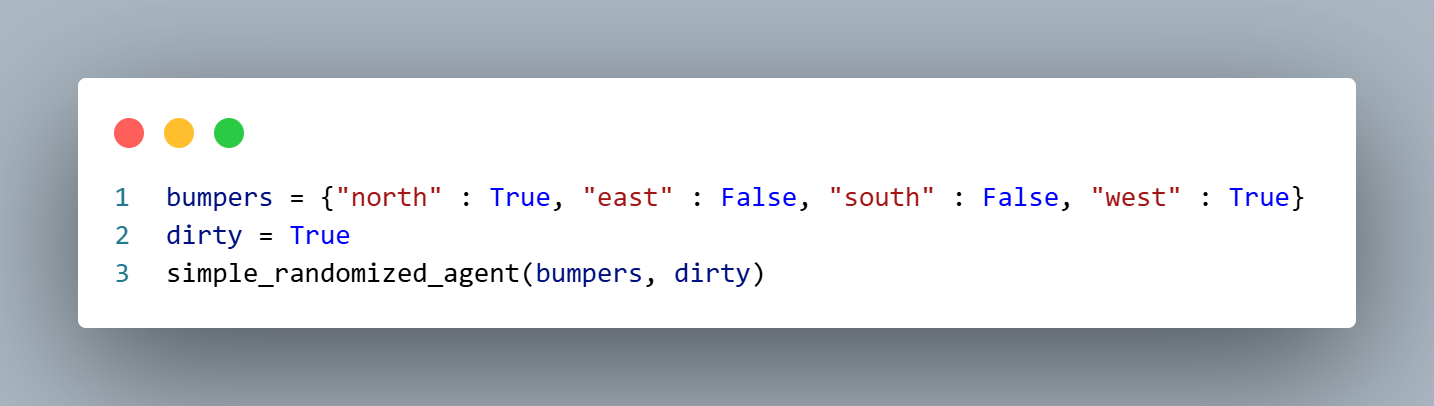
Ý tưởng chính của code

* Phần đầu notebook giới thiệu Learning Outcomes, Instructions, hướng dẫn dùng Colab/VS Code, và ví dụ simple\_environment (môi trường vô hạn để minh hoạ).
* Phần chính xây dựng môi trường đúng PEAS: setup\_room() khởi tạo bẩn/sạch; real\_environment() chạy vòng lặp cảm biến → tác vụ → cập nhật trạng thái & đếm hiệu năng đến khi sạch/ hết năng lượng.
* Tác tử:
  + simple\_randomized\_agent(bumpers, dirty): chọn ngẫu nhiên trong ["north","east","west","south","suck"].
  + simple\_reflex\_agent(bumpers, dirty): nếu bẩn thì “suck”, ngược lại chọn ngẫu nhiên trong các hướng không bị bumper chặn.
  + Agent: model-based, đưa robot về trạm (0,0) rồi quét “răng lược” (serpentine) toàn phòng, mỗi bước gọi “suck” nếu ô đang bẩn.
  + Agent\_advance: tương tự Agent nhưng thêm bỏ phiếu cảm biến (majority vote) để giảm sai lệch 10% từ cảm biến bẩn.
* So sánh hiệu năng:
  + Compare\_the\_performance\_1(...): so sánh 3 agent (random, simple reflex, model-based).
  + Compare\_the\_performance(...): so sánh 4 agent (thêm Agent\_advance).
  + In kết quả trung bình (steps, % sạch) theo nhiều kích thước/ số lần lặp.

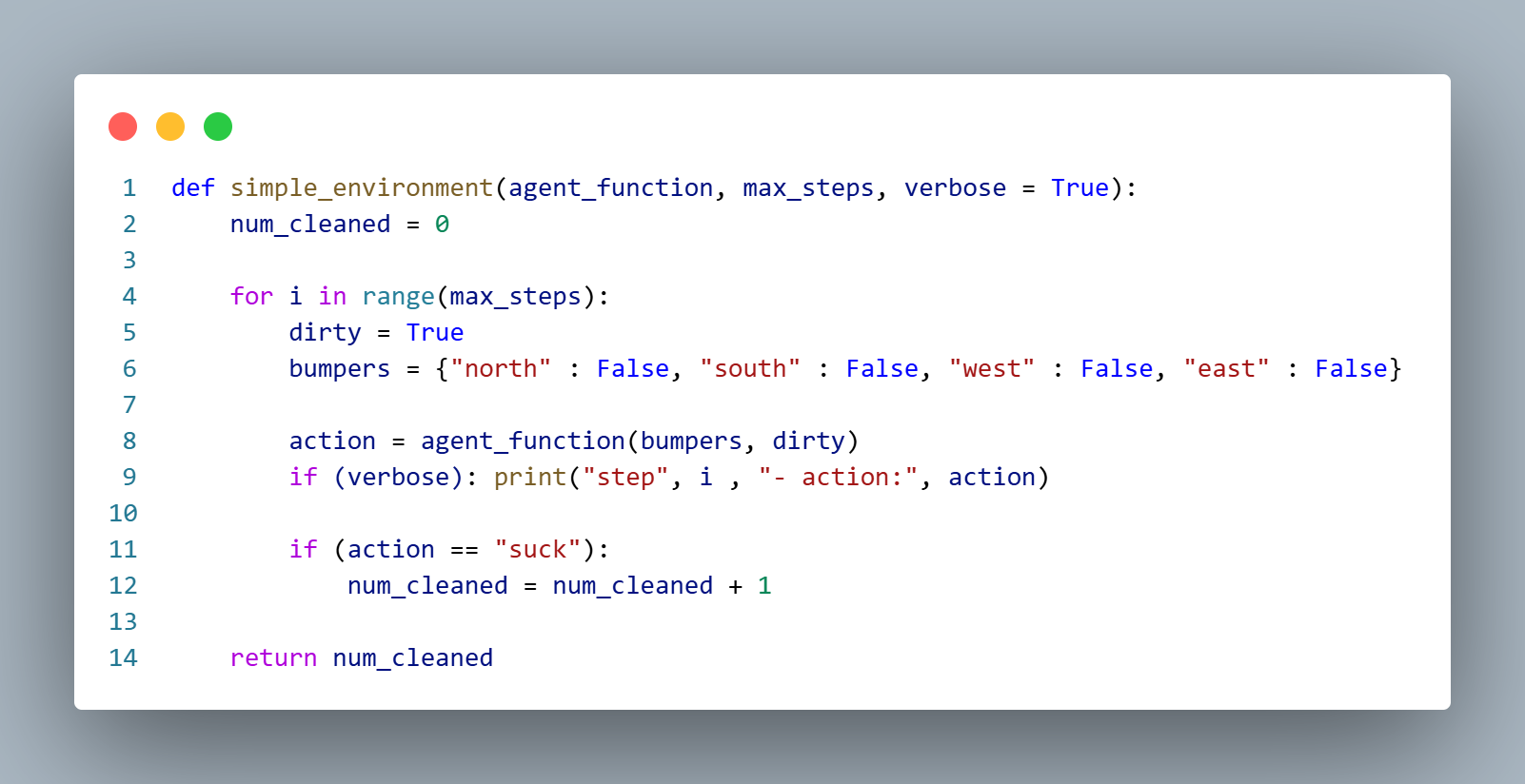
## Cài đặt và chạy

Cài đặt thư viện & agent ngẫu nhiên

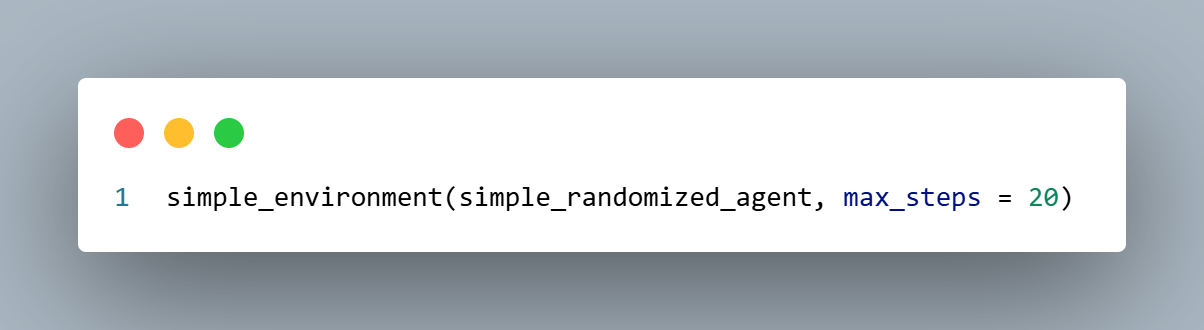


Gọi thử agent với percepts mẫu: 

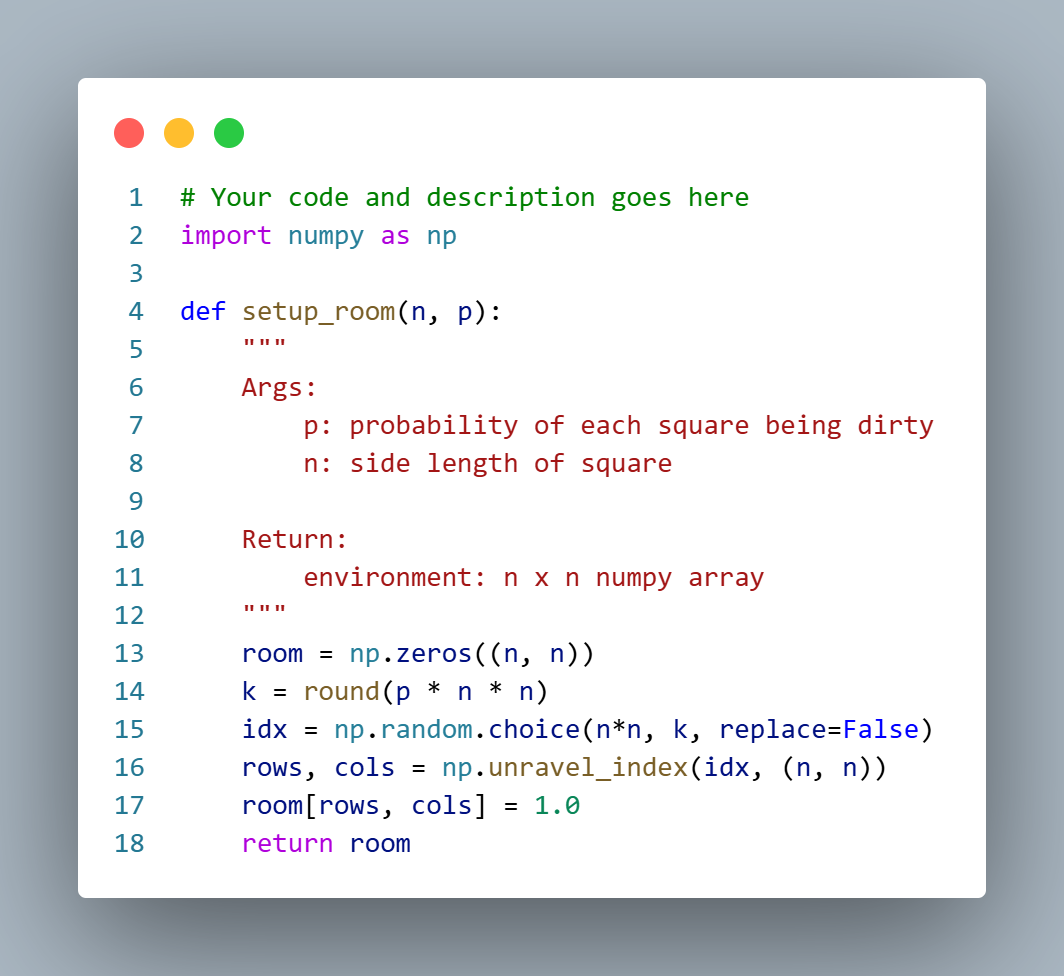
Ví dụ môi trường đơn giản

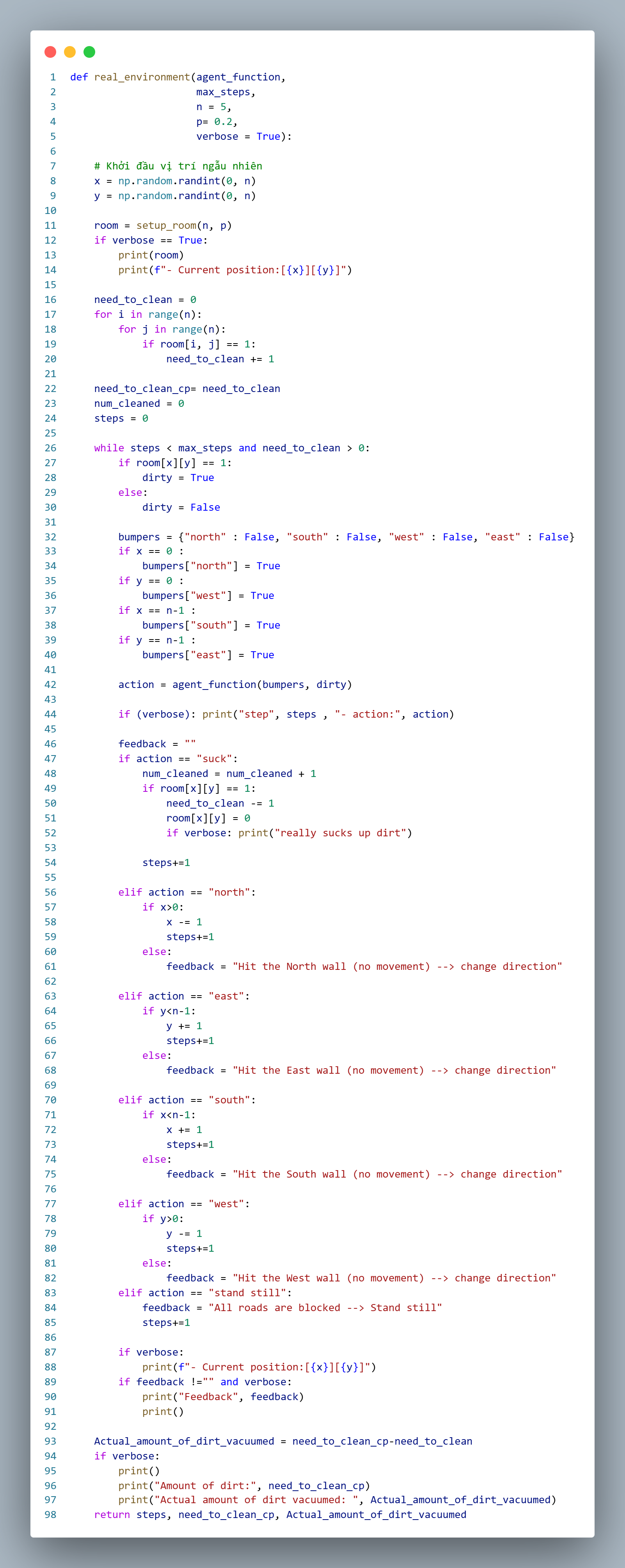


Chạy thử:

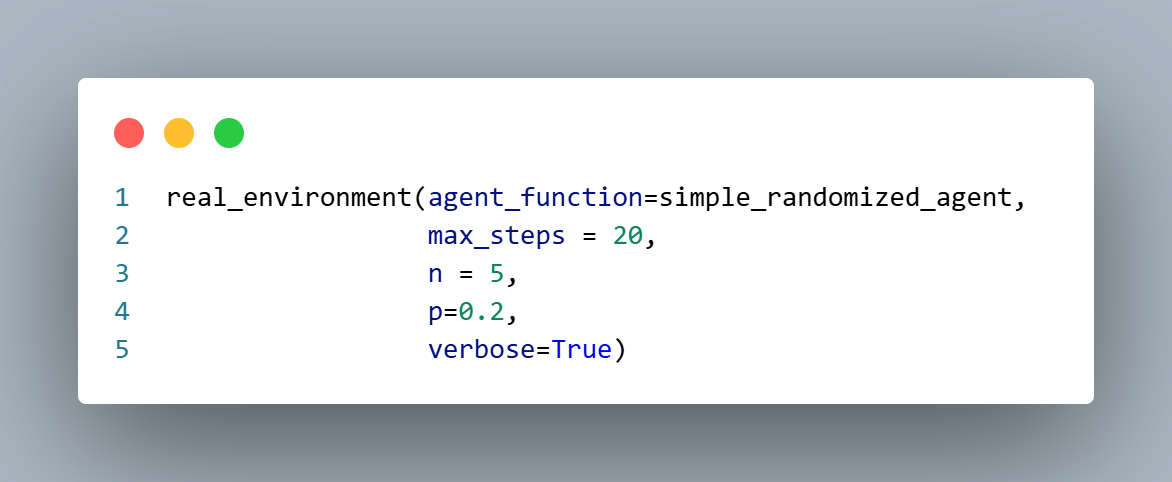


Môi trường PEAS thật sự: khởi tạo & vòng lặp mô phỏng

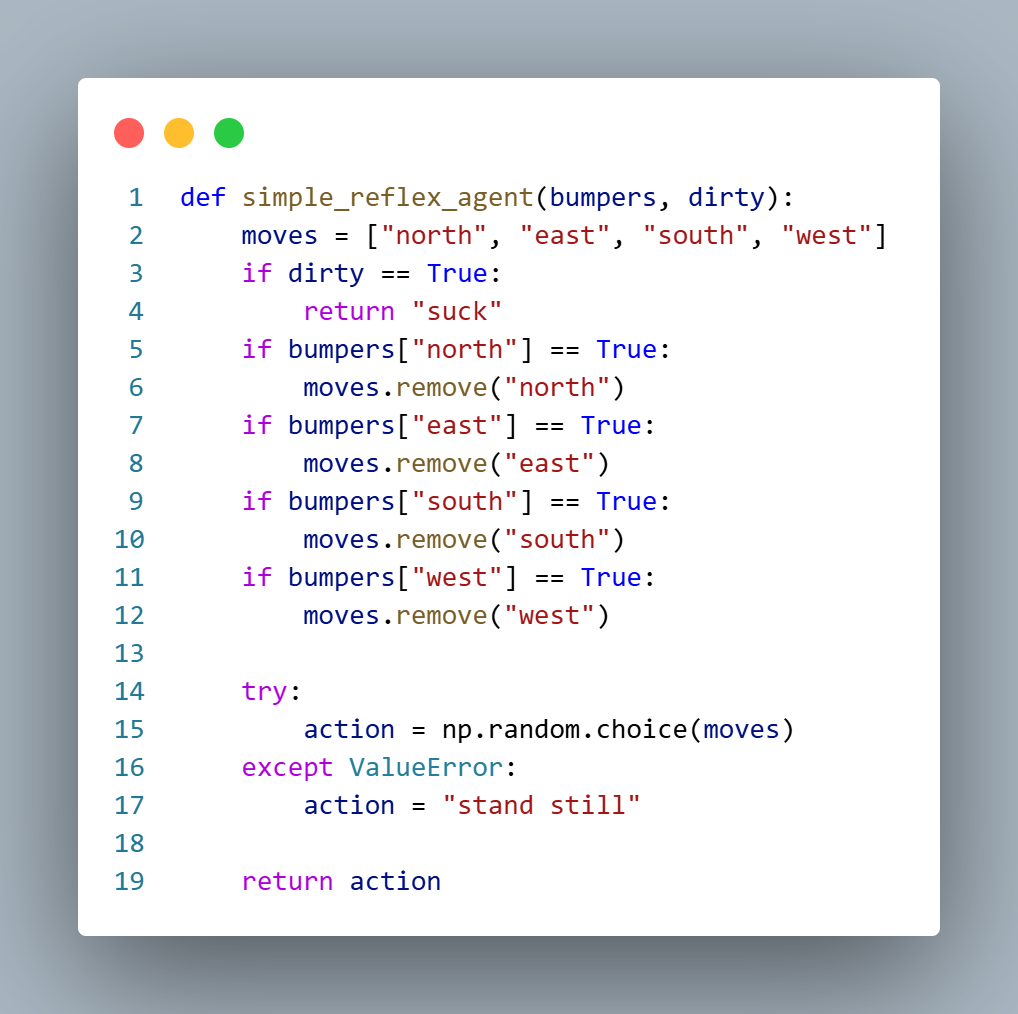




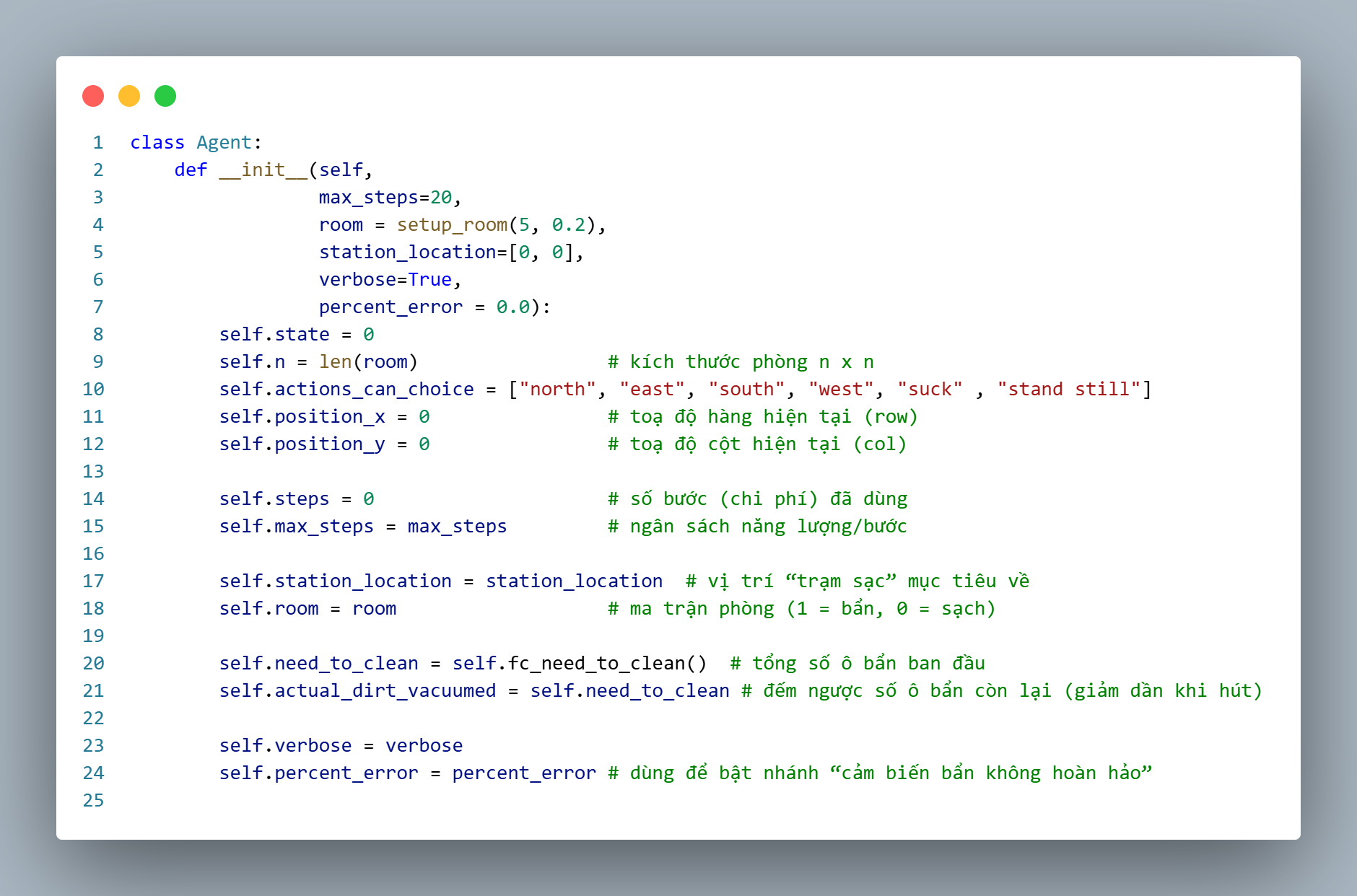
Chạy môi trường với agent ngẫu nhiên:



Simple Reflex Agent

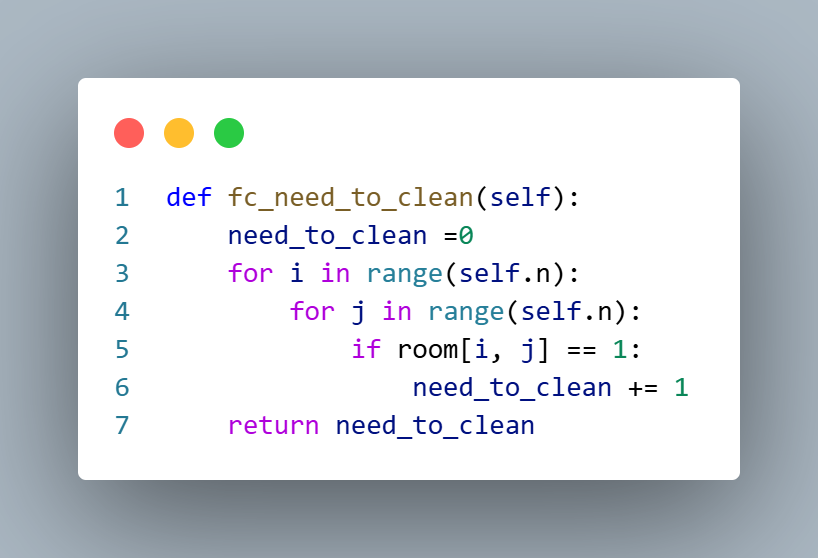


Model-based Reflex Agent (lớp Agent)



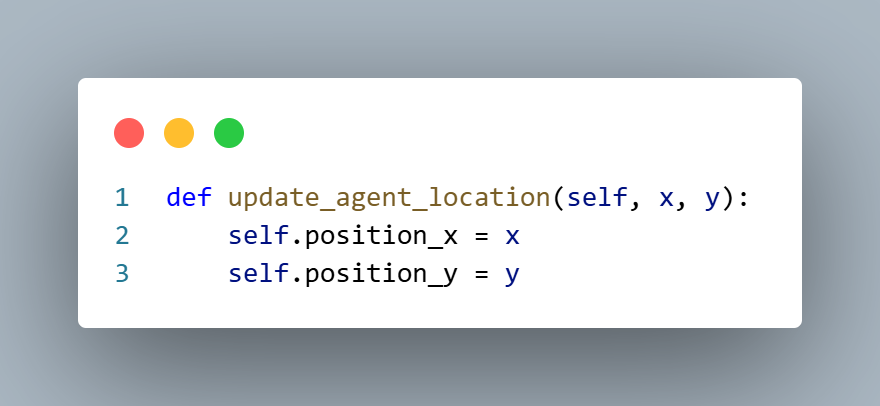
Ý tưởng: agent nắm kích thước phòng, vị trí hiện tại, số bước đã dùng, ma trận bẩn/sạch. Sẽ đi theo một chiến lược quét toàn phòng (serpentine).

fc\_need\_to\_clean(self)

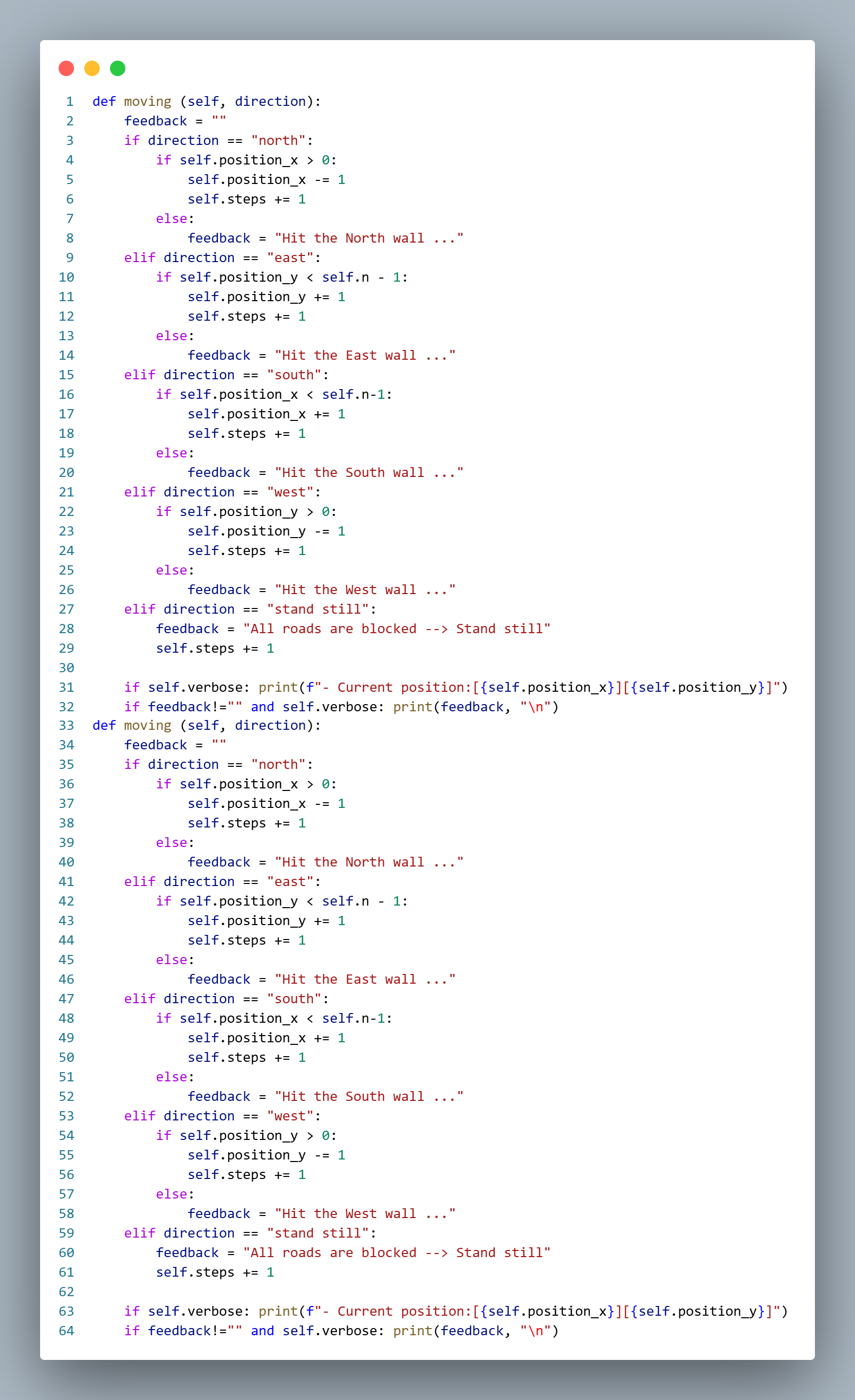


* Đếm tổng số ô bẩn ban đầu.
* Lưu ý: hàm đang đọc room toàn cục (không phải self.room). Trong notebook của bạn có biến toàn cục room = setup\_room(...), nên hàm chạy được. Nhưng về logic, dễ nhầm lẫn vì đối tượng cũng có self.room.

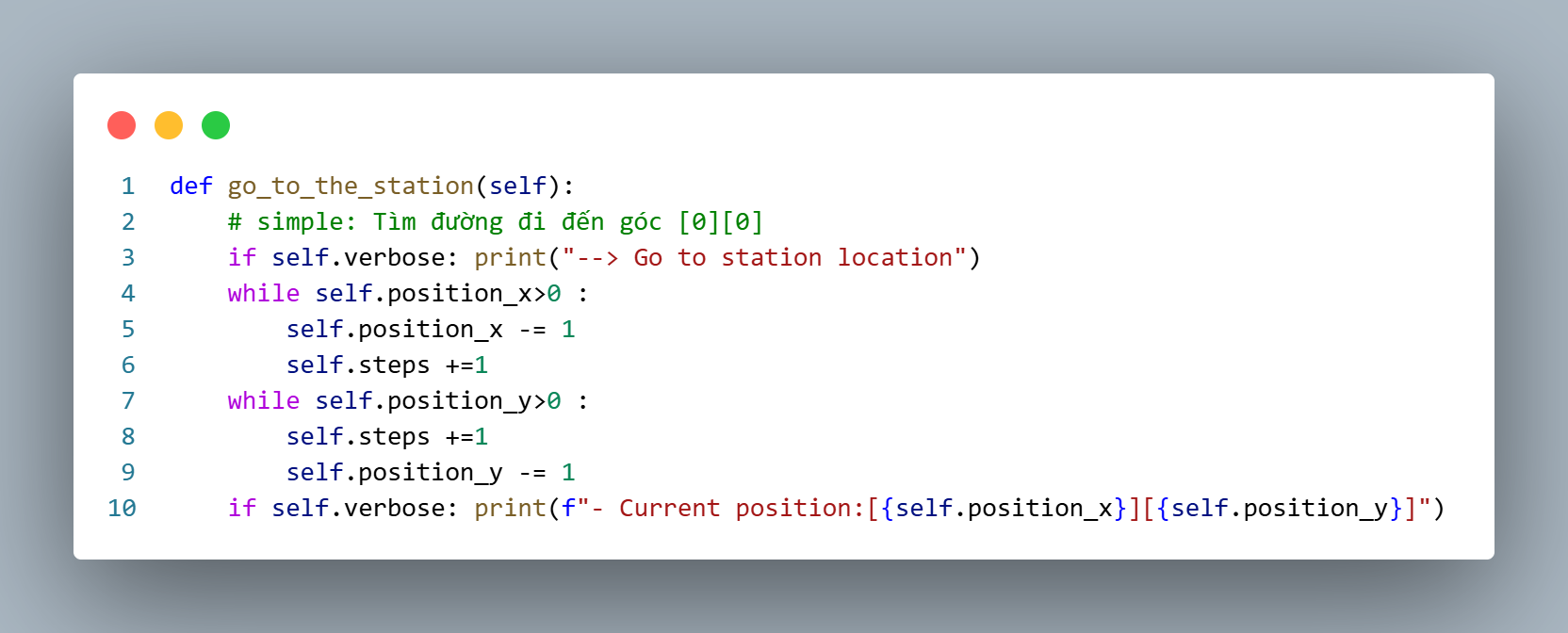
Cập nhật vị trí agent trước khi bắt đầu dọn



moving(self, direction)

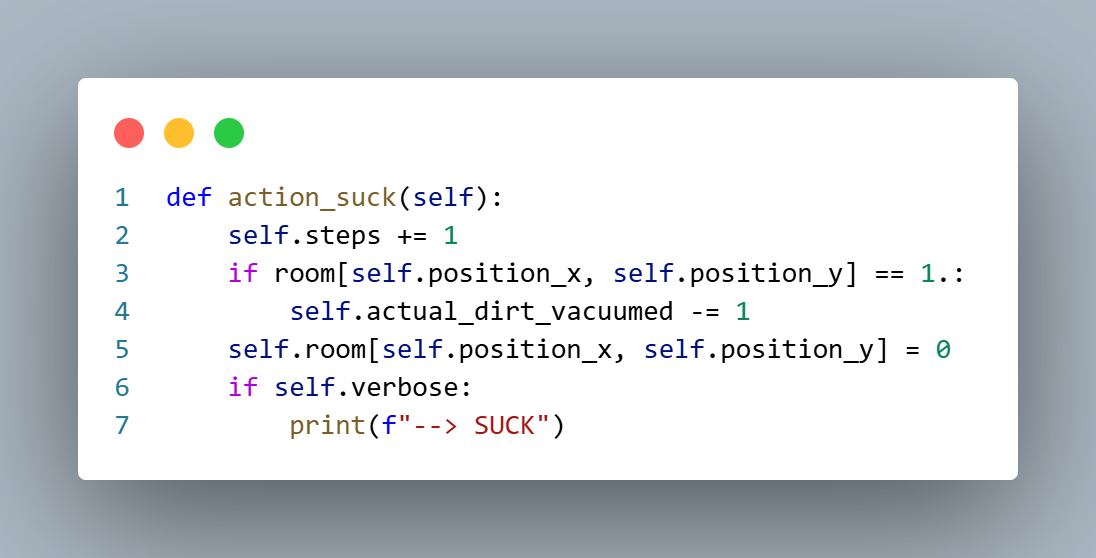


go\_to\_the\_station(self)



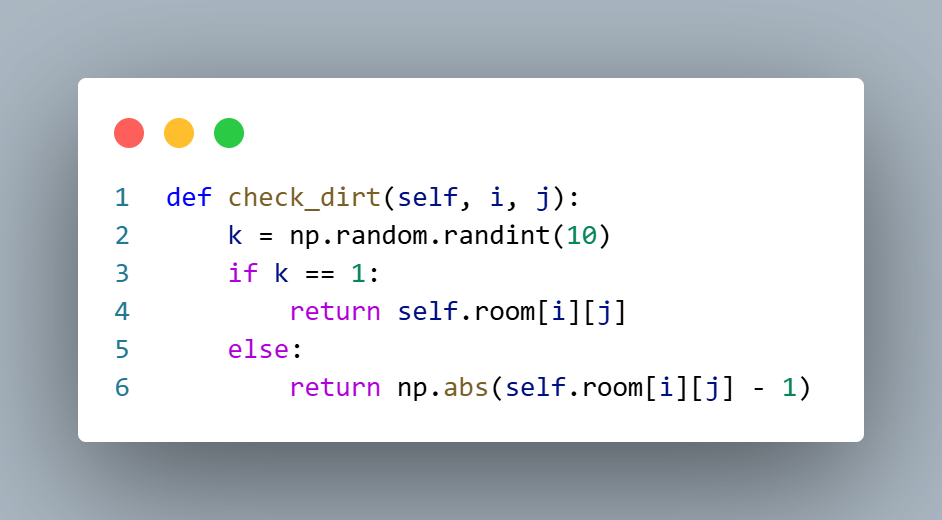
* Đưa agent trở về góc (0,0) bằng cách lần lượt đi lên đến hàng 0 rồi đi trái đến cột 0.
* Mỗi bước di chuyển đều tính chi phí.

action\_suck(self)



* Thực hiện hành động hút bụi tại ô hiện tại:
* Tăng chi phí (steps += 1).
* Nếu ô hiện tại (theo room toàn cục) đang bẩn (==1), giảm bộ đếm actual\_dirt\_vacuumed đi 1.
* Đặt ô tương ứng trong self.room về 0 (sạch).
* Lưu ý: ở đây có hai nguồn dữ liệu: đọc bẩn/sạch từ room toàn cục, nhưng ghi vào self.room. Điều này vẫn chạy trong notebook của bạn vì cả hai ma trận thực tế trùng nội dung lúc khởi tạo, nhưng về mặt nhất quán, dễ gây nhầm.

check\_dirt(self, i, j)



* Mục đích: mô phỏng cảm biến bẩn không hoàn hảo.
* Cách hoạt động hiện tại:
* Với xác suất 1/10 (k==1), trả về giá trị thật tại ô (0 = sạch, 1 = bẩn).
* Với xác suất 9/10, trả về ngược lại (|x-1| đảo 0↔1).
* Nghĩa là 90% sai lệch, 10% đúng (rất nhiễu). Trong comment ở notebook bạn đặt percent\_error, nhưng hàm này không dùng biến đó; nó luôn cho kịch bản 90% sai.

whole\_house\_vacuuming(self)



* Luồng chính:
* Nếu chưa ở (0,0), gọi go\_to\_the\_station() để về góc.
* Bắt đầu quét phòng theo kiểu serpentine (răng lược):
  + Vòng i theo hàng: mỗi hàng đi ngang hết một dải,
    - Nếu i chẵn → đi “east” liên tiếp,
    - Nếu i lẻ → đi “west” liên tiếp.
  + Cuối mỗi hàng (trừ hàng cuối) → đi “south” xuống hàng tiếp theo.
* Tại mỗi ô, nếu bật chế độ lỗi cảm biến (percent\_error != 0.0) thì gọi check\_dirt; nếu kết quả “bẩn” → action\_suck(). Nếu không bật lỗi, kiểm tra trực tiếp self.room[x][y].
* Nếu vượt quá max\_steps thì dừng sớm.
* Lưu ý:
* Vòng lặp dùng range(self.n-1) cho cả hàng và cột → bạn di chuyển n-1 lần mỗi chiều; việc di chuyển và số cột/biên có thể để lại ô cuối tuỳ vị trí khởi đầu (vì di chuyển sau khi xử lý ô). Đây là đặc trưng chiến lược hiện tại chứ không phải bug; chỉ cần hiểu là lộ trình đi quét hàng dựa vào i%2.
* Khi percent\_error != 0.0, check\_dirt như đã nói đang 90% đảo kết quả, 10% đúng — nghĩa là agent sẽ thường hút khi ô sạch và bỏ qua khi ô bẩn. (Bạn đã có bản nâng cao Agent\_advance để xử lý bằng majority vote.)