





Snekoban

You are not logged in.

Please [Log In](#) for full access to the web site.

Note that this link will take you to an external site (<https://shimmer.mit.edu>) to authenticate, and then you will be redirected back to this page.

Table of Contents

- [1\) Preparation](#)
- [2\) Introduction](#)
- [3\) Game Rules](#)
 - [3.1\) Board](#)
 - [3.2\) Objects](#)
 - [3.2.1\) The Player](#) 
 - [3.2.2\) Walls](#) 
 - [3.2.3\) Computers](#) 
 - [3.2.4\) Targets](#) 
 - [3.3\) Check Yourself](#)
 - [3.4\) Interface](#)
 - [3.5\) GUI](#)
 - [3.5.1\) Testing and Debugging](#)
 - [3.6\) Implementation](#)
 - [3.7\) Hint: Breaking The Problem Down](#)
- [4\) Solver](#)
- [5\) Code Submission](#)
- [6\) Checkoff](#)
- [7\) Optional: Speeding Up The Solver](#)

1) Preparation

This lab assumes you have Python 3.11 or later installed on your machine (3.12 recommended).

The following file contains code and other resources as a starting point for this lab: [snekoban.zip](#)

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- passing the tests in `test.py` (4 points), and
- a brief "checkoff" conversation with a staff member about your code (1 point).

Note in order to receive credit for the lab, you will need to come (in person) to any open lab time after the lab deadline has passed and add yourself to the queue asking for a checkoff.

It is a good idea to submit your lab early and often so that if something goes wrong near the deadline, we have a record of the work you completed before then. We recommend submitting your lab to the server after finishing each substantial portion of the lab, which will also display the results of our style checks.

Reminder: Academic Integrity

Please also review the [academic integrity policies](#) before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources.**

2) Introduction

In this lab, we'll implement a version of a popular game called [Sokoban](#). The original game involves moving a person around a virtual warehouse floor, pushing boxes onto target locations, until each target is covered with a box. In our version ("Snekoban"), rather than a warehouse worker, the player controls a little python (🐍), and the goal is to push computers (💻) around a world surrounded by walls (🧱) until every target (🚩) is covered with a computer.

We'll start by implementing the rules of the game (which are described in much more detail below), and then we'll write an additional program that *solves* Sokoban puzzles, producing as output a sequence of moves that takes us from a starting configuration to one that solves the puzzle.

Reference Implementation

We have also made a [reference implementation](https://hz.mit.edu/snekoban/) of our version of the game available at <https://hz.mit.edu/snekoban/>, in case you want to try playing it before implementing it for yourself!

The reference implementation can be a useful tool to help with understanding the rules and for debugging (comparing against your own implementation); you should feel free to make use of it throughout this lab, and you should also feel free to ask us questions if any behavior doesn't make sense!

3) Game Rules

This section talks through the various components that make up the game, as well as the rules of the game. It may be worth reading through all of section 3 before implementing anything.

In this lab, you are free to come up with your own representations for each of these components. You may use any Python data structures (dictionary, list, tuple, set, frozenset, etc.). Anything is possible (if it works!), so it is important to read through the lab and design your game representation that would fit best with your implementation. In particular, it is worth thinking about: based on the different functions we want

to implement, what *questions* am I asking about the data frequently? It will be important to make sure that those questions can be answered efficiently by your choice of data structure.

Meanwhile, the server and our test cases will use a **canonical** representation of the game. Eventually, you will have to write a function that converts to and from this canonical representation and your internal representation of the game. See [subsection 3.6](#) for more information.

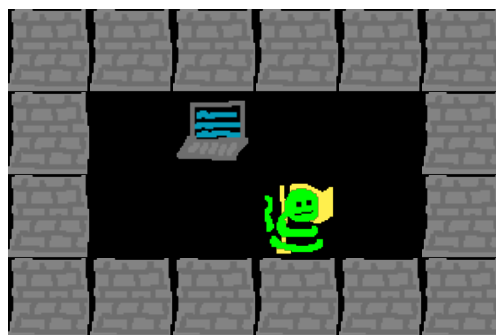
3.1) Board

The game board is an $m \times n$ grid, where m and n are the numbers of rows and columns, respectively. The location of each cell at row i and column j can be represented as a Python tuple, (i, j) . Each cell of the board may contain zero, one, or multiple objects. In a valid puzzle, there will always be the same number of computers as targets.





The canonical representation represents the board as a Python list of lists of lists of strings, where the first two layers of lists are for rows and columns, and the third layer of list is to list all the objects in each location. For instance, here is a board with a computer in location $(1, 2)$ and a player and a target in location $(2, 3)$. This board has the following canonical representation:

```
[
    ["wall",  ["wall"],  ["wall"],      ["wall"],      ["wall"],
    ["wall"]],
    ["wall",  [],       ["computer"], [],              [],
    ["wall"]],
    ["wall",  [],       [],              ["target", "player"], [],
    ["wall"]],
    ["wall",  ["wall"],  ["wall"],      ["wall"],      ["wall"],
    ["wall"]]]
]
```


And our GUI for the game would render this board as follows:




3.2) Objects

There are four possible different kinds of objects in the game: the player () , computers () , walls () , and targets () .

3.2.1) The Player



The player of the game controls a python (). The player can move around the board by pressing the arrow keys. Pressing an arrow key will attempt to move the player in the given direction, subject to some interactions described below.

3.2.2) Walls

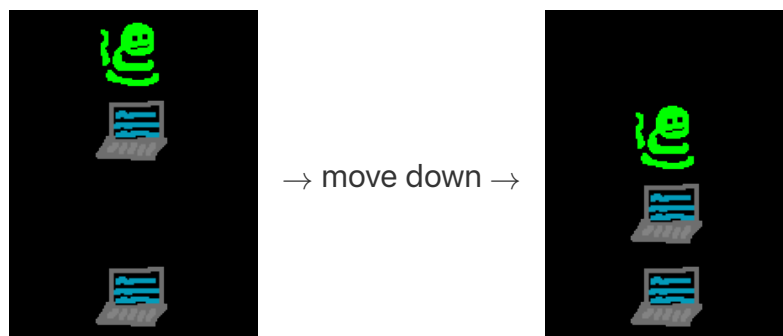
Walls () are stationary objects that prevent movement. Any object attempting to move to a location occupied by a wall instead remains in its original position.

In all of the provided puzzles, the player starts in an area that is completely enclosed by walls. Your code does not need to handle other cases.

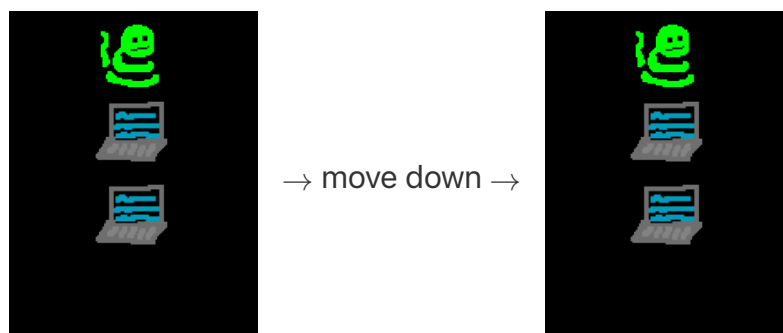
3.2.3) Computers

Computers () are objects that the player can push around the board. If the player () attempts to move to a location containing a computer, the computer should be "pushed" in the same direction in which the player was moving, *unless* doing so would move the computer in question onto a wall or another computer (in which case all objects, including the player, remain in their original positions instead).

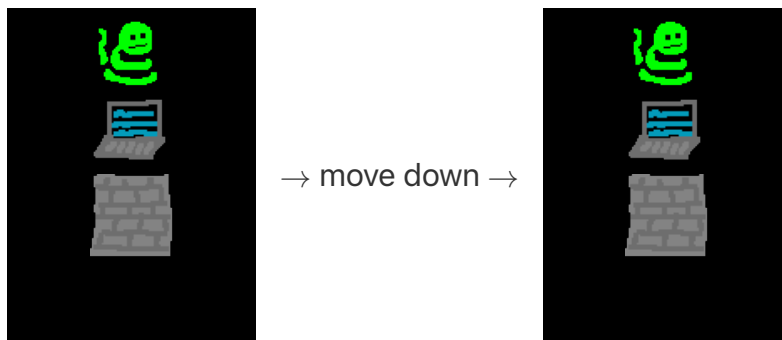
For example, starting from the board configuration shown on the left below, moving downward will push the computer down, resulting in the configuration on the right:





But starting from the other board shown on the left below, attempting to move downward will not work (because the player can only push one computer at a time).





Similarly, starting from the board shown on the left below, attempting to move downward will not work (because the player cannot push a computer through a wall).



3.2.4) Targets

Targets () represent locations to which we would like to push computers (). Targets are always stationary. The player or a computer can move onto targets.

The goal of the game is to push computers () onto targets (). The game is won when every spot containing a target also contains a computer. However, if there are no targets, the player does not automatically win; instead, this makes the game unwinnable.

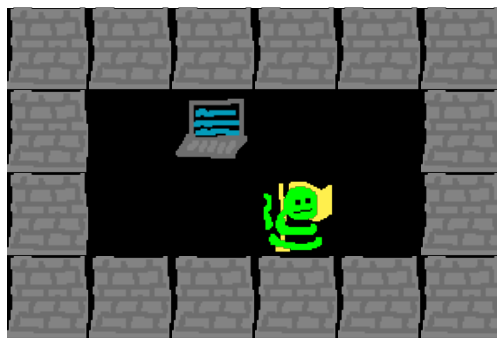
It is possible that some targets already begin in the same spots as computers, but in some cases, the solution may involve moving that computer away, either temporarily or permanently.

3.3) Check Yourself

To check your understanding of the rules of the game, answer the following questions.

Consider the board from above:

```
[
  ["wall", "wall", "wall", "wall",
  "wall", "wall"],
  ["wall", [], "computer", [],
  [], "wall"],
  ["wall", [], [], "target", "player",
  [], "wall"],
  ["wall", "wall", "wall", "wall",
  "wall", "wall"]
]
```



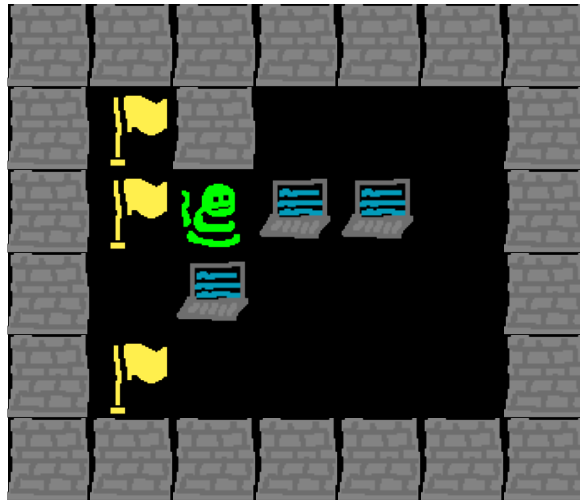
Has the player won the game?

In that same board, is it possible for the player to reach the victory state?

For the next four questions, consider this board:

```
[
  [ "wall", "wall", "wall", "wall", "wall",
  "wall", "wall" ],
  [ "wall", "target", "wall", [], [],
  [], "wall" ],
  [ "wall", "target", "player", "computer", "computer",
  [], "wall" ],
  [ "wall", [], "computer", [], [],
  [], "wall" ],
  [ "wall", "target", [], [], [],
  [], "wall" ],
]
```

```
[ ["wall"], ["wall"], ["wall"], ["wall"], ["wall"],  
["wall"], ["wall"] ]  
]
```



Starting from the board above, if you press the up arrow key, what would the player's new position be? Your answer should be a Python tuple of the form `(row, col)`, where the wall in the top-left corner is in position `(0, 0)`.

Starting from the board above, if you instead press the down arrow key, what would the player's new position be? Your answer should be a Python tuple of the form `(row, col)`.

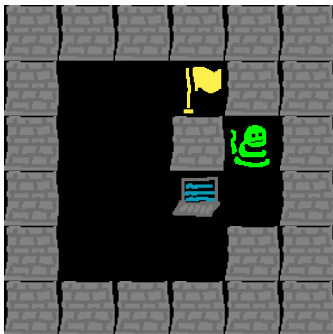
Starting from the board above, if you instead press the right arrow key, what would the player's new position be? Your answer should be a Python tuple of the form `(row, col)`.

Starting from the board above, if you press the left arrow key, what would the player's new position be? Your answer should be a Python tuple of the form `(row, col)`.

Check Yourself:

Consider a different board:

```
[
  ["wall", "wall", "wall", "wall", "wall",
  "wall"],
  ["wall", [], [], "target", "wall",
  "wall"],
  ["wall", [], [], "wall", "player",
  "wall"],
  ["wall", [], [], "computer", [],
  "wall"],
  ["wall", [], [], [], "wall",
  "wall"],
  ["wall", "wall", "wall", "wall", "wall",
  "wall"]
]
```



Starting with this board, what is the shortest sequence of actions you could take to reach the victory state?

► Show Answer

3.4) Interface

To the pass test cases that we provide, your `lab.py` file should implement the following functions, which will provide an interface to the game without restricting your choice of internal representation:

- `make_new_game` takes in the canonical representation of the game and returns your internal representation of the game state.
- `step_game` takes in the player's action and a game representation (as returned from `make_new_game`) and returns a *new* game in that same representation, updated according to one step of the game (without modifying the object that was passed in). The possible actions that a player

can take are "up", "down", "right", and "left".

- `victory_check` takes in a game (in your chosen representation) and returns a Boolean: `True` if the game has been won and `False` otherwise. A game with no computers or targets should never satisfy the victory condition.
- `dump_game` takes in a game representation (as returned from `make_new_game`) and converts it back into the canonical representation (which is used by the test cases and the GUI).

You are, of course, welcome to write whatever additional helper functions you want.

3.5) GUI

The functions above provide the programmatic interface to the game, and they should be sufficient for small-scale testing and debugging (for example, by creating a new game, stepping it multiple times, and then printing the results to see if they match your expectations), and this same interface is used by our test cases.

That said, it's not very fun to play the game that way, so we have also provided a way to play the game in the browser (using your code). To do so, run the following command in your terminal (where `/path/to/server.py` refers to the location of `server.py` in the code distribution for this lab):

```
$ python3 /path/to/server.py
```

After doing so, if you navigate to `http://localhost:6101` in your web browser, you should be able to play the game!

There are several different levels to play through, all of which were designed by [David W. Skinner](#). The particular levels we include are from the "Microban," "Mas Microban," "Sasquatch," and "Mas Sasquatch" puzzle sets. These same puzzles also act as the starting points for many of our test cases.

3.5.1) Testing and Debugging

Testing and debugging a complicated program like this is tricky! But there are strategies that you can use when, for example, a test case fails. In many cases, the GUI will be a helpful tool, as it allows you to play the game and see where the results don't match expectations. The GUI allows loading the games that the test cases use, so you can use it to walk through the steps described in the test cases (to see the sequence of steps associated with a given test case, you can open the `test_inputs/TEST_NAME.txt` file in a text editor). It can also be useful to run the same sequence of steps in both your implementation and the [reference implementation](#), to see where they differ.

In many cases, when a test case fails, the output will also provide a representation of the configuration of the board at the point where your code failed, as well as what direction the test was trying to move the player when the code failed. You can load this configuration into the GUI by copy/pasting it into the box labeled "Raw Level JSON" and clicking the "load json" button, and then you can hit the appropriate arrow key to see what happens. Trying this in your GUI and also in the [reference implementation](#) is likely to

reveal the discrepancy.

It's also the case, though, that many of the test cases we provide are quite large. You may find it helpful to define new test cases for yourself as well, using much smaller inputs.

There is also a level builder available at `http://localhost:6101/builder` to help with your testing (you need to be running the server to access this page).

3.6) Implementation


Now that you've reviewed an overview of the game rules, interface, and available testing tools, now is a great time to start planning and coding your implementation for Snekoban.

As mentioned above, you are welcome to use any internal representation of your choice for the game state. You may use the canonical form we described in [subsection 3.1](#) or really anything that you fancy as long as it works with the interface.





3.7) Hint: Breaking The Problem Down

This is a big and complicated problem, to the extent where it's not advisable to try to think about the whole problem at once. Rather, it will be useful to think about smaller pieces that can be implemented independently of each other.

Here is one possible way you could think about breaking the problem down (but feel free to approach it a different way if you prefer):

1. We could start by (temporarily) ignoring everything except the player () , making sure that the player moves as expected when `step_game` is called, but ignoring *all* object interactions and ignoring the victory-condition check.

Note that the `direction_vector` dictionary defined for you in `lab.py` may be useful for figuring out how moving in a given direction should affect the player's `(row, column)` position.

2. Next, we could make sure that interactions with walls () are handled properly, ignoring computers () and targets () . After this step, when the player tries to move into a location containing a wall, both objects should remain in their original positions.
3. Next, we could add support for pushing computers () , according to the rules described above. Even this step you may wish to break down into smaller pieces, perhaps making sure that pushing any computer works (but ignoring the effects of pushing that computer onto a wall or another computer) before introducing those additional complications.
4. Finally, we could add our victory check, at which point the game should be completely implemented!

4) Solver

By this point, you should have a functioning implementation of the logic for the Snekoban game, and you should be able to play it using the server described above. However, some of the puzzles are quite difficult!

For the remainder of the lab, we'll write a program that solves Sokoban puzzles (given a game configuration, our program will produce a list of moves that the player can make to reach the victory state).

We'll implement our solver as a function `solve_puzzle` in `lab.py`. This function takes a single input (a game, in the same format that `make_new_game` returns), and it should produce a list of moves to get to the solution (as a list of strings, e.g. `["down", "up", "left"]`) if the puzzle can be solved, or `None` if the puzzle can't be solved. In order to pass the test cases, your code should return the *shortest possible list of moves* that solves the puzzle.

Note that some of the strategies from the week 3 and 4 readings and recitations might be helpful here, but there are several complicating factors that make this pretty tricky. Among them:

- Unlike search applications we have seen in the past, this problem is asking you to return a list of *actions* rather than a list of *states*.
- Your choice of representation for the game may have had mutable elements in it, which makes keeping track of a visited set difficult.
- Depending on your choice of representation, there may also be a lot of redundant information in the game representation (which doesn't change at all as we're working through the possible solutions), and making many copies of those things can be slow.

You are welcome to tackle these issues however you wish, but our suggested approach is:

- Start by implementing a search function that returns a sequences of game board states, and then a separate helper function that takes in that sequence of states and returns a sequence of necessary actions.
- Start with a simple graph search algorithm that *does not use* a visited set at all. With that kind of algorithm, even without a visited set, your code should be possible to pass the `test_solver[tiny]` test in a few seconds (or, well under 1/10 of a second with a particularly efficient game representation). These tests are intended to allow for testing the correctness of your general approach before we start thinking about speeding things up. Of course, passing those small tests doesn't *guarantee* that you have a correct implementation, but it's a good indicator.
- Once you have that working, think about implementing a visited set. Given some of the concerns mentioned above, this step is not as easy as it may sound at first. One approach to consider would be making a helper function that takes in a board state in your chosen representation and returns a hashable object representing only the necessary information to distinguish one state from another (i.e., that avoids storing redundant information). Then, your visited set could store (and compare against) that representation, rather than storing your game representation directly.

When you think you have something working, try your code out by solving, for example, the puzzle in `m1_008.json` or `m1_001.json` and then use the GUI to test that the solution works!

5) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using

the `6.101-submit` script. The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a snekoban /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

When this page was loaded, you had not yet made any submissions.

If you make a submission, results should show up here automatically; or you may click [here](#) or reload the page to see updated results.



6) Checkoff

Note that the checkoff for this lab is due March 13th at 10pm.

Once you are finished with the code, you will need to come (in person) to any open lab time and add yourself to the [queue](#) asking for a checkoff in order to receive credit for the lab. **You must be ready to discuss your code in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions; see the notes on [style](#) for more information). In particular, be prepared to discuss:

- `new_game` Discuss your internal representation of the game (including the board, the rules, and the objects).
- `step_game` Discuss your implementation of the pushing behavior (the player pushing computers around the board) using your chosen representation.
- How did you make your solver work?
- How did you get your solver pass under the time limit?

You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.

7) Optional: Speeding Up The Solver

It turns out that writing general-purpose Sokoban solvers is complicated! You can find lots of academic papers about solving Sokoban (for example, [this paper](#) about a recent innovation), and a fair amount of

effort is still being devoted to improving solvers (you can see statistics about some of the best solvers at [this page](#)).

But even without getting into all of the fancy techniques, there are other things that can be done to speed up the solver dramatically. For example, among other things, you might consider:

- Before considering anything else, it's worth double-checking your game representation. There are certain operations that we're going to need to perform repeatedly on the game (both when playing the game and when solving it), and it would be good to try to make those as efficient as possible (for example, if there's a common operation that involves looping over the board, could we store a different representation of the game that would let us answer that question without a loop?). Those kinds of changes can make a big difference in terms of efficiency.
- Are there positions for which we know it never makes sense to move a computer? If so, are there ways that we might be able to make the search process avoid those places?