Lab 10: Snek is You

Table of Contents

1) Preparation

This lab assumes you have Python 3.6 or later installed on your machine (3.10 recommended).

The following file contains code and other resources as a starting point for this lab: `lab10.zip`

You can see and participate in online discussion about this lab in the "Labs" Category in the forum.

This lab is worth 4 points.

End of Term and Checkoff Deadline

All non-checkoff questions on this page (including your code submission) are due at 5pm on Friday, 6 May.

Because of end-of-semester regulations, the checkoff for this lab will come due on Sunday, 8 May at 7pm, which is also the last time we can accept late submissions for this semester.

Because of the smaller amount of time between lab submission deadline and checkoff deadline, checkoffs for lab 10 are already available in the help queue (and so can be completed as soon as you are finished with the lab).

Your Checkoff This Week

You are not logged in. Please log in to see information about your checkoff for this lab.

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

Baba Is You is a 2D puzzle video game with a unique twist: the rules of the game are made out of ***text** objects* that the player can *move around*, enabling the player to *change the rules* of the game as they are playing.

For example, the rule `BABA IS YOU` is made out of three **text** objects: `"BABA"`, `"IS"`, and `"YOU"`. With this rule active, all "baba" objects (which look like sheep) are controlled by the player's input: up, down, left, and right arrow keys. Each of these word objects can be pushed around by the player (like blocks in many puzzle games). In this way, the player can change the rules. For example:

- If the `"BABA"` word object gets replaced by (pushed out of the way by) a `"ROCK"` word object, then the new rule `ROCK IS YOU` means that the player's inputs control all rock objects (instead of baba objects).
- If the `"BABA"` word object gets pushed away, leaving only the invalid rule `IS YOU`, then the player inputs no longer control anything.

To get a quick sense of how the game works, you can watch the following videos (or other videos) demonstrating Baba Is You:

- Basics, Advertisement
- More Details

In this lab, we will implement a simplified version of this game called ***Snek Is You***, described below. This game will have the same core mechanics as ***Baba Is You***, but it will have a smaller set of rules and objects than the original game.

Note

This lab is fairly complex, and there is a lot of detail in this document. A good strategy is to start by looking over the document and answering the questions on the page to help make sure you understand the game, and to do so before writing any code.

Once you have read and understood the rules of the game and answered the associated concept questions, you can get started with implementation. But it is not generally adviseable to try to implement everything at once. To that end, you may find the hints in section 4.3 helpful after reading the rules but before diving in to writing the code.

Reference Implementation

We have also made a reference implementation of our version of the game available at https://hz.mit.edu/snek_is_you/, 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 also feel free to ask us questions if any behavior doesn't make sense!

3) Game Components

The game is composed of three components: the board, objects on the board, and rules that govern the behaviors of the objects. The sections below provide more information on each component. 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, etc.) and/or classes for the representations. 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. You may also find inspiration in your choice of representation for lab 2, which had some similar features to the game we'll implement here.

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 to your internal representation of the game. See section 4 for more information.

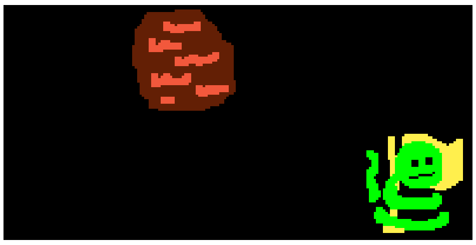
3.1) Board

The game board is an $m \times n$ grid, where m and n are the number of rows and columns, respectively. The location of each cell at row `i` and column `j` is defined as a Python tuple, `(i,j)`. Each cell of the board may contain zero, one, or multiple objects.

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, a 2×4 board with a `"rock"` in location `(0,1)` and a `"snek"` and a `"flag"` in location `(1,3)` has the following canonical representation:

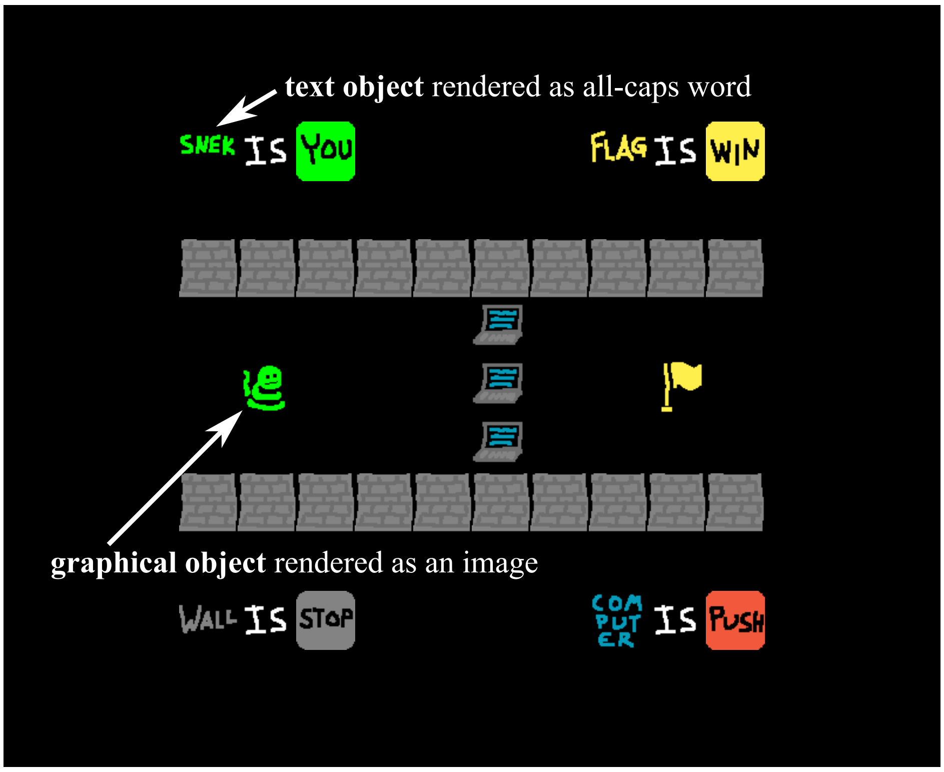
```
[
   [[], ["rock"], [], []],
   [[], [],       [], ["flag", "snek"]]
]
```

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



3.2) Objects

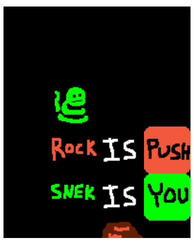
Each object on the board is either a **graphical object** or a **text object**. Here is an annotated board illustrating the difference:


text object rendered as all-caps word
SNEK IS YOU
FLAG IS WIN
graphical object rendered as an image
WALL IS STOP
COMPUTER IS PUSH

- A **graphical object** is represented by a lowercase string and rendered with an image that represents that string. For example, a `"snek"` describes a **graphical** object that renders as . In this lab, there are six graphical object types: `"snek"` ( ), `"flag"` ( ), `"rock"` ( ), `"wall"` ( ). `"bug"` ( ), and `"computer"` ( ). A **graphical** object can be walked on top of, pushed, pulled, or moved by the player (or do something else!) depending on the rules of the game defined by the **text** objects on the board.
- A **text object** is represented by an uppercase string and rendered as all-caps text. For example, `"SNEK"` is a **text** object that renders as SNEK. Multiple text objects put together make up the rules of the game, and each text object can be pushed around by the player to modify existing rules or create new rules for the game. A text object can be further classified into the following types:
  - A **noun** is a text object that represents a family of **graphical** objects on the board. Nouns are rendered as colored all-caps text. For instance, the **text** object `"ROCK"` is used to refer to all **graphical** objects represented by `"rock"` . In this lab, we will allow six nouns: `"SNEK"`, `"WALL"`, `"ROCK"`, `"COMPUTER"`, `"BUG"`, and `"FLAG"`, which render as SNEK, WALL, ROCK, COMPUTER, BUG, and FLAG, respectively.
  - A **property** refers to a behavior that can be assigned to nouns. Properties are rendered as black all-caps text with colored background. In this lab, we will focus on six properties, namely `"YOU"`, `"STOP"`, `"PUSH"`, `"PULL"`, `"DEFEAT"`, and `"WIN"`, which render as YOU, STOP, PUSH, PULL, DEFEAT, and WIN, respectively. Behavior associated with each property is described in the next section. To distinguish properties from other types of words, they are drawn as black text on a colored square, instead of the usual colored text on black background.
  - A **verb** `"IS"` ( ) gives object types specific behaviors. Specifically, a sequence of three text objects *`NOUN`* `IS` *`PROPERTY`* forms a rule that assigns all behaviors represented by *`PROPERTY`* to all graphical objects represented by *`NOUN`*.
  - A **conjunction** `"AND"` ( ) can combine (union) together nouns or properties in a rule.

Answer the following questions about the concepts described above. For the next four questions, consider this board:

```
[
  [ [],       ["snek"], [],       [] ],
  [ [],       ["ROCK"], ["IS"],   ["PUSH"] ],
  [ [],       ["SNEK"], ["IS"],   ["YOU"] ],
  [ [],       [],       ["rock"], [] ],
```



```
    [ [],      [],      [],      [] ]
]
```

What is the size of the board in the example above? Your answer should be a Python tuple of the form `(number_of_rows, number_of_columns)`.

In the example above, how many **text** objects are there in the board?

In the example above, how many **graphical** objects are there in the board?

In the example above, how many **graphical** objects does `"ROCK"` refer to on the board?

3.3) Properties

Next we describe the precise behaviors of the properties in Snek Is You. For specificity, we will use `"ROCK"` as the running example of the noun that properties are assigned to, but the descriptions below apply equally to any noun.

3.3.1) Default Behavior

If no properties apply to `"ROCK"`, the default behavior of a `"rock"` **graphical** object () is to do nothing. The object does not interact with other objects. Other objects are allowed to occupy the same location (they will be "on top of" the `"rock"`).

In contrast, **text** objects like `"ROCK"` (Rock) always have the behavior of `"PUSH"`, defined below. Nothing that happens in the game can change this.

3.3.2) "YOU" You

If `"ROCK"` has the `"YOU"` property, then all `"rock"` graphical objects are controlled by the player. Pressing the arrow keys will attempt to move all the `"rock"`s in the corresponding direction. The move will fail (and that particular `"rock"` won't move at all) if the `"rock"` tries to move outside the game board, or if it encounters a graphical object with `"STOP"` behavior, possibly after a chain of graphical objects with `"PUSH"` behavior (see definitions below).

Note that if nothing has the `"YOU"` property, then the player cannot control anything!

3.3.3) "STOP" Stop

If `"ROCK"` has the `"STOP"` property, then other objects cannot move onto the same location as a `"rock"` graphical object. More precisely, any object attempting to move onto a `"rock"` graphical object will instead not move at all.

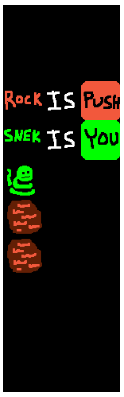
For example, if a graphical object that has the `"YOU"` property tries to move onto a `"rock"` by pressing an arrow key, then that `"YOU"` object won't move (but other graphical objects with the `"YOU"` behavior might still move).

3.3.4) "PUSH" Push

If `"ROCK"` has the `"PUSH"` property, then all `"rock"` graphical objects can be **pushed**. This means that, if another object (e.g. with `"YOU"` or `"PUSH"` property) tries to move into the same location as a `"rock"`, then that `"rock"` also tries to move one cell in the same direction. If the `"rock"` is unable to move (because it would go outside the game board, or would try to move onto an object with `"STOP"` behavior, or there is a chain of pushes that ends in such), then the object trying to move into the same location as the `"rock"` is also unable to move.

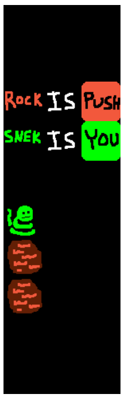
As an example, consider the following game board:

```
[
    [["ROCK"], ["IS"], ["PUSH"]],
    [["SNEK"], ["IS"], ["YOU"]],
    [["snek"], [],     []],
    [["rock"], [],     []],
    [["rock"], [],     []],
    [[],       [],     []]
]
```



If the player presses the down arrow, the result will be:

```
[
    [["ROCK"], ["IS"], ["PUSH"]],
    [["SNEK"], ["IS"], ["YOU"]],
    [[],       [],     []],
    [["snek"], [],     []],
    [["rock"], [],     []],
    [["rock"], [],     []]
]
```



If the player presses the down arrow again, the board will not change, because the second `"rock"` object cannot be pushed in the down direction, so the first "rock" can't move, and so the `"snek"` can't move either.

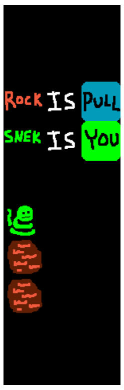
If `"ROCK"` has both the `"PUSH"` and `"STOP"` properties, then the `"PUSH"` behavior takes priority. Essentially, a `"rock"` graphical object will (try to) get pushed out of the way before an object tries to move onto the `"rock"` (which would trigger the `"STOP"` behavior).

3.3.5) `"PULL"` PULL

If `"ROCK"` has the `"PULL"` property, then all `"rock"` graphical objects can be **pulled**. This means that, if another object adjacent to a `"rock"` moves *directly away* from the `"rock"`, then the `"rock"` tries to moves in the same direction. (The move might fail, though, if pulling the `"rock"` would move it onto something with the `"STOP"` behavior).

For example, consider this game board:

```
[
    [["ROCK"], ["IS"], ["PULL"]],
    [["SNEK"], ["IS"], ["YOU"]],
    [[],       [],     []],
    [["snek"], [],     []],
    [["rock"], [],     []],
    [["rock"], [],     []]
]
```



If the player presses the up arrow, the result will be

```
[
    [["ROCK"], ["IS"], ["PULL"]],
    [["SNEK"], ["IS"], ["YOU"]],
    [["snek"], [],     []],
    [["rock"], [],     []],
    [["rock"], [],     []],
    [[],       [],     []]
]
```

Rock IS PULL
SNEK IS YOU

And then if the player presses the right arrow, the result will be

```
[
    [["ROCK"], ["IS"],   ["PULL"]],
    [["SNEK"], ["IS"],   ["YOU"]],
    [[],       ["snek"], []],
    [["rock"], [],       []],
    [["rock"], [],       []],
    [[],       [],       []]
]
```

Rock IS PULL
SNEK IS YOU

Note that the `"rock"`s did not move the second time because `"snek"` was not moving directly away from them.

3.3.6) "DEFEAT"

If `"ROCK"` has the `"DEFEAT"` property, and if a graphical object that has the `"YOU"` property stands on the same cell as a `"rock"` graphical object, then the `"YOU"` object disappears (though the player will still be able to control any other remaining graphical objects that have the `"YOU"` property).

As a special case, if `"ROCK"` has both the `"DEFEAT"` property and the `"YOU"` property, then all `"rock"` graphical objects will disappear.

For example, consider this game board:

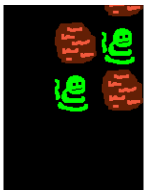
```
[
    [["ROCK"], ["IS"],   ["DEFEAT"]],
    [["SNEK"], ["IS"],   ["YOU"]],
    [[],       ["snek"], ["rock"]],
    [["snek"], ["rock"], ["snek"]],
    [["snek"], ["snek"], ["rock"]]
]
```

Rock IS DEFEAT
SNEK IS YOU

If the player presses the right arrow, the result will be

```
[
    [["ROCK"], ["IS"],   ["DEFEAT"]],
    [["SNEK"], ["IS"],   ["YOU"]],
    [[],       [],       ["rock"]],
```

```
    [[],       ["rock"], ["snek"]],
    [[],       ["snek"], ["rock"]]
]
```



Note that three of the "snek"s were removed from the game (specifically, those that ended up in the same cell as a "rock" after moving).

"DEFEAT" rules are evaluated after all movement rules but before "WIN" (which is described below).

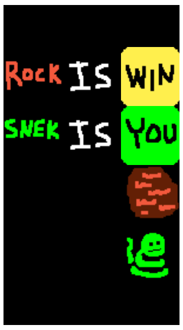
3.3.7) "WIN"

If "ROCK" has the "WIN" property, then the player wins the game if any graphical object that has the "YOU" behavior stands on the same cell as a "rock" graphical object.

As a special case, if "ROCK" has both the "WIN" property and the "YOU" property, then the player wins automatically as long as there is at least one "rock" graphical object in the game.

For example, the player will win the game after pressing the up arrow from either of the following game boards:

```
[
    [["ROCK"], ["IS"], ["WIN"]],
    [["SNEK"], ["IS"], ["YOU"]],
    [[],       [],     ["rock"]],
    [[],       [],     ["snek"]]
]
```



```
[
    [["ROCK"], ["IS"],  ["YOU"]],
    [["ROCK"], ["IS"],  []],
    [[],       [],      ["WIN"]],
    [[],       [],      ["rock"]]
]
```

Rock IS YOU
Rock IS
WIN

"WIN" rules are evaluated *after* all other rules. For example, if a graphical object that has the "YOU" property stands on the same cell as an object with "DEFEAT" property and another (or the same) object with "WIN" property, the "DEFEAT" will be handled first and the player won't win the game.

In particular, "WIN" rules aren't evaluated until after the first step in the game. So if the initial state has a winning condition, it doesn't count; the player needs to makes a step and try to preserve that condition.

3.4) Rules

At any point in the game, the relative locations of the **text** objects on the board determine what rules are currently in force in the game. The simplest case is when there are three text objects adjacent to each other horizontally or vertically:

```
SUBJECT IS PREDICATE
```

or

```
SUBJECT
IS
PREDICATE
```

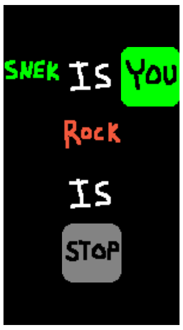
where *SUBJECT* is a noun (text object), and *PREDICATE* is either a noun or a property (text object).

3.4.1) Property Rules

If the given *PREDICATE* is a property, then the associated behavior applies to all **graphical** objects of the type given by the *SUBJECT* (that is, objects represented by the lowercase form of that noun). The behaviors for all of the properties are described in the preceding section.

For example, if the current state of the game is as follows:

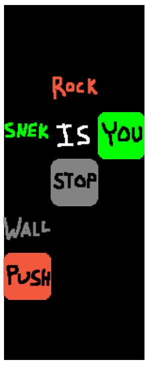
```
[
    [["SNEK"], ["IS"],    ["YOU"]],
    [[],       ["ROCK"],  []],
    [[],       ["IS"],    []],
    [[],       ["STOP"],  []]
]
```



Then there are two rules currently in force: `SNEK IS YOU` and `ROCK IS STOP`. These rules mean that all `"snek"` **graphical** objects have the behavior of `"YOU"` and all `"rock"` **graphical** objects have the behavior of `"STOP"`.

Depending on how things are laid out, the same **text** object may be part of multiple rules. For example, consider this game board:

```
[
    [[],       ["ROCK"],  []],
    [["SNEK"], ["IS"],    ["YOU"]],
    [[],       ["STOP"],  []],
    [["WALL"], [],        []],
    [["PUSH"], [],        []]
]
```



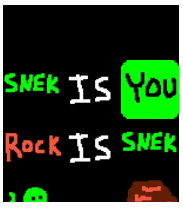
The rules currently in force are `SNEK IS YOU` and `ROCK IS STOP` (they share the same `"IS"` text object). `"WALL"` and `"PUSH"` are not currently part of any rules.

3.4.2) Noun Rules

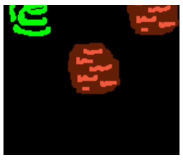
Instead of a property, the *PREDICATE* of a rule could be a noun. In that case, the rule means that, after all movement is complete for a given step, we should replace all graphical objects associated with the *SUBJECT* by graphical objects associated with the *PREDICATE*.

For example, `ROCK IS SNEK` is a rule that tells the game to change all `"rock"` graphical objects into `"snek"` graphical objects. If the game board looks like the following after movement (e.g., after the `snek` graphical object pushed the `ROCK` text object up one):

```
[
    [["SNEK"], ["IS"],   ["YOU"]],
    [["ROCK"], ["IS"],   ["SNEK"]],
    [["snek"], [],       ["rock"]],
```

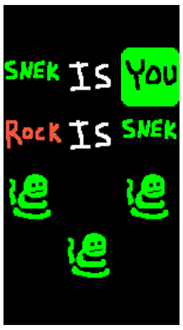


```
    [[],        ["rock"], []]
]
```



Then the `ROCK IS SNEK` rule will apply, resulting in

```
[
    [["SNEK"], ["IS"],   ["YOU"]],
    [["ROCK"], ["IS"],   ["SNEK"]],
    [["snek"], [],       ["snek"]],
    [[],       ["snek"], []]
]
```



This type of rule allows rules to overlap in an interesting way. For instance, `SNEK IS ROCK IS YOU` being on adjacent tiles (in order) gives us two rules: `SNEK IS ROCK` and `ROCK IS YOU`.

Note also that it is possible to have a sequence of these rules; for example, we may have two rules `SNEK IS ROCK` and `ROCK IS WALL`. In this case, any object that is a `"snek"` will change into a `"rock"`, and every `"rock"` will be replaced with a `"wall"`. But note that any single object may change types at most once per turn (so if the rules above are in force, it takes two turns for `"snek"` objects to ultimately turn into `"wall"` objects).

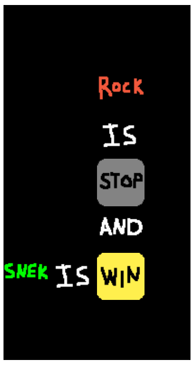
3.5) AND

A rule can also have more than one subject and more than one predicate by using `"AND"` **text** objects. For example, all the following formats are valid:

```
SUBJECT IS PREDICATE AND PREDICATE
SUBJECT AND SUBJECT IS PREDICATE
SUBJECT AND SUBJECT IS PREDICATE AND PREDICATE AND PREDICATE
```

In such cases, the rules of all the PREDICATEs apply to all the SUBJECTs. For example:

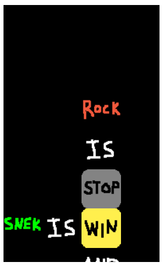
```
[
    [[],       [],      ["ROCK"],   []],
    [[],       [],      ["IS"],     []],
    [[],       [],      ["STOP"],   []],
    [[],       [],      ["AND"],    []],
    [["SNEK"], ["IS"],  ["WIN"],    []]
]
```



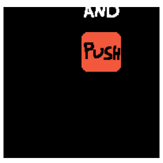
There are three rules currently in force: `SNEK IS WIN`, `ROCK IS STOP`, and `ROCK IS WIN`. All `"snek"` **graphical** objects have the behavior of `"WIN"`. All `"rock"` **graphical** objects have both the behaviors of `"STOP"` and `"WIN"`.

Alternatively, in the next example:

```
[
    [[],        [],     ["ROCK"],  []],
    [[],        [],     ["IS"],    []],
    [[],        [],     ["STOP"],  []],
    [["SNEK"],  ["IS"], ["WIN"],   []],
    [[],        [],     ["AND"],   []],
```



```
    [[],        [],      ["PUSH"],  []]
  ]
```



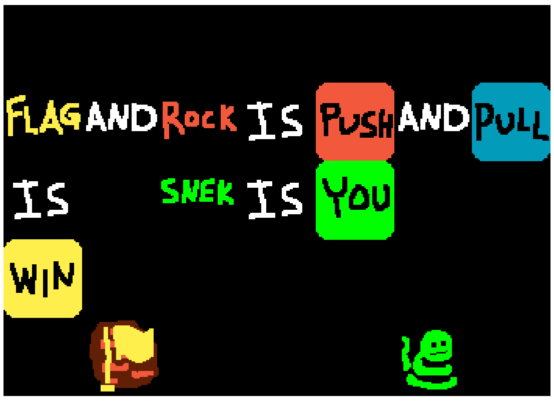
"snek" objects still only have the behavior of "WIN". "rock" objects only have the behavior of "STOP". "PUSH" does not apply to any object.

In our tests, we ensure that at most one noun exists in the set of predicates associated with any given subject. For example, we will never have both `SNEK IS BUG` and `SNEK IS ROCK` in play. You are welcome to handle that ambiguity however you wish, but our tests will never expect any particular behavior for that set of rules.

3.5.1) Check Yourself

Answer the following questions about the rules of the game described above. For the questions below, consider this board:

```
  [
    [ [],       [],                [],       [],      [],        [],       []],
    [ ["FLAG"], ["AND"],           ["ROCK"], ["IS"],  ["PUSH"],  ["AND"],  ["PULL"]],
    [ ["IS"],   [],                ["SNEK"], ["IS"],  ["YOU"],   [],       []],
    [ ["WIN"],  [],                [],       [],      [],        [],       []],
    [ [],       ["flag", "rock"],  [],       [],      [],        ["snek"], []]
  ]
```



What are all the rules? Each rule should be broken down into its simplest form. The answer should be a Python set of strings, where each string is in all-caps.

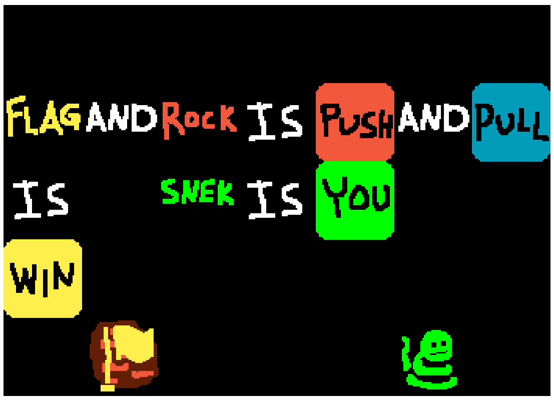
Starting from the board above, if you move up by 1 cell, what would be the new location for "rock"?

Starting from the board above, if you instead move left by 1 cell, what would be the new location for "snek"?

3.6) Check Yourself

Answer the following questions about the rules of the game described above. For the questions below, consider the same board as above:

```
[
  [ [],        [],                []],        [],       [],         [],        []],
  [ ["FLAG"], ["AND"],            ["ROCK"], ["IS"],  ["PUSH"], ["AND"], ["PULL"]],
  [ ["IS"],    [],                ["SNEK"], ["IS"],  ["YOU"],  [],        []],
  [ ["WIN"],  [],                []],        [],       [],         [],        []],
  [ [],        ["flag", "rock"], [],        [],       [],         ["snek"], []]
]
```



Starting from the board above, if you move 4 cells to the left, what will be the new locations of all the **graphical** objects on the board? Your answer should be a Python set of tuples, where each tuple represents the location (`row`, `col`) of a **graphical** object.

Is the resulting state a victory?

--

Starting from the board above, if you instead took the moves ["left", "up", "up"], what will be the new locations of all the **graphical** objects on the board? Your answer should be a Python set of tuples, where each tuple represents the location (`row, col`) of a **graphical** object.

Is this a state of victory?

--

3.7) Order of Operations

All of the pieces above are expected to happen in a particular order. Putting everything together, the overall process for one timestep of the game is:

1. Move objects according to the movement-based properties (`"YOU"`, `"STOP"`, `"PUSH"`, `"PULL"`)
2. Parse the **text** objects and update the rules accordingly
3. Adjust object types based on rules whose predicate is a noun
4. Handle the `"DEFEAT"` property
5. Handle the `"WIN"` property

4) Implementation

You don't need to be writing any code yet, but now is a great time to start thinking about implementing Snek Is You.

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 a dictionary as we did in the Minesweeper lab, or really anything that you fancy. You should also consider how to represent the different types of objects. A Python string object may be sufficient to get the job done, or Python classes could make sense if you'd like to organize the different objects with methods and/or hierarchically.

4.1) Interface

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

- `new_game` takes in the canonical representation of the game and returns your internal representation of the game state. Recall that the canonical representation is a Python list of lists of lists of strings, where UPPERCASE strings represent **text** objects and lowercase strings represent **graphical** objects.
- `step_game` takes in the player's action and a game representation (as returned from `new_game`). Unlike our function from lab 2, here `step_game` should **modify that game representation in-place** according to one step of the game. The possible actions that a player can take are `"up"`, `"down"`, `"right"`, and `"left"` (which should cause all objects associated with the `"YOU"` property to move in that direction).

  `step_game` should then return a Boolean: `True` if the game has been won at the end of the step, and `False` otherwise.
- `dump_game` takes in a game representation (as returned from `new_game`), and converts it back into our canonical representation (which is used by the test cases and the GUI).

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

4.2) 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:6009` in your web browser, you should be able to play the game! We have implemented several different levels that you can play through, and the starting point for each test case is also included to help with testing and debugging.

You can add your own levels for testing out your implementation in the GUI as well, if you are so inclined (see subsection 5.1).

4.3) Hints

4.3.1) Representation

The choice of how to represent the state of the game is entirely in your hands, but eventually you will need representations for *at least* the following things:

- the board,
- the objects described above (including both **graphical** and **text** objects), and
- the rules of the game (which, when we're done, will be able to change).

You may represent these things however you want in your code, but a good place to start is by asking yourself: what do we need to keep track of related to each of those things? And what kinds of operations do we need to perform on those things? Those questions can help you figure out how to represent things, and what supporting structure you need in order to update them appropriately.

Now (before you start implementing anything!) is a great time to start thinking about representation! It's also worth noting, though, that as you work through the process of writing the code, you may want or need to change your choice of representation. That is OK, too, and is an expected part of the process!

It may be helpful to start by implementing a small subset of the desired functionality, and building up from there (see one suggested approach in the section below).

4.3.2) Breaking The Problem Down

This is going to be a big and complex program, to the extent where it's not adviseable 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) hard-coding the rule `SNEK IS YOU`, i.e., that `"snek"` **graphical** objects should move according to the `"YOU"` property. To start, we can ignore all **text** objects and all **graphical** objects that are not `"snek"` objects and only implement the `"YOU"` behavior.

   For now, you can also hard-code `False` as the return value for `step_game`, i.e., we will never consider ourselves to have won the game.

   Doing so, you should be able to pass 2 test cases: `snek_is_you` and `snek_is_you2` (though `little_rock`, and `rock_is_not_a_word` will also pass now, by coincidence); and you should be able to move `"snek"` objects around in the GUI by using the arrow keys (and it should not interact with other objects in any way).

2. Next, we can add new rules one-at-a-time by hard-coding them and checking the resulting behaviors. For example, hard-coding `WALL IS STOP` and implementing the `"STOP"` behavior allows passing `test_wall_is_stop`; and similar relationships exist for the following rules, all of which can be (temporarily) hard-coded:

   - `WALL IS STOP`
   - `ROCK IS PUSH`
   - **text** objects are *all* `PUSH` as well
   - `COMPUTER IS PULL`
   - `BUG IS DEFEAT`
   - `FLAG IS WIN`

   With all of these rules hard-coded and the associated behaviors implemented, we expect that a correct implementation should pass the first 19 test cases.

   However, note that passing these tests not not necessarily guarantee that the associated behaviors are completely correct; they may still be worth another look when debugging.

3. Next, we can remove our hard-coded rules and add basic support for parsing the rules on the board. For simplicity's sake, a good place to start might be by only looking for rules of the form *`NOUN`* `IS` *`PROPERTY`*.

   That is, we can keep complexity down by temporarily ignoring `AND`, and also temporarily ignoring rules whose predicate is a noun.

   A correct implementation up to this point should pass the first 39 test cases.

4. Lastly, we can add support for `AND` and for rules that have nouns as predicates. These pieces can be added in either order, but each should allow us to pass a few more test cases.

   And, of course, when those are implemented, your code should pass all of the test cases!

4.3.3) Testing and Debugging

Note About `test.py`

The `test.py` file for this lab differs a little bit in terms of structure from what we've seen before in other labs. Note that while there is only a single test function defined in `test.py`, that function will be used to test the behavior of your program in several situations.

If you want to run a particular test case, you can use the `-k` flag to `pytest`; for example, `$ pytest test.py -k weird_solution` will run the test called `weird_solution`.

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.

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. See subsection 5.1 for how to test custom levels in the GUI.

5) Optional Extensions

Of course, even once you're passing the test cases, there is more room to play around! Here are some possible suggestions for extensions to the game:

5.1) More Levels

The GUI determines the levels that are available by looking for files in the `ui/puzzles` directory in the code distribution. You can add puzzles of your own creation (or following one of the examples in this document) by making one of two different kinds of files:

- a file whose name ends with `.json`, containing the canonical representation of the game in JSON format (like any of the examples listed in this document)
- a file whose name ends with `.txt`, containing a smaller one-character-per-cell representation that is easier to type (but can only represent one object in each cell)

A GUI to help generate these files is available at `http://localhost:6009/builder` when running `server.py`.

5.2) More Baba Is You Features

A natural extension is to add more features from Baba Is You, for example, you could add:

- more subjects
- more predicates (see this wiki page for examples)
- more verbs and conjunctions, and even prepositions! (see this wiki page for examples)

One thing to note is that our GUI will not know about any new words you add to the language unless you also make the associated images and put them in the `ui/` directory!

6) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `submit-009-lab` 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:

```
$ submit-009-lab -a lab10 /path/to/lab.py
```

Running that script should submit your file to be checked, and it should also provide some information about how and where to get feedback about your submission. Reloading this page after submitting will also show some additional information:

> You have not yet made any submissions to this assignment.

7) Checkoff

Once you are finished with the code, please come to office hours and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code and test cases 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). In particular:

- Briefly describe your internal representation of the game (including the board, the rules, and the objects). Were there key points during the lab at which you needed to reconsider your representation? How else could you have represnted things, and why did you end up with the representation you chose.
- Briefly describe your high-level approach to implementing `*NOUN* IS *NOUN*`-style rules. How does your code prevent multiple rules, for example, `SNEK IS ROCK` and `ROCK IS BUG` from turning `snek` objects directly into `bug` objects?
- Briefly describe your high-level approach to parsing rules from the game state

You have not yet received this checkoff.