Tent Packing

Submission to website: Monday, 20 March, 10pm

Checkoff by LA/TA: Thursday, 23 March, 10pm

This lab assumes you have Python 3.5 installed on your machine. Please use the Chromium or Firefox web browsers.

Our expectation is that `pack` will be **recursive**! Look over the lectures on recursion and recursive search for ideas to use in your implementation.

This lab has 15 tests and 3 coding points. The coding points will be assigned during check-off based on the organization and readability of your code.

- remember not to use imports!
- include documentation (docstrings or comments) for all your helper functions that explain the expected argument values/types and what's get returned. If the function modifies one of its arguments (e.g., a list or dict) that should be mentioned.
- good comments include a "mission statement" for each block of code: functions, loops, if's, etc. For example, "Try all pieces to see if they fit at the current location." Or "No empty squares means we've solved the puzzle." It should be possible to get an accurate sense of the algorithm you've implemented just by looking at the comments. If one has to read the code to know what's happening, you don't have enough comments yet! You don't have to comment every line, but you should include comments if it's not obvious what's happening or if there's some particular constraint your code relies on for correct operation.

Introduction

You decided to go camping with your `N` friends over Spring Break. Unfortunately, you have one big tent and your lazy friends didn't bother to bring their own. To accomodate your friends, you must figure out a way to arrange sleeping bags optimally in your tent. To make matters worse, after setting it up, you realize that several spots under your tent have rocks.

Your assignment is to find a way to pack the tent with sleeping bags such that: 1. No one is sleeping on a rock. 2. Every usable (non-rock) portion of the tent is being utilized. If no such arrangement exists, you must correctly conclude that no such packing exists.

Representing Tents and Friends

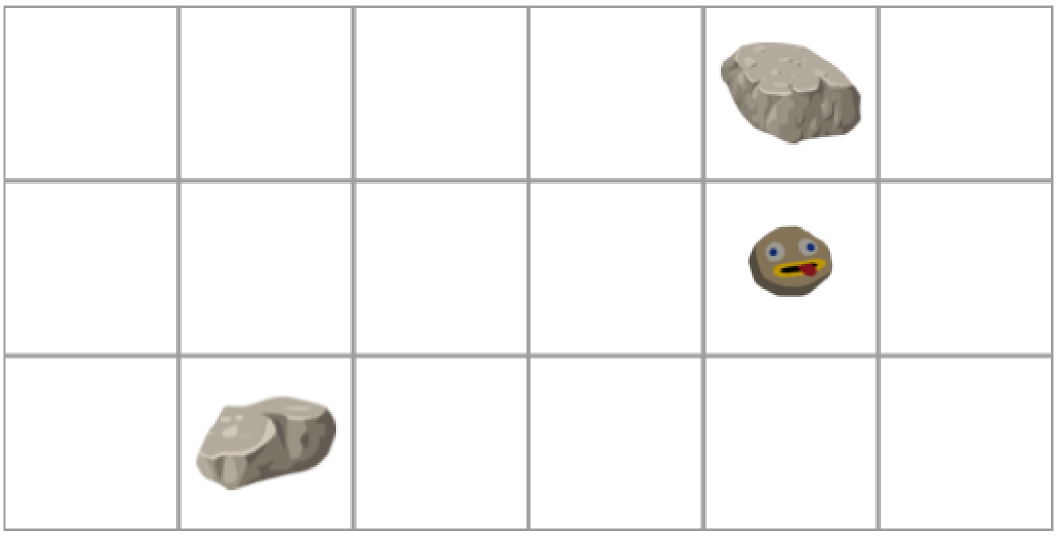
We will use a 2-dimensional grid to represent the tent. Each rock occupies one square in this grid. A tent configuration is described by variables:

- `tent_size`, which is a Python `tuple` with two integers `(nrows,ncols)` -- the dimensions of the tent in terms of number of rows and number of columns in the grid.
- `missing_squares`, which is a Python `set` (possibly empty) of the grid squares with rocks under them. Each square is represented as a tuple with two integers `(row,col)`, the coordinates of the square. `(0,0)` is at the top left corner, with row numbers increasing down the page and column numbers increasing left-to-right.

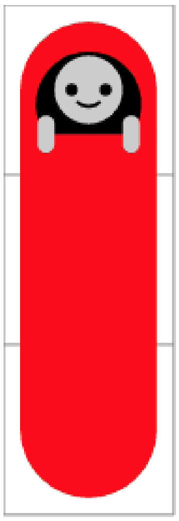
For example, a tent configuration would be represented as:

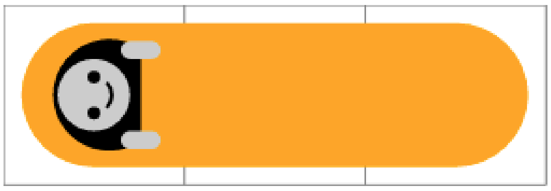
```
tent_size = (3,6)
missing_squares = { (2,1), (0,4), (1,4) }
```

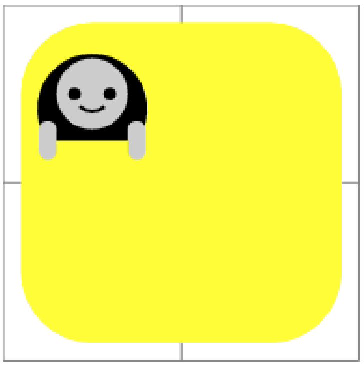
This could denote the following tent configuration -

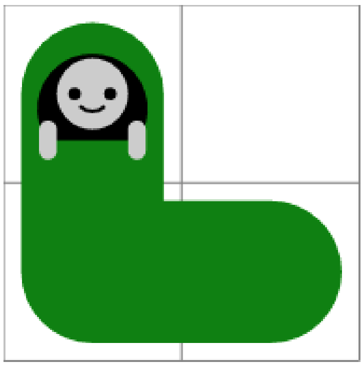


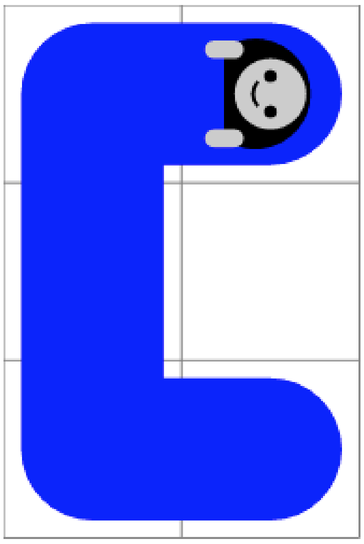
It turns out your friends are quite flexible and can sleep in any of the following sleeping bag shapes:

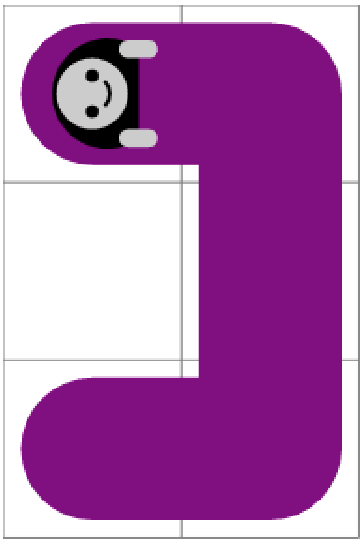

bag_list[0]


bag_list[1]


bag_list[2]


bag_list[3]


bag_list[4]


bag_list[5]

You only have to place the bags in the tent in the orientation shown -- don't worry about rotations or reflections.

A sleeping bag shape is described by a set of tuples that give the row and column of each square occupied by the sleeping bag, relative to the top-left square of the bag which has coordinate `(0,0)`. Note that all sleeping bag shapes will occupy their top-left square.

For example, the orange horizontal 1x3 bag is described by the set

```
{ (0,0), (0,1), (0,2) }
```

and the blue C-shaped bag by the set

```
{ (0,0), (0,1), (1,0), (2,0), (2,1) }.
```

You will be given a list of possible sleeping bag shapes, i.e., a list of sets, called `bag_list`. We'll be using the shapes shown above, but your code should be general enough to handle any shape given in `bag_list`.

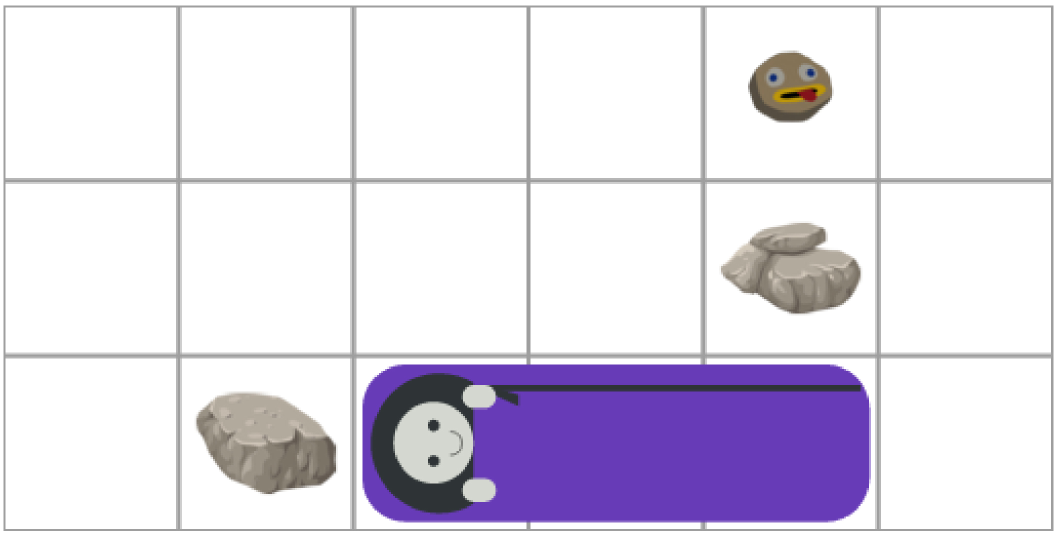
A friend's position and shape are represented by a dictionary with two keys:

- `"anchor"` is a tuple `(row,col)` giving the coordinates of **top left square** of the position occupied by the friend's sleeping bag.
- `"shape"` specifies a particular sleeping bag shape as an integer index into `bag_list`.

For example, a person would be represented as a dictionary:

```
{ "anchor": (2,2), "shape": 1 }
```

In the example tent above this would correspond to:



Valid Tiling

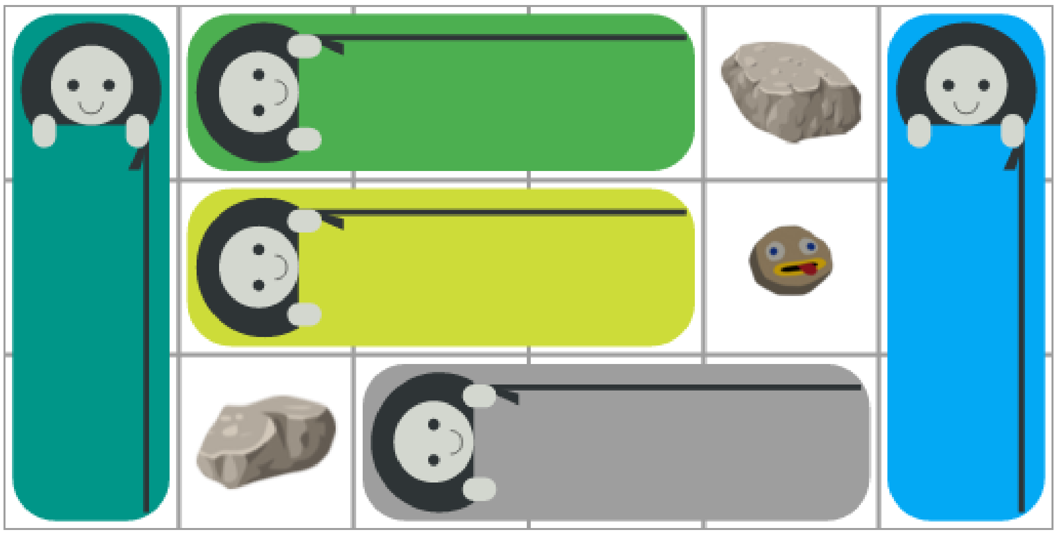
Let's say our tent has dimensions `nrows` by `ncols`. A valid tiling is a list of people (with `anchor` and `shape` values) such that:

- For each square `(r,c)` occupied by a sleeping bag
  - `0 <= r < nrows` and `0 <= c < ncols` (i.e. each person lies entirely within the tent).
  - No rock exists under `(r,c)` (i.e. no person sleeps on a rock).
- No two people have a square in common (i.e. no two people overlap).
- Every non-rock square is occupied by a person.

For example, let's say we are given the input from the **Representation** section.

```
tent_size = (3,6)
missing_squares = { (2,1), (0,4), (1,4) }
```

The following is a valid tiling for this tent with three 1x3 bags and two 3x1 bags.



The corresponding list of people (in no particular order) would look like:

```
[
  {"anchor": (0,1), "shape": 1},
  {"anchor": (1,1), "shape": 1},
  {"anchor": (2,2), "shape": 1},
  {"anchor": (0,0), "shape": 0},
  {"anchor": (0,5), "shape": 0},
]
```

lab.py

You must implement your code in this file. You are not expected to read or write any other code provided as part of this lab. You have to correctly implement the function

```
pack(tent_size, missing_squares, bag_list)
```

in the file `lab.py`. The meaning and type of the three arguments are described above.

If there exists a complete tiling of the the non-rock squares with no overlap, the function should return a list of people which results in a valid tiling. Each person should be represented with a dictionary (as described in the previous section) with keys `"anchor"` and `"shape"` and valid corresponding values.

If there is no valid tiling, the function should return `None`.

Your code will be loaded into a tiny web server (`server.py`) which, when running, serves the Tent Packing interfaces from your very own computer acting as a web server (at http://localhost:8000 -- your computer's own address at port 8000).

Run `./server.py` and go to the url http://localhost:8000 on your browser.

In-Browser UI `./server.py`

Once your code produces output of a correct type (list of dictionaries with keys `"anchor"` and `"shape"`), it's time to debug your logic!

You can visualize the output to help debug your code. Run `server.py` and open your browser to `localhost:8000`. You will be able to select any of the test inputs from the `./cases` folder and examine them in the browser.

You can visualize the output produced by your code by pressing the `RUN` button. This will display the tiling that your code outputs. If you output `None`, it will color the grid red.

Auto-grader (unit tester)

As before, we provide you with a `test.py` script to help you verify the correctness of your code. The script will call `pack` from `lab.py` with the test cases in the `cases` folder and verify their output.

You will find that `test.py` is not very useful for debugging. We encourage you to use the UI and to write your own test cases to help you diagnose any problems and further verify correctness.

Go forth and code. Good luck. Start early!

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `lab.py` at web.mit.edu/6.009 and get your lab checked off by a friendly staff member.