CS 120: Introduction to Computer Programming II

Carcassonne Project, Long A

see the Short A spec for due dates

1 Overview

See the spec for Short A to see an overview for the entire multi-part project.

In this part of the project, you will implment the `CarcassonneMap` class, along with 8 new tile types.

Place the class `CarcassonneMap` in the file `carcassonne_map.py`. The class `CarcassonneTile`, along with the various `tile*` objects, must be defined in the file `carcassonne_tile.py`.

2 New Required Tiles (Long A)

In this part, all tiles from previous part(s) are still required, and we will add the tiles below.

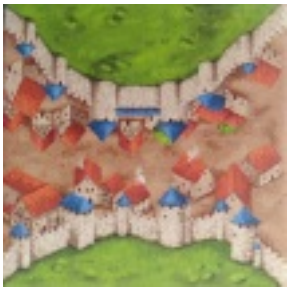
Tile 05

(Ignore the "shield" icon in the upper-left portion of the tile. It's part of Carcassonne, but not sometime we'll model in this project.)

Tile 06

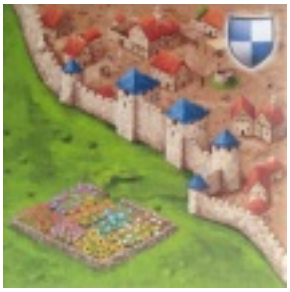
Tile 07

Notice that this tile has two edges which are cities, but which are not connected to each other.



Tile 08

Notice that this tile has **exactly** the same edges as the previous one, but different city connectivity.



Tile 09

Tile 10

Tile 11

Tile 12

(spec continues on next page)

3 New Method for `CarcassonneTile`: `rotate()`

In addition to the methods required in Short A, your Tile class must add a `rotate(self)` method. This method takes no parameters (other than `self`) and returns a **new** object, which represents the same tile, but rotated clockwise by 90 degrees.

The user may call `rotate()` on a rotated tile, to rotate it another 90 degrees; this can happen as many times as you wish.

It is **critical** that the returned object be different than the original object; do **not** modify the original object. This is because the map may have many copies of the same tile - some rotated, some not.

3.1 Caching?

Generally, I assume that you will create a new object each time that `rotate()` is called. This means that if you call `rotate()` twice on the same tile, like this:

```
a = tile01.rotate()
b = tile01.rotate()    # probably not the same as a
```

then the two objects will be different objects. Likewise, if you rotate a tile 4 times (bringing it back to its original state), I assume that you will create 4 new objects, the last one not the same as the start:

```
c = tile01.rotate().rotate().rotate().rotate()   # probably not the same as tile01
```

However, if you choose to add a feature which eliminates these duplicates, it is **allowed but not required.**

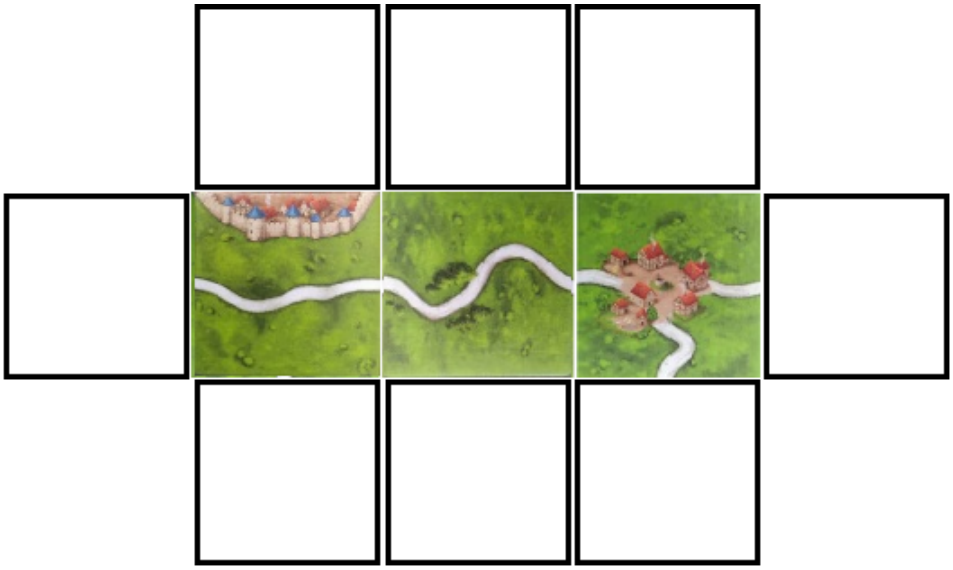
4 `CarcassonneMap` - Overview

A Map object represents a bunch of tiles, which have been laid on the board. The same tile can be added in many different places (we suppose that, in real life, there are duplicates of some tiles), and some of the tiles that are placed may be rotated any number of times. (If a tile is to be rotated, we will always do this **before** placing it.)

The first tile must always be `tile01` (not rotated), placed at $(0, 0)$; the constructor for `CarcassonneMap` takes no parameters, and must place that tile, at that location.

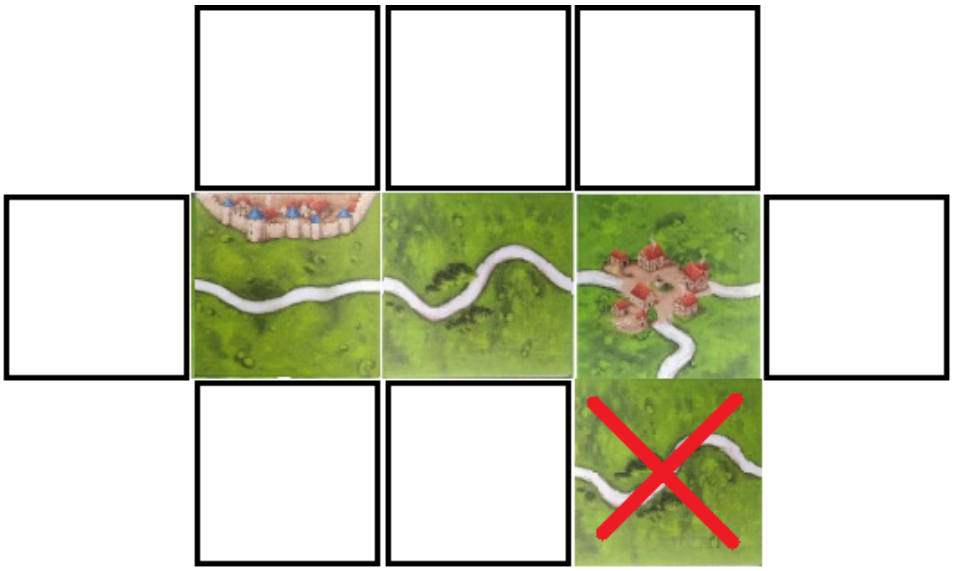
From there on, it is only legal to place new tiles that share at least one edge with an existing tile (not just a corner), and every tile that is placed must exactly match the "edge type" of any existing tiles. Thus, any tile placed to the West of the starting tile (that is, at $(-1, 0)$) must have `"grass+road"` on its East edge; similarly, any tile placed to the North, at $(0, 1)$, must have `"city"` on its South; and any tile placed on the South, at $(0, -1)$, must have `"grass"` on its North. Sometimes, a newly-placed tile will touch two or more existing tiles; in that case, it must match on **all** of the sides where it touches.

EXAMPLE: Places where a new tile could be added, if the map held three tiles in a certain arrangement.

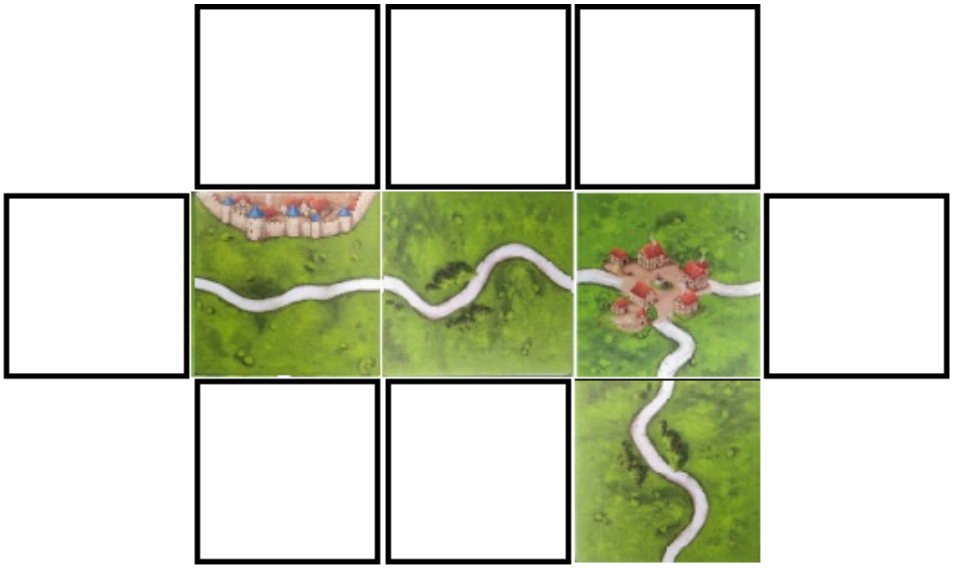


FAILURE EXAMPLE: The tile cannot be added, because the new tile has grass on its South side, but the existing tile requires a city.

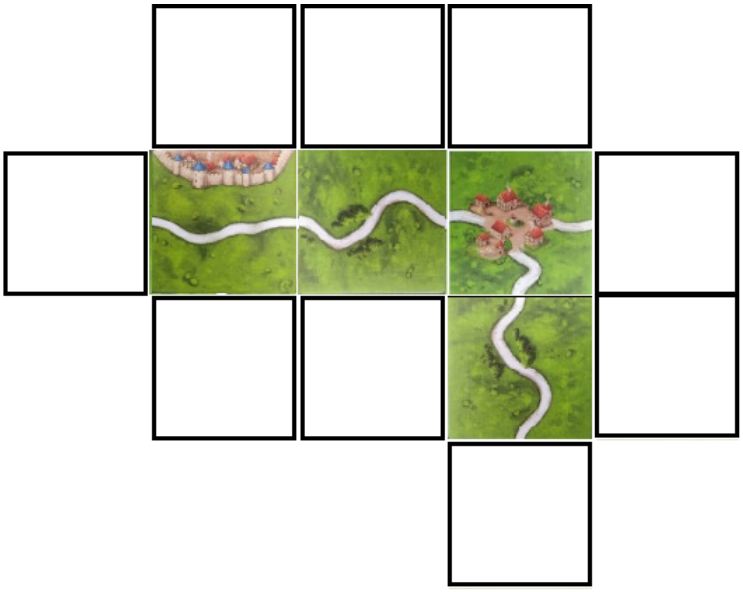
FAILURE EXAMPLE: The new tile cannot be placed in this location, beause it has not been rotated.



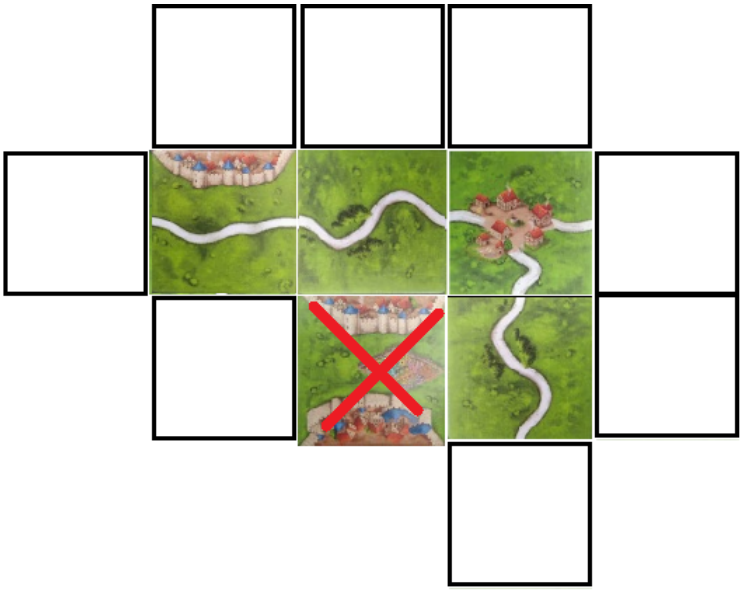
CORRECTION EXAMPLE: The new tile has been rotated, and so now fits the location.



EXAMPLE: After the new tile has been added, the set of locations where tiles can be placed has changed.



FAILURE EXAMPLE: The new tile cannot be placed in this location. While it matches the existing tiles on one side, it does not match on another.



4.1 Representing the Grid

Since the tiles are all placed on a grid, you may be tempted to represent it using a 2D array. This is permissible, but it may be difficult to use because the map routinely uses negative indices - and also because the map can have many holes in it.

While you have complete freedom, one simple option would be to build a dictionary, which maps (x, y) coordinates to tiles.

5 Required Methods of `CarcassonneMap`

In Long A (we will add more in the future), your Map class must support the following methods.

- `__init__(self)`

  Constructor. The initial map must contain `tile01`, not rotated, at $(0, 0)$, and no other tiles.

- `get_all_coords(self)` This returns all of the (x, y) coordinates of the current tiles in the map, as a set. Right after your constructor builds the object, it must return `{(0,0)}` .

- `find_map_border(self)`

  Returns a `set`, which contains all of the (x, y) locations of the places where new tiles can be added, based on the current tiles in the map.

- `get(self, x,y)`

  Returns the Tile at the specified (x, y) location, or `None` if no such tile exists.

  In all methods of this class, you may assume that `x,y` are both integers, although you should not assume anything about their range.

- `add(self, x,y, tile, confirm=True, tryOnly=False)`

  Adds a given tile, at the given x,y location. Returns `True` if the tile was added to the map, or `False` if there was some reason it could not be added. If the caller wants to rotate the tile, the tile must be rotated **before** it is passed to this method.

  It has two parameters with defaults: `confirm` defaults to `True`, and `tryOnly` defaults to False. They mean:

  `confirm=True, tryOnly=False`
  The default. You should perform all of the error checking, and add the tile if possible. Return a boolean to indicate whether it was added or not.

  `confirm=True, tryOnly=True`
  Perform all of the error checks, and return a boolean indicating whether or not adding the tile is possible. But **do not** actually add it to the map,

no matter what. (This will be used to test your error-checking code in testcases.)

`confirm=False, tryOnly=True`
Invalid combination. The testcases will never do this.

`confirm=False, tryOnly=False`
Simply add the tile, with no error checking at all. (This will be used, in testcases, to set up the map before the test begins; we never want you to reject one of these `add()`s.)

6 Turning in Your Solution

You must turn in your code using GradeScope.