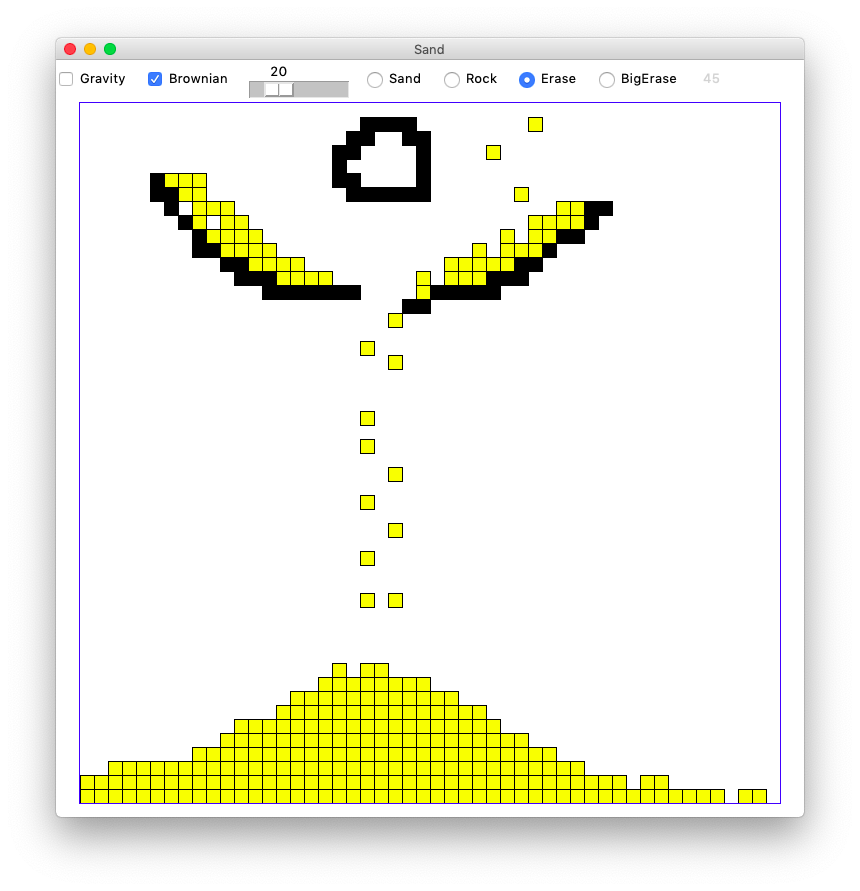
# Falling Sand

For this project you will write 2-d algorithmic code to implement a kind of 2-d world of sand. What my kids have described as the world's worst version of Minecraft. When it's working, it's kind of fun to play and watch in its low-key way.

The starter code handles setting up the GUI window, and handling the controls and drawing. All the logic that makes the world work will be built by you.



## Grid Class

We will use the simple CS106A Grid utility class to store the 2-d date. See [Grid Reference](https://web.stanford.edu/class/cs106a/handouts/reference-grid.html)

Using the Grid looks like this

grid = Grid(3, 2) # make 3 by 2 grid, initially all None

grid.set(0, 0, 'hi') # set a value at 0,0

val = grid.get(0, 0) # get a value out at 0,0

## Decomposition

This is a big program, and we want the advantages of a divide-and-conquer strategy. We'll build the program up as a series of decomposed functions. We will leverage Python Doctests, testing each function in isolation before moving on to the larger functions. You need to write 5 functions to make the world work.

## How Does The Sand Grid Work?

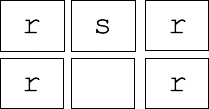
Every square in the Sand grid holds one of three things:

1. Sand represented by 's'

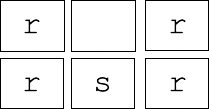
2. Rock represented by 'r'

3. Empty represented by None

Here is a 3 by 2 grid with some rocks and one sand. The 's' is at x=1 y=0 (aka 1,0) with our usual origin upper-left system.



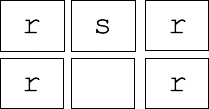
If we move the sand down from 1,0 to 1,1 the grid looks like this:



Moving sand down by one square like this is the central operation of this game.

## Grid Literals in Python

There is syntax in Python code for writing out a 2-d grid. It's not too pretty, but it works fine for small grids. Here is the first grid from above:



Here is the Python syntax to create that grid in memory:

grid = Grid.build([['r', 's', 'r'], ['r', None, 'r']])

The ['r', 's', 'r'] is the first row (y=0), followed by the second row ['r', None, 'r']. You would not want to write out a big grid this way. Fortunately the tests can all be done with small grids like this one.

## a. do\_move()

Look at the do\_move() function. The """Pydoc""" defines what this function needs to do: get the value that is at x1,y2, and move it to be at x2,y2 in the grid, and return the changed grid to the caller. Two Doctests are provided.

def do\_move(grid, x1, y1, x2, y2):

"""

Given grid and 2 coordinates.

Move the value that is at x1,y1 to x2,y2,

and return the resulting grid.

Assume that this is a legal move: all coordinates are in

bounds, and x2,y2 is empty.

(i.e. a different function checks that this is a

legal move before do\_move() is called)

(Doctests provided)

>>> grid = Grid.build([['r', 's', 's'], [None, None, None]])

>>> do\_move(grid, 1, 0, 1, 1)

[['r', None, 's'], [None, 's', None]]

>>>

>>> grid = Grid.build([['r', 's', 's'], [None, None, None]])

>>> do\_move(grid, 2, 0, 2, 1)

[['r', 's', None], [None, None, 's']]

"""

The code for do\_move() is short, but it is good example of using Doctests to spell out test cases. In this case, the 2 Doctests are provided.

Here is the first Doctest which moves the 's' at 1,0 down to 1,1.

>>> grid = Grid.build([['r', 's', 's'], [None, None, None]])

>>> do\_move(grid, 1, 0, 1, 1)

[['r', None, 's'], [None, 's', None]]

What the 3 lines of the Doctest do:

1. grid = .. This sets up a grid variable for the next 2 lines. The grid is tiny: width 3 height 2, just enough to write a little test.

2. do\_move(grid, 1, 0, 1, 1) - calls your do\_move() function, moving the 's' at 1,0 to 1,1. The function returns the changed grid.

3. The third line shows what the grid should look like post-move: [['r', None, 's'], [None, 's', None]]. The Doctest machinery verifies that the grid coming out of do\_move() matches this written state.

## do\_move() Code

Write the code for do\_move(). It's a short function. Here's a reminder of the two key grid functions:

grid.get(x, y) - returns what value is stored at an x,y - None or 's' or 'r'

grid.set(x, y, val) - sets a new value in the grid at an x,y

Run the Doctests until they all pass. Get a feel for how the Doctests work, so in later parts you can write your own.

Doctest pro-tip: when you have run a Doctest once, you can re-run it by clicking the green "play" button at the left-edge of the window. Running a Doctest again and again is the common pattern as work out each function.

## b. check\_move() Legal Moves

The check\_move() function is given a prospective x1,y1 x2,y2 for a move, and returns True if the move is ok, or False otherwise. The grid is not changed by this operation.

Here is the Pydoc for the check\_move() function.

def check\_move(grid, x1, y1, x2, y2):

"""

Given grid and x1,y1 and destination x2,y2.

Check if it's possible to move the value at x1,y1 to x2,y2.

The x1,y1 location is always in bounds, but x2,y2 may not be.

Return True if the move is ok, or False otherwise.

Ok move: x2,y2 in bounds, empty, not violating corner rule.

We'll call x2,y2 the "destination" of the move. In the Sand world, there are five possible moves: left, right, down, down-left, and down-right. Here are the rules for a legal move:

Rule 1. The destination must be within the edges of the grid.

Three Doctests for this rule are provided. These tests build a 1 row by 1 col grid, checking that a few out-of-bounds moves return False.

>>> # Provided out-of-bounds tests

>>> # Make a 1 by 1 grid with an 's' in it to check in-bounds cases

>>> grid = Grid.build([['s']])

>>> check\_move(grid, 0, 0, -1, 0) # left blocked

False

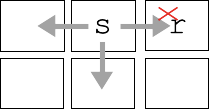
>>> check\_move(grid, 0, 0, 0, 1) # down blocked

False

>>> check\_move(grid, 0, 0, 1, 1) # down-right blocked

False

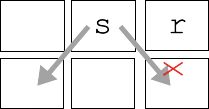
Rule 2. The destination square in the grid must be empty.



Above is a picture os a single 's' at 1,0 in a 3 by 2 grid. The move left to 0,0 and the move down to 1,1 are both ok (return True). The move right to 2,0 is bad, blocked the the 'r' there (return False).

Rule 3. For a diagonal down-left or down-right move, the corner square must be empty (None).

Consider the down-left and down-right moves of the 's' here:



The "corner" rule: for a down-left or down-right move, the corner square **above** the destination must also be empty. So in the picture, the down-left is ok since 0,0 is empty, but down-right is bad due to the rock at 2,0. Sand at 2,0 would also block the move.

## check\_move() Doctests

The starter code includes one test for a left move in a 3 by 2 world:

>>> # Provided checks of left and right moves of 's'

>>> grid = Grid.build([[None, 's', 'r'], [None, None, None]])

>>> check\_move(grid, 1, 0, 0, 0) # left ok

True

Add at least 4 more tests for other moves in the 3 by 2 world, such as right, down, down-left, down-right. Include tests with a mixture of True and False results. The tests do not need to be comprehensive. The beauty of Doctests is that, in practice, a handful of tests can expose most bugs.

## check\_move() code

With the tests describing the many cases done, write the code for the check\_move() function. Note that the grid has a grid.in\_bounds(x y) function that returns True if a particular x,y is in bounds or not.

There are many reasonable ways to structure this code. Obviously the one requirement is that the code returns the correct answer for all cases. My solution has a single return True at the bottom of the function, and many if ... return False detecting the various cases where the move is bad.

Use the Doctests as you work out the code. Sometimes a test fails because the function is wrong. Other times, upon review, you realize that your *test* is wrong, not spelling out the correct grid state.

Note that it's fine to use == and != in comparisons like this if x != None. Disregard warnings PyCharm gives you about that line. There warnings in this case are misguided.

## c. do\_gravity()

Consider an x,y in the grid. This function implements one "gravity" move for that x,y as follows. In our gravity algorithm, the moves are handled in a specific order:

1. If there is not a sand 's' at x,y, do nothing, the move is over.

2. down: if the sand can move down, do it, this ends the move.

3. down-left: otherwise if the sand can move down left, do it, this ends the move.

4. down-right: otherwise if the sand can move down right, do it, this ends the move.

In all cases, return the grid when the function is done. Use your helper functions to do the work. That is the key to this function. How can you tell if the way is clear for the sand to move, for example, down? We provide a rich set of Doctests for this function, but of course you still need to write the code to actually solve the problem. You are always free to add more tests.

def do\_gravity(grid, x, y):

"""

Given grid and a in-bounds x,y. If there is a sand at that x,y.

Try to make one move, trying them in this order:

move down, move down-left, move down-right.

Return the grid in all cases.

>>> # not sand

>>> grid = Grid.build([[None, 's', None], [None, None, None]])

>>> do\_gravity(grid, 0, 0)

[[None, 's', None], [None, None, None]]

>>>

>>> # down

>>> grid = Grid.build([[None, 's', None], [None, None, None]])

>>> do\_gravity(grid, 1, 0)

[[None, None, None], [None, 's', None]]

>>>

>>> # bottom blocked

>>> grid = Grid.build([[None, 's', None], ['r', 'r', 'r']])

>>> do\_gravity(grid, 1, 0)

[[None, 's', None], ['r', 'r', 'r']]

>>>

>>> # rock-below down-left

>>> grid = Grid.build([[None, 's', None], [None, 'r', None]])

>>> do\_gravity(grid, 1, 0)

[[None, None, None], ['s', 'r', None]]

>>>

>>> # sand-below down-right

>>> grid = Grid.build([[None, 's', None], ['s', 's', None]])

>>> do\_gravity(grid, 1, 0)

[[None, None, None], ['s', 's', 's']]

>>>

>>> # sand corner: down-right

>>> grid = Grid.build([['s', 's', None], [None, 's', None]])

>>> do\_gravity(grid, 1, 0)

[['s', None, None], [None, 's', 's']]

>>>

>>> # at bottom already

>>> grid = Grid.build([[None, None, None], [None, 's', None]])

>>> do\_gravity(grid, 1, 1)

[[None, None, None], [None, 's', None]]

## d. do\_whole\_grid()

For the moment, ignore the "brownian" parameter which is handled in a later step.

Write code and tests for a do\_whole\_grid() function which calls do\_gravity() once for every x,y in the grid. Return the grid when done. Write two tests, with at least one test featuring a 3x3 world with sand at the top row. Pass 0 for the brownian parameter to do\_gravity().

The standard y/x nested loops go through the coordinates top-down, and normally that's fine. However, in this case, it's important to reverse the y-direction, going bottom-up, i.e. visit the bottom row y = height-1 first and the top row y = 0 last. Suggestion: use the reversed() function.

What's wrong with regular top-down order? Suppose the loops went top-down, and at y=0, a sand moved from y=0 down to y=1 by gravity. Then when the loop got to y=1, that sand would get to move *again*. Going bottom-up avoids this problem.

Run your Doctests in do\_whole\_grid() to see that your code is plugged in and working correctly.

Clarification: do\_whole\_grid() does one "turn" of the world, calling do\_xxx() a single time for each square. The provided GUI code calls this function again and again when the gravity checkmark is checked.

## Milestone - Run sand.py

With your functions tested, you can try running the whole program. Gravity should work, but brownian is not done yet. Normally when a program runs the first time, there are many problems. But here we have leaned on decomposition and testing pretty hard, so there is a chance your code will work perfectly the first time. If your program works the first time, try to remember the moment. More typically, new code is not so well tested and so exhibits many problems when run the first time.

Bring up the terminal and run the program like this (no command line arguments are required, on Windows its "py" or "python"):

$ **python3 sand.py**

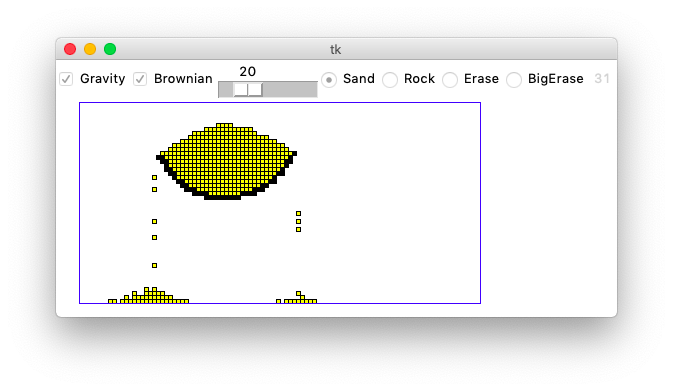
## Running With Different Sizes

You can provide 2 command line numbers to specify the number of squares wide and high for the grid. The default is 50 by 50 squares. So the following creates a grid 100 by 50

$ **python3 sand.py 100 50**

An optional third parameter specifies how many pixels wide each square should be. The default is 14. So this creates a 100 by 50 world with little 4 pixel squares:

$ **python3 sand.py 100 50 4**



The Sand program is pretty demanding on your computer - it's a lot of computation run continuously. You may notice the fans on your laptop spinning up. At the upper right of the window is a little gray number, which is the frames-per-second (fps) the program is achieving at that moment, like 31 or 52. The animation has a nice, fluid look when the fps is higher.

The more grid squares there are, and the more grains of sand there are, the slower the program runs. For each gravity round, your code needs to at least glance at every square and every grain of sand, and we want to do that 40 times per second. Play around with different grid and pixel sizes to try out the different esthetics. My laptop is 5 years old, so I'm curious how much better fps figures a newer computer can get.