Flood Fill and Maze Path Finding

This reading is relatively new, and your feedback will help us improve it! If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions, please get in touch during office hours or open lab hours, or via e-mail at `6.101-help@mit.edu`.

Table of Contents

1) Introduction

For the last couple of weeks worth of readings, we've been kind of down in the weeds, so to speak, looking at the details of Python's operation through the lens of our environment model. In this reading, we're going to come at things from a much higher level and think about constructing a program. Over the term, we'll tend to kind of alternate between these two perspectives, and we'll hopefully see how one informs the other.

Our main focus for this reading is going to be writing a program to implement a behavior we call *flood fill*, an operation on images which we'll describe in much more detail below. Before we get there, though, let's take a moment to introduce (or review) a couple of useful features of Python.

2) Some Cool/Useful Python Features

Before we dive in to this week's material, we'll spend a bit of time to provide a brief introduction to a couple of nice built-in features of Python, which we might make use of in code in readings and/or recitations in the future.

These introductions will go by a little bit quickly, so we encourage you to try things out on your own computer, to read Python's documentation, and/or to ask the staff for help with these new structures.

2.1) Built-in Function `zip`

`zip` is a nice Python built-in that allows us to easily find corresponding elements in multiple iterable objects. ("Iterable" means we may use `for` loops to step through all their elements.) For example, consider the following code, which is designed to perform an element-wise subtraction of two lists:

```
def subtract_lists(l1, l2):
    assert len(l1) == len(l2)
    out = []
    for ix in range(len(l1)):
        out.append(l1[ix] - l2[ix])
    return out
```

We can come at this problem in a slightly different way using the `zip` built-in function in Python. `zip` takes multiple iterable objects as input, and it returns a structure that can be looped over. Let's look at a small example to get a bit familiar with `zip` before we use it to rewrite `subtract_lists` from above.

Imagine we have two lists, `x` and `y`, defined as below:

```
x = [100, 200, 300, 400]
y = [9, 8, 7, 6]
```

If we call `zip` on these two objects, Python gives us back a `zip` object:

```
print(zip(x, y))  # prints <zip object at SOME_MEMORY_LOCATION>
```

That doesn't look very useful on its own, but `zip` objects exist to be looped over. For example, we can look at the following code:

```
for element in zip(x, y):
    print(element)
```

This prints the following:

```
(100, 9)
(200, 8)
(300, 7)
(400, 6)
```

Can you see the pattern? Each element that this `zip` object produces is a tuple containing corresponding elements from `x` and `y`. That is, the first tuple we see contains `x[0]` and `y[0]`; then the next contains `x[1]` and `y[1]`; and so on. So (assuming `x` and `y` have the same length) this is equivalent to the following code:

```
for i in range(len(x)):
    print((x[i], y[i]))
```

2.2) Tuple Unpacking

The `zip` function is often used in combination with another handy Python feature: using *multiple assignment* to name each element of a tuple. For example:

```
i, j = (100, 9)
```

This code has the effect of assigning *two* variables, `i` and `j`, to the corresponding parts of the tuple. So `i` ends up with the value `100`, and `j` with the value `9`. You can do this with any number of variables, and with lists as well:

```
x, y, width, height = [0, 0, 100, 200]
```

If the tuple or list on the righthand side doesn't have exactly the same number of elements as there are variable names on the left side, then this code would raise an error.

Since `zip` creates tuples, tuple unpacking is a handy way to name the elements of the lists that are being zipped together. For example:

```
for x_value, y_value in zip(x, y):
    print(x_value + y_value)
```

For each 2-element tuple that `zip` returns, the variables `x_value` and `y_value` are assigned to the two parts of the tuple --- meaning, really, that `x_value` ends up stepping through the elements of the `x` list, while `y_value` simultaneously steps through the corresponding elements of the `y` list. This code prints the following:

```
109
208
307
406
```

This is useful in the context of our list-subtraction problem, which we can rewrite to make use of `zip` and tuple unpacking:

```
def subtract_lists(l1, l2):
    assert len(l1) == len(l2)
    out = []
    for i, j in zip(l1, l2):
        out.append(i - j)
    return out
```

Using `zip` is never strictly necessary, but there are many situations where using it can produce some very nice code. We will see more examples throughout 6.101 readings and recitations, and we encourage you to practice with it on your own!

For now, try out some additional examples to answer the following questions:

What happens if the arguments we give to `zip` have different lengths?

- We will get an exception.
- We will only get a number of tuples equal to the length of the shortest of all of the inputs.
- We will get a number of tuples equal to the length of the longest of all the inputs, as though we had added `None` to the end of the shorter inputs to make the lengths equal.

What happens if we give more than two arguments to `zip`?

- We will get an exception.
- All arguments beyond the second will be ignored.
- The tuples that we get will be longer than two elements (they will have one element for each input we gave to `zip`).

2.3) The `*` Unpacking Operator

We can also unpack a tuple or list into its individual elements when we are calling a function. For example, here is a function we'll be using in the example code in this reading:

`get_pixel(image, row, col)`: return the color of the pixel at location (row, col) in the given image.

This function expects row and column as separate arguments, but our code will be working with two-element `(row,col)` pairs, like `location = (23, 11)`. We can extract the parts of the pair and pass them as separate arguments using the unpacking operator `*`:

```
get_pixel(image, *location)
```

which in this case is the same as

```
get_pixel(image, location[0], location[1])
```

As with tuple unpacking, we can use this operator not just on tuples, but also on lists, and the tuple or list can have any length. But if it has the wrong length, so that we end up trying to give the wrong number of arguments to the function, then Python will raise an error.

2.4) List Comprehensions

Let's also discuss one more very helpful feature of Python, which you may have seen before. It is often the

case, when we're writing programs, that we want to build new lists based on the contents of other lists. For example, let's imagine we have a list defined for us, called `L`:

```
L = [9, 8, 7, 6, 5, 4, 3]
```

and let's imagine we wanted to make a new list that contains the double of each odd number in `L`. Here is one way we could write that piece of code:

```
out = []
for number in L:
    if number % 2 == 1:  # if number is odd
        out.append(number * 2)  # add double that number to our output
```

This form of program is very common, the general structure being:

```
out = []
for VARIABLE_NAME in SOME_ITERABLE_OBJECT:
    if SOME_CONDITION:
        out.append(SOME_EXPRESSION)
```

Python gives us a convenient short-hand notation for this kind of structure, which we call a "list comprehension." We can more concisely create a list like the above by using the following general structure instead:

```
[SOME_EXPRESSION for VARIABLE_NAME in SOME_ITERABLE_OBJECT if
SOME_CONDITION]
```

So, for our example from above, we could have done something like the following instead:

```
out = [number * 2 for number in L if number % 2 == 1]
```

If we had instead wanted the double of *every* number in `L`, we could have omitted the conditional altogether:

```
out = [number * 2 for number in L]
```

You can also put multiple nested `for` loops and/or `if` conditions into a list comprehension. The easiest way to think about this is to first imagine how you would write it with `for` and `if` statements. Then take those `for`s and `if`s and just move them to the end of your list comprehension, in the same order. So:

```
out = []
for x in <sequence of x>:
    if <condition on x>:
        for y in <sequence of y>:
            if <condition on y>:
                out.append(<some expression using x,y>)
```

... becomes:

```
[
    <some expression using x,y>
    for x in <sequence of x>
    if <condition on x>
    for y in <sequence of y>
    if <condition on y>
]
```

Notice that the `for`s and `if`s are in exactly the same order in both ways of writing the code. We have also written the list comprehension on multiple lines here for clarity. (Python allows multiple lines inside parentheses, but it doesn't require them like it did when we were using `for` loops.) Breaking a long list comprehension onto multiple indented lines is definitely a good idea. Finally, note that the colons after the `for` and `if` need to be omitted when you're writing a list comprehension, because it's an expression, not a block of statements.

Can the `if` in a list comprehension be followed by `else` or `elif`, like a normal `if` statement? It turns out the answer is no, because the purpose of this `if` is to *filter* the resulting list -- producing an element if the condition is true but omitting it entirely if not. So the "else" part is always just "skip this element," rather than code you can write. In cases where you want `else` behavior, then you need to put it in the leading expression instead, using Python's "one-line if" syntax (also called the conditional expression or ternary operator):

```
[(SOME_EXPRESSION if SOME_CONDITION else OTHER_EXPRESSION) for
VARIABLE_NAME in SOME_ITERABLE_OBJECT]
```

It's worth mentioning that it is never *necessary* to use list-comprehension syntax, but it is a very common and convenient Python idiom. We will use it a lot in example code in 6.101, and it's worth practicing in your own code as well!

Now that we've seen a few example list comprehensions, try your hand at answering the following questions (and if you get stuck, that's OK; feel free to ask for help/clarification!).

Consider the following piece of code, which constructs a list called `out`:

```
out = []
for val in x:
    if val < 0:
        out.append(-val)
```

Firstly, try to read this code and make sure you understand what it is trying to do. Then, write a list comprehension in the box below that produces the same list (you may assume that `x` is defined for you):

`out` = [-val for val in x if val<0]

Assuming we have a list of numbers defined for us called `y`, write a list comprehension that makes a new list containing the squares of all the numbers in `y`.

`squares` = [x*x for x in y]

Check Yourself:

Can you use this list-comprehension idea to write an even more concise version of `subtract_lists` from above?

Show/Hide

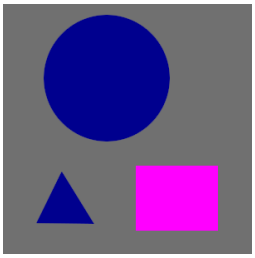
Here is one solution:

```
def subtract_lists(l1, l2):
    assert len(l1) == len(l2)
    return [i - j for i, j in zip(l1, l2)]
```

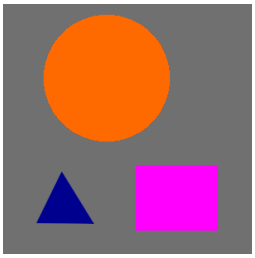
With these features in hand, let's turn our attention to the problem for this week's reading.

3) Flood Fill

In this week's reading, we'll continue using a similar image representation to the one we used in Labs 1 and 2, adding a new feature (common in many image-editing programs) known as "flood fill" (or, sometimes, "paint bucket"). The idea in that context is that, when using the paint-bucket tool and clicking on a particular spot in an image, the result will be an image where the color that was clicked on is replaced with a new color (but only in the contiguous region around the pixel that was clicked). For example, consider the following small image:

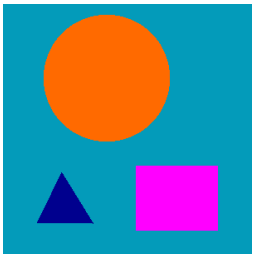


If we were to use the paint-bucket tool and click anywhere in the circle to replace it with an orange color, we would see the following as a result:



Note that the whole blue region around the spot where we clicked has turned orange but that the blue triangle did not change color since it was not connected to the blue circle.

If we were then to click anywhere in the grey background to replace it with a cyan color, we would see the following as a result:



This is the problem we will tackle this week. We will approach this problem by writing a function called `flood_fill`, described as follows:

```
def flood_fill(image, location, new_color):
    """
    Given an image, replace the same-colored region around a given
location
    with a given color.  Mutates the original image to reflect the
```

```
change.

    Parameters:
      * image: the image to operate on
      * location: a (row, col) tuple representing the starting location
of the
                  flood-fill process
      * new_color: the replacement color, as an (r, g, b) tuple where all
values
                   are between 0 and 255, inclusive
    """
    pass
```

> **Note: Color Representation**
>
> Note that the choice of representation here (in particular using a tuple to represent a color) is very similar to the representation we are using in Image Processing 2. You may wish to take another look at Lab 2's description of color images before continuing on.

3.1) Following Along

We encourage you to follow along in your own text editor as we develop this program over the course of the reading. As usual, we encourage you to type out the programs and go through all the same steps we describe in the reading so that you can experience the process for yourself.

The following zip file has some starter code that you can use to get started: `flood_fill.zip`

The relevant code is in `flood_fill.py`, and a few example images are included as well. This code uses a library called `pygame` to display images to the screen while it is running, so, if you want to follow along, you will need to install `pygame`, which you should be able to do with the following command (or the equivalent for your Python setup):

```
$ pip3 install --upgrade pygame
```

If you are having trouble installing `pygame`, please feel free to reach out in open lab hours or via the 6.101-help@mit.edu mailing list!

3.2) Helper Functions

Throughout this reading, we will make use of a few helper functions to make reading and writing the code a

little bit nicer. In particular, we will assume the existence of the following functions when writing our code for `flood_fill`:

- `get_width(image)`: given an image, return its width in pixels (as an integer).
- `get_height(image)`: given an image, return its height in pixels (as an integer).
- `get_pixel(image, row, col)`: return the color of the pixel at location (row, col) in the given image.
- `set_pixel(image, row, col, color)`: mutate the given image to replace the color at location (row, col) with the given color.

The `flood_fill.py` file contains implementations of these functions corresponding to a `pygame`-based image representation, but we could also implement versions of these functions for our image representation from the image-processing labs. With those helpers in hand, we will have set up a nice set of abstractions for ourselves; and, if we write our `flood_fill` function using those functions exclusively (rather than directly accessing the underlying image representation), then our same `flood_fill` code can be used for either representation without any changes!

As we look at more complex and interesting programs, it is increasingly important to create helper functions like this, as part of structuring our code for easier understanding and easier change.

In the box below, fill in the definitions of these functions, assuming that $(x = 0, y = 0)$ is in the upper-left corner of the image, that x increases as we move to the right, and that y increases as we move down. You may assume that the given images are all in the format from Labs 1 and 2.

```
def get_width(image):
    pass  # replace with your code

def get_height(image):
    pass  # replace with your code

def get_pixel(image, row, col):
    pass  # replace with your code

def set_pixel(image, row, col, color):
    pass  # replace with your code

```

3.3) High-Level Strategy

With those pieces in hand, we can make our first attempt at a high-level plan. Before reading on, it's worth taking a bit of time to think for yourself about this problem. Can you come up with a high-level strategy for solving this problem? We'll discuss one such approach below.

Ultimately, we want to color in all of the pixels in one contiguous region. But, when we first start, we only know the first pixel that we are interested in. So, we will need an approach that will allow us to "discover"

new locations that need to be colored in, as we are coloring in those that we already know about.

Here is one possible high-level strategy, which we will use as a starting point for our conversation in this reading:

- Store the original color of the starting location in the image (let's call it C_0).
- Keep a list L of all locations we need to color in, starting with only the location that we started at.
- While we still have locations to color in (while L is nonempty), repeat the following:
  - Pick a location from L, remove it from L, and color it in with our new color.
  - Add that location's neighbors to L, so long as they have color C_0.

We put a single pixel in L as a starting point and then grow from there. L keeps track of all the pixels that we know about so far but that we haven't colored yet.

On each iteration of the algorithm, we're going to pull one pixel location out of L, color it in, and then add all four of its neighbors back into L. Then, for each of those, we'll pull them off L eventually, recolor them, and add *their* neighbors back to L, and so on. But, if we ever reach a neighbor that is a different color than the original starting pixel, we won't add that neighbor to L because we shouldn't color it. This process continues until L is empty.

The hope is that, eventually, we will add every location within the region that we first clicked on to our list of locations to color in, and we will eventually work our way through the whole list of locations; so, in this way, we hope eventually to color in all of the right spots.

This strategy of gradually accumulating things that we need to act on and then considering them one-by-one is a common approach to solving a wide variety of problems, as we'll see over the next couple of weeks of labs and readings in 6.101. In this kind of setup, we'll refer to the list L as an *agenda* (sometimes also called a *work queue*).

> **Check Yourself:**
>
> Think a bit about this plan. Will this work as expected? Do you see any potential issues with it? Are there any details we've left out of the plan above that might affect its operation?
>
> If you're having trouble understanding the high-level idea, that's OK; it's somewhat complex! In that case, we encourage you to ask for help/clarification, either in-person or via 6.101-help.

3.4) A First Attempt at Code

For now, there is no need to try to write out code for this part for yourself, but you are welcome to give it a shot if you would like!

When you're ready, take a look at the code below, which is an attempt to implement the strategy described above, and answer the question that follows it.

It's also worth noting that this code makes use of some interesting features of Python that we expect some of you may not have seen before. These new features are called out in the code and will be explained later in this reading, but, if something in the code below doesn't make sense and it isn't explained later, please feel free to ask (either in-person or via 6.101-help)!

Show/Hide

```
def flood_fill(image, location, new_color):
    # store the original color that we clicked on
    # (recall that "*location" is equivalent to
"location[0],location[1]")
    original_color = get_pixel(image, *location)

    # helper function to get neighboring locations of a given
row, col coordinate
    def get_neighbors(cell):
        row, col = cell
        return [(row+1, col), (row-1, col), (row, col+1), (row,
col-1)]

    # all of the cells we need to color in
    to_color = []

    # while there are still things to color in
    # ("while to_color" is equivalent to "while len(to_color) !=
0")
    while to_color:
        # remove a single (row, col) tuple from to_color and call
it this_cell
        this_cell = to_color.pop(0)

        # replace that spot with the given color
        set_pixel(image, *this_cell, new_color)

        # add each neighbor to to_color
        for neighbor in get_neighbors(this_cell):
            to_color.append(neighor)
```

Check Yourself:

How well does this code match the plan described above? Is anything missing?

Which of the following most accurately describes the code as written above?

- It contains a Python syntax error.
- It will raise an exception when it runs.
- It will not raise an exception but it will color in too many spots.
- It will not raise an exception but it will color in too few spots.
- It will work as expected.

4) Debugging

As a running example for today, let's consider the following image, which is included in the zip file above as `flood_input.png`:

Either add something to `flood_fill.py` or use your code from Lab 2 to determine the height and the width of this image. What are they? Enter your answer below, as a tuple of integers `(height, width)`:

But, before we get too lost in the sheer beauty of this image, let's remember that our goal for this reading is not just art appreciation but implementing flood fill. As such, we'll use this image to test our flood-fill

implementation as we work.

Let's start by using the code from above to change the color of the unsettling humanoid creature on the left. We'll do this by calling `flood_fill` with a location in the center of that creature.

> Here's how you call `flood_fill` on your own computer:
>
> 1. Run `python flood_fill.py`.
> 2. In the image that pops up, click on any pixel, and it will call `flood_fill` with that location.
> 3. The `new_color` used for flood fill is purple by default, but you can switch to using another color by typing a letter on your keyboard: `r` for red, `w` for white, `k` for black, `g` for green, `b` for blue, `c` for cyan, `y` for yellow, `p` for purple, `o` for orange, `n` for brown, `e` for grey.

So what happens when you run the code and click on a location?

Oops. In fact, the code does nothing. This behavior may be a little bit surprising! Let's try debugging it. Along the way, we will be seeing some common techniques for debugging and testing your code.

4.1) High-level Strategy

Here's the general strategy to pursue whenever you have a bug:

1. **Locate**: find where the bug is happening.
2. **Understand**: understand the cause of the bug.
3. **Fix** the bug.
4. **Verify** that the fix worked.
5. **Learn the bug's lesson**. Every bug is a learning opportunity in disguise! What can we learn from this one, and how can we avoid making similar mistakes in the future?

It's very important to avoid jumping straight to step 3. Making random changes to your code hoping to fix it is not a strategy for successful programming, nor an efficient way to fix the issue! It's a recipe for creating a complex and confusing spaghetti of code, or for "band-aids" that address the immediate symptom without fixing the underlying cause (which will likely rear its ugly head again somewhere down the line). Locate and understand the bug first, then plan how to fix it and execute your plan. Sometimes the fix is a simple change, but sometimes it requires rethinking your approach to the problem.

4.2) Print Debugging

The most common debugging technique -- available in virtually every programming environment you might use -- is printing. Printing is useful in a broad variety of different scenarios, but, for now, let's see how to use this to ask the question: where is this bug? Is our `flood_fill` code being run at all? Let's put in some print statements, just to decide: is `flood_fill` being called or not?

Check Yourself:

Where should we put print statements here to answer the question of whether the function is being called or not?

Show/Hide Our Approach

We'll take the following approach (the four calls we've added to `print` are highlighted in yellow).

Show/Hide Line Numbers

```
1  def flood_fill(image, location, new_color):
2      print('start of function')
3
4      # store the original color that we clicked on
5      # (recall that "*location" is equivalent to
   "location[0],location[1]")
6      original_color = get_pixel(image, *location)
7
8      # helper function to get neighboring locations of a
   given row, col coordinate
9      def get_neighbors(cell):
10         row, col = cell
11         return [(row+1, col), (row-1, col), (row,
   col+1), (row, col-1)]
12
13     # all of the cells we need to color in
14     to_color = []
15
16     print('before loop')
17
18     # while there are still things to color in
19     # ("while to_color" is equivalent to "while
   len(to_color) != 0")
20     while to_color:
21         # remove a single (row, col) tuple from to_color
   and call it this_cell
22         this_cell = to_color.pop(0)
23         print('inside loop', this_cell)
24
25         # replace that spot with the given color
26         set_pixel(image, *this_cell, new_color)
27
```

```
28          # add each neighbor to to_color
29          for neighbor in get_neighbors(this_cell):
30              to_color.append(neighor)
31
32      print('after loop')
```

Why did we choose those places for our print statements? Well, to be honest, we're already starting to think a little bit ahead. The first print statement (which prints `"start of function"`) is the one that really answers the question that we originally posed. Since it's the very first thing at the top of the function, that value will be printed if we call `flood_fill` at all.

The other print statements are actually going a little bit deeper; we've added those so that we can get a sense of how the process we planned out above evolves once the function is called.

When running the code with the print statements added, here's what we see printed to the screen when we click the middle of the creature in the image:

```
start of function
before loop
after loop
```

This output answers our original question: yes, the `flood_fill` function is running. But, it also tells us something else. Notice that we don't see "inside loop" anywhere in our output! What does this mean about how our program was running?

4.2.1) Aside: Truthiness

While we're thinking about that, it's worth bringing up another feature of the code: the condition of the `while` loop probably looks a little strange. Why is it just `while to_color:`? Why not some kind of Boolean expression? It turns out that Python, like many languages, allows types other than Boolean to appear in places where a Boolean is expected, like the condition of a `while` or `if` statement. For these other types, Python uses the object's "truthiness" (yes, that's the real term people use) to decide whether to take the given action. Each object in Python, regardless of type, is either "truthy" or "falsy." Which objects are truthy and falsy depends on the type we're dealing with, but here are a few examples:

Type	Falsy Values	Truthy Values
`bool`	`False`	`True`
`list`	`[]` (empty list)	all other lists

`tuple`	`()` (empty tuple)	all other tuples
`str`	`""` (empty string)	all other strings
`int`	`0`	all other integers

In general, the convention in Python is that values that are empty in some sense are falsy, and all other values are truthy.

4.2.2) Back to the Code!

So, what does this mean for the code we were just considering? We were looking at a looping structure like the following:

```
while to_color:
    # ...
```

So, now, we can see why we never entered the loop: the agenda `to_color` was initialized to an empty list, so `while to_color` found that it was falsy and immediately skipped the loop.

It's worth mentioning that we could have written this loop differently. Given Python's notion of truthiness, the structure above is equivalent to either of the following forms:

```
while to_color != []:
    # ...
```

```
while len(to_color) > 0:
    # ...
```

While the first form is nice for its compactness, either of the other two forms is arguably clearer in its intentions; so, stylistically, it may make sense to write out the slightly longer (but more explicit) form.

Regardless of how we write that looping condition, we can think about the bug here by recalling our plan: we were supposed to start with our original pixel location in the agenda and then loop until we run out of locations in the agenda.

Answer the following question about this code and then view the related answer/explanation to get a sense of how to proceed from here.

Given our plan from earlier, which single line should we change in the code above to fix this bug (that we never enter the loop)? Enter the Python code that should be changed:

Great! Our first bug located, understood, and hopefully fixed. Now, let's carry on and continue testing, by once again trying to color in the character in our testing image.

4.3) Interpreting Errors

When we run the code again with this change, here's what we get this time:

```
start of function
before loop
inside loop (40, 60)

Traceback (most recent call last)
  File "flood_fill.py", line 29, in <module>
    flood_fill(image, (event.pos[1] // SCALE, event.pos[0] // SCALE)
cur_color)
  File "flood_fill.py", line 29, in flood_fill
    to_color.append(neighor)
                    ^^^^^^^
NameError: name 'neighor' is not defined. Did you mean: 'neighbor'?
```

Oh no, it's an error! Unexpected things (like exceptions) happen all the time, even to experienced programmers. But, even after you've been programming for a long time, errors like this can be frustrating, especially after putting a lot of work into the code that caused the error.

So, how should we respond when we encounter an error like this? Well, firstly, it's OK to take a moment to feel that frustration; it's normal. But, let's not let that frustration affect us too much; let's take a deep breath (maybe take a short walk or have a cup of tea) and turn toward understanding what went wrong. Our first step here should be to *read the error message*. It's tempting, when you're just getting started programming, to try to ignore the jargon and scary nonsense in the error and try to plow ahead without it.

But, it's much better to take the time to read the error message. Once you learn to interpret them (which does take some time and practice), these error messages turn out to contain a lot of useful information! The error tells us not only what happened, but it also tells us *where* in the code it happened -- that's step 1 of debugging, and an error message like this is giving it to us for free!

The bottom of the error tells us exactly which line failed: `to_color.append(neighor)`, and it even tells us why: it can't find the name `neighor`. And why not? Because, if we look closely, it turns out that we've made a typo: while we meant to type `neighbor` (referring to an adjacent cell), we accidentally left off the "b", ending with `neighor` instead.

At this point, it's worth noting that we've done both steps 1 and 2 in our bug-squashing process (namely, locating and understanding the bug) just by reading the error message carefully. As you encounter new messages, if you have trouble understanding what they are trying to tell you about your code, please feel free to use a web search containing the error message or to reach out to us for help.

Now that we've identified the error, we can go in and fix it by changing that:

```
to_color.append(neighbor)
```

4.4) "Printing" More Complex Things

Let's run it again:

```
start of function
before loop
... (many locations are printed)
inside loop (75, 74)
inside loop (73, 74)
inside loop (72, 75)
inside loop (73, 73)
inside loop (73, 74)
inside loop (71, 76)
inside loop (71, 76)
inside loop (71, 76)
inside loop (72, 75)
inside loop (75, 76)
... (many more locations are printed)
```

OK, we're printing a lot of stuff, so there is evidence that Python is doing something here; but, in terms of our output, still nothing seems to be happening, and we're never seeing a result. Maybe it's an infinite loop?

But, the printing isn't telling us a whole lot, so let's output something different. Let's "print" the displayed image as we go -- in other words, let's update the output image on screen every time we change a pixel. This is a bit advanced, but the general lesson is that it's often useful to find the equivalent of a print statement in the domain you're working in. Since we're using images, we want to put the image on the screen in order to help us debug it. Seeing the intermediate results of the image as we go will serve a similar purpose as printing text, and it may be easier for us to diagnose what's going on.

If you are following along on your own computer, you can enable this behavior by uncommenting the bottom two lines in the `set_pixel` function in `flood_fill.py`:

```
    # screen.blit(image, (0, 0))
    # pygame.display.flip()
```

Now, let's see what happens when we try to fill the cat's face instead, using grey. Feel free to try clicking the middle of the cat's face on your own computer and/or watching the video below, which shows the results:

Horrifying.

> If you are following along on your own computer, you may find that the image window refuses to close when you click on its X button, and you can't stop the program! This is because Python is so busy running `flood_fill` that it never gets around to responding to your mouse click on the window's close button.
>
> When this happens, you need to switch back to the terminal window where you started `python flood_fill` and use Control-C to stop the program there. That's a more reliable way to end a misbehaving Python program.

The good news is that, by showing the updated pixel colors to the screen as we're going, we can see that some progress is being made. But, there is a problem, in that our function is currently misbehaving and eating the poor cat's face; it looks like we're coloring *all* the pixels of the cat's face, not paying any attention to whether they match the original red color we clicked on or not. What are we missing? Before clicking to open the box below, think about this for a while (and, if it's helpful, feel free to try tweaking the code, adding more print statements, etc.). What is causing this issue, and what can we change in our code to fix it?

> Show/Hide Our Explanation
>
> Remember that our strategy was to stop when we reach a pixel that isn't the same color as the original pixel. We shouldn't add those pixels to the agenda. But, we forgot to include that check in our implementation. Let's add an `if` statement in the loop that looks at the neighbors (and

also change the comment to reflect the change!):

```
# add each neighbor to to_color, but only if it's the same color
as original_color
for neighbor in get_neighbors(this_cell):
    if get_pixel(image, *neighbor) == original_color:
        to_color.append(neighbor)
```

This actually looks like a good place to use a list comprehension! Here's one way to do it:

```
to_color += [
    neighbor
    for neighbor in get_neighbors(this_cell)
    if get_pixel(image, *neighbor) == original_color
]
```

Note that we are using `+=` here to *append* the list of neighbors to the end of our agenda.

4.5) Printing Just Before the Error

Now let's try filling the cat's face again:

Great! This looks right. If you're keeping count, we've now identified, localized, and fixed three bugs! 🎉🥳🎈 We can take a moment now to celebrate by filling in the cat's ears and body as well, so that we end up with a nice grey cat:

Woo-hoo! It seems to work on the cat. This is also a time to remind ourselves that this process (of repeatedly discovering, identifying, localizing, and fixing bugs) is a natural part of the programming process. While careful planning and good style can helps us avoid lots of bugs we might make otherwise, it's still very hard to get things right on the first try. Errors are still going to creep into the code of even the most experienced programmer, and we should do our best to embrace the act of debugging for what it is: an important part of the programming process, as well as an important part of the *learning* process.

Alright, onward! Let's celebrate further by coloring in this drab sky with a nice light blue color. Here, we click toward the right side of the sky:

Oh no! We've been making good progress, but at this point, the program crashes, showing us another error message, which will look something like the following:

```
Traceback (most recent call last):
  File "flood_fill.py", line 126, in <module>
    flood_fill(image, (event.pos[1] // SCALE, event.pos[0] // SCALE),
cur_color)
  File "flood_fill.py", line 29, in flood_fill
    if get_pixel(image, *neighbor) == original_color:
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "flood_fill.py", line 45, in get_pixel
```

```
    color = image.get_at((col * SCALE, row * SCALE))
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
IndexError: pixel index out of range
```

Oops. Another bug. But, this isn't the end of the world. We've prepared for this, and we know what to do: we take a moment to feel our frustration, and then we take a deep breath. And, in fact, we maybe ought to take a moment to celebrate too, because every bug is a learning opportunity in disguise, and, when we get an error message, we're already much of the way towards finding, understanding, and fixing it.

This error is a little bit more complex than the one we saw earlier, but let's try to break it down. We're getting the message `"IndexError: pixel index out of range"`, and the first place that it happens (look at the bottom of the traceback) is in `get_pixel`, which is a function we probably wrote a long time ago (in last week's lab, in fact). We've been using that function a lot -- is that really where the bug is?

But, let's look at how we got there. The error message includes a traceback of the calls that led down to the point where we had the error. These are actually the active frames on the stack (the same stack we draw in an environment diagram), so a traceback is often called a *stack trace*.

If we look just above `get_pixel` in the traceback, we see that `get_pixel` is being called by the code we just wrote in `flood_fill`:

```
if get_pixel(image, *neighbor) == original_color:
```

While it's not impossible that there is an issue with `get_pixel` (and we should keep in mind that that's a place to look if we can't figure things out here), code that we just wrote is a much more likely location for the actual bug. So, we have a possible location; but, before we go diving into the code, let's see if we can understand more about the cause of the bug. What does the image look like just before the error happens?

Ah! The right corner of that blue diamond, representing the pixels we've painted so far, has just touched the right edge of the image. It's happening when we try to go off the edge of the image, hence the message "pixel index out of range".

So, we've localized the error, and we have a hypothesis as to what is going wrong (as of right now, we think our function might be trying to color in pixels outside the bounds of the image). But, let's try to validate that hypothesis a bit more before we go diving into the code.

And, to help us out, here's our old friend, `print`. Let's put a print *before* the line where the error happened in this code, showing every variable that might be relevant to causing the bug:

```
# add each neighbor to to_color, but only if it's the same color as
original_color
for neighbor in get_neighbors(this_cell):
    print(neighbor, original_color)
    if get_pixel(image, *neighbor) == original_color:
        to_color.append(neighbor)
```

This is a helpful tactic for exploring an error: look where the error happened and try to print relevant variables just before it. This strategy is great because, since the exception stops Python, the very last thing that gets printed will be the thing that caused the error:

```
... (lots of printing)
(44, 97) (255, 255, 255)
inside loop (45, 99)
(46, 99) (255, 255, 255)
(44, 99) (255, 255, 255)
(45, 100) (255, 255, 255)
Traceback (most recent call last):
  File "flood_fill.py", line 127, in <module>
    flood_fill(image, (event.pos[1] // SCALE, event.pos[0] // SCALE),
cur_color)
  File "flood_fill.py", line 30, in flood_fill
    if get_pixel(image, *neighbor) == original_color:
       ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "flood_fill.py", line 46, in get_pixel
    color = image.get_at((col * SCALE, row * SCALE))
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
IndexError: pixel index out of range
```

Aha! It looks like we did indeed reach a point where `neighbor` was `(45, 100)`, which is outside the bounds of our 100x100 image.

4.5.1) Aside: "F-Strings"

Sometimes, when you are printing multiple variables, it can be hard to tell them apart. So, here is one more useful Python idiom: you can use format strings (also called "f-strings") to automatically add a label for

each variable. Here's how.

First, a format string is a string prefixed by the letter f, which can have expressions embedded in it. For example, `f'{1+2}'` evaluates to the string `'3'`, and (in this context) `f'{neighbor}'` evaluates to the string representation of the variable `neighbor`, say, `'(100, 45)'`.

The second part of the trick is to put `=` just before the `}`, which precedes the value of the expression with the original expression itself. So `f'{1+2=}'` evaluates to `'1+2=3'`, and `f'{neighbor=}'` to `'neighbor=(100, 45)'`.

So if you change your print statement to:

```
    print(f'{neighbor=} {original_color=}')
```

then your output will be labeled with the variable names too, which is super useful for helping us keep track of everything we're printing.

```
... (lots of printing)
neighbor=(45, 99) original_color=(255, 255, 255)
neighbor=(43, 99) original_color=(255, 255, 255)
neighbor=(44, 100) original_color=(255, 255, 255)

Traceback ...
IndexError: pixel index out of range
```

4.5.2) Back to the Code!

OK, this further validates our hypothesis from above, so, at this point, we can be reasonably confident that we've located and understood the bug: as it currently stands, our code is trying to color in pixels that are outside the bounds of the original image.

Now that we have found and understood the bug, we can make a plan for fixing it, and, eventually, we can write some code. Take some time here to think about some questions:

- What do we need to change about the code's behavior to fix this bug?
- What are all of the places we could make the necessary change(s)?
- Are there pros and cons to the different approaches we could take here?

Show/Hide Our Explanation

We've already identified the core issue: we shouldn't be coloring in pixels that are outside the bounds of the image! But, we have some options for how to fix it! We should think through them a little bit before we start writing any code. Among other options, maybe we could:

1. Change `set_pixel` so that, if it gets an out-of-bounds location, it returns immediately rather than trying to color it in.
2. We could add a conditional just above `set_pixel` so that we only ever attempt that call if `location` is in-bounds.
3. We could change our conditional just above `to_color.append(neighbor)` so that we only add neighbors that are in-bounds to the agenda.
4. We could change our notion of a "neighbor" by modifying `get_neighbors` such that it only includes valid locations in its output.

What are some of the pros and cons of these approaches? Potential pitfalls?

Show/Hide Comparison

Option **1** is a tempting change to make because it's very local to the issue we noticed. And it's likely that this would help us make the error message go away. But, it has a downside: if we made this change, we'd still be considering those out-of-bounds pixels in our agenda and adding their neighbors in, so we'd spend a lot of time still thinking about pixels that are outside our image!

Option **2** can have the same downside if we're not careful. If we were to take this approach, it would be safest to indent *the whole rest of the function*, including the piece that considers adding neighbors, which prevents considering *too many* out-of-bounds pixels (since we won't add the neighbors of any out-of-bounds pixels), but this approach still waits until we've pulled the out-of-bounds pixel out of our agenda, rather than not adding it to the agenda in the first place.

Options **3** and **4** both do a better job of addressing the issue early: both avoid ever adding any out-of-bounds locations to the agenda. Both of these are plausible solutions, but there are still a few tradeoffs. Option **4**, redefining the notion of a neighbor, lets us keep the core code for flood fill relatively clean (without a lot of extra conditions), so we're going to take that approach here. But, it might be good additional practice to try turning both of these approaches into code, so you can get a feel for your own personal preferences between the two.

Show/Hide Code

Here is the code that we came up with for changing `get_neighbors` so that it only includes in-bounds locations as valid neighbors. Here, we start by finding all possible neighbors by looking in all four directions from the given location, and then we use a list comprehension to filter out the locations that are out of bounds.

```
    def get_neighbors(cell):
        row, col = cell
```

```
        potential_neighbors = [(row+1, col), (row-1,
col),
                               (row, col+1), (row,
col-1)]
        return [
            (nr, nc)
            for nr, nc in potential_neighbors
            if 0 <= nr < get_height(image) and 0 <=
nc < get_width(image)
        ]
```

4.6) Avoiding Extra Work

Having made this change, we can try running the code again and clicking near the edge of the image to see whether we have fixed our issue. Feel free to try this on your own computer, or here is a video of our code working through this example:

OK, we've made some progress! Now, when we click on the sky, it's no longer crashing as soon as it touches the boundary. But, it also seems to slow down as it goes, until finally it doesn't seem to be making any progress at all, and the sky is nowhere near being filled! What is going on?

It's a little bit hard to tell just from the output here, but there is some additional information in our print statements, which are still going to the terminal and showing the location of every pixel we're coloring in. Let's take a careful look:

```
...
(36, 98)
(37, 99)
```

```
(37, 99)
(36, 98)
(36, 98)
(37, 99)
...
```

Wait, there's a lot of repetition here! And we can also see this repetition in the following visualization, which briefly flashes each pixel in red as we're coloring it (notice how the pixels on the edges of the colored-in region, whose color we've already changed, are being repeatedly colored in):

So, now, we've seen this in two different ways, but this is perhaps still a confusing issue to encounter. Why is our code as written trying to color in the same spot repeatedly? When we come back to a pixel we've already colored in, shouldn't its color now be different from the original pixel's color, and, thus, shouldn't we avoid coloring it in again through the following check?

```
for neighbor in get_neighbors(this_cell):
    if get_pixel(image, *neighbor) == original_color:
        to_color.append(neighbor)
```

Take a moment to think about this and see if you can figure out why the code end up coloring in the same cells multiple times, before continuing on.

Show/Hide Explanation

The complication here has to do with the fact that we end up adding locations to the agenda multiple times before we color them in.

Let's look more closely at what we're doing.

We start with the location we clicked on the agenda.

Then, we pull that location off the agenda, color it in, and add its neighbors to the agenda. In so doing, we've now added each location that is one step away from the starting location to the

agenda. We're working through the agenda in the same order we added things to the agenda, so we'll color in each of those pixels and add their neighbors.

But, here is where we can first notice the issue: some of those locations we've just added to the agenda were neighbors of multiple of the locations we've already colored in! For example, the pixel to the upper-right of the original location is a neighbor of both the pixel to the right of the original location and the pixel above the original location, so it will be added to the agenda twice before coloring it in!

We end up adding pixels within 2 steps of the original pixel to the agenda twice, before we've been able to color them at all. And those within 3 steps, more than twice! And, in fact, as we get farther and farther away from the original pixel, there are more and more paths through the pixel grid that can reach the pixel we're adding to the agenda, and we'll add it to the agenda once for each such path. The number of paths grows explosively, so our agenda does too. This program won't just be slow, and we can't just go get a sandwich and come back and hope it will be done; it will take longer than the lifetime of the universe for this to finish, even on this small 100x100 image.

Fixing this efficiency issue is not a simple one-line change. We need to change our strategy substantially. In particular, let's add a structure that will help us avoid adding locations to the agenda more than once. Our strategy will be to introduce a new variable to keep track of *every location we've ever added to the agenda* (we'll call this `visited`), and we'll adjust our code such that we only add pixels to the agenda if they have never been added to the agenda.

To accomplish this, we'll make some changes to the main loop in our code (highlighted in yellow below):

Show/Hide Line Numbers

```
def flood_fill(image, location, new_color):
    original_color = get_pixel(image, *location)

    def get_neighbors(cell):
        row, col = cell
        potential_neighbors = [(row+1, col), (row-1, col), (row, col+1),
(row, col-1)]
        return [
            (nr, nc)
            for nr, nc in potential_neighbors
            if 0 <= nr < get_height(image) and 0 <= nc < get_width(image)
        ]

    to_color = [location]  # agenda: all of the cells we need to color in
    visited = [location]  # all pixels ever added to the agenda

    while to_color:
```

```
        this_cell = to_color.pop(0)
        set_pixel(image, *this_cell, new_color)
        for neighbor in get_neighbors(this_cell):
            if (neighbor not in visited
                    and get_pixel(image, *neighbor) == original_color):
                to_color.append(neighbor)
```

Answer the following question about the code, paying careful attention to the pieces we changed.

With the code written exactly as above, how does its speed compare to that of the original function? The new code is around 10x faster

4.7) Timing the Program

Aha! Now, with the change suggested in the explanation to the question above, we can color the sky, and the other pieces as well, to make the picture even more beautiful (if you can believe that). You can try this out on code on your machine, and/or you can see it play out in the video below:

We've come a long way since we started. We've fixed a lot of bugs along the way, and our code is finally capable of accomplishing its goal!

But, something here is still perhaps a little bit bothersome: filling in this image took quite a long time, despite its being a small image! The details will depend on the machine that this code is running on, but, regardless, this code takes more time than we might expect. In a regular image-editing program, using a paint-bucket tool on a 100x100 image would be instantaneous, but, here, it took a noticeable amount of time!

One subtle thing to note here is that our code is actually being slowed down substantially by our debugging statements (both the `print`ing we're doing and, more noticeably, the fact that we are updating the display of the image after each pixel we change). Let's remove our print statements and also comment out the two lines in `set_pixel` responsible for drawing the image after every change (the two that we

uncommented near the start of the reading). Instead, we'll just display the results of the whole operation once it's done. In general, this is a good idea for your own programs as well. Print statements are great for debugging, but, once we've localized, understood, and squashed the bug for which we added those statements, it's a good idea to remove them.

After doing that, it seems much faster. But, how fast is it, really? Let's measure how long it's taking, using a function from Python's standard library, `time.time()`. This function returns a float value representing the number of seconds since January 1, 1970, at midnight UTC. That exact detail doesn't matter too much for us, but what does matter is that it is a stable measure of time in units of seconds. So, if we call that function twice and compute the difference of the respective results, it will tell us the number of seconds that elapsed between the two calls.

So, let's print the difference between the clock before and the clock after working through the flood-fill process:

Show/Hide Line Numbers

```
import time

def flood_fill(image, location, new_color):
    original_color = get_pixel(image, *location)

    def get_neighbors(cell):
        row, col = cell
        potential_neighbors = [(row+1, col), (row-1, col), (row, col+1), (row, col-1)]
        return [
            (nr, nc)
            for nr, nc in potential_neighbors
            if 0 <= nr < get_height(image) and 0 <= nc < get_width(image)
        ]

    start = time.time()

    to_color = [location]  # agenda: all of the cells we need to color in
    visited = [location]  # all pixels ever added to the agenda

    while to_color:
        this_cell = to_color.pop(0)
        set_pixel(image, *this_cell, new_color)
        for neighbor in get_neighbors(this_cell):
            if (neighbor not in visited
                    and get_pixel(image, *neighbor) == original_color):
```

```
26                 to_color.append(neighbor)
27                 visited.append(neighbor)
28
29     print(f'this took {time.time() - start} seconds')
```

Now let's fill in the sky again. The results will vary quite a bit from machine to machine, but, even on a relatively modern computer, you're likely to see something like the following:

```
this took 0.5847759246826172 seconds
```

That's definitely faster than before. But, more than half a second for this operation? That still seems incredibly slow for a modern computer filling a tiny image.

So, while we appear to have a *working* program now, there's still room for improvement! Let's see what we can do to speed things up here.

4.8) Using the Right Data Structure

Let's start by making a small change to our code. Feel free to follow along here as well. Let's make the following two changes:

- Replace `visited = [location]` with `visited = {location}`, using squiggly brackets instead of round brackets.
- Replace `visited.append(neighbor)` with `visited.add(neighbor)`.

Running the code again after this change, you're likely to see a *substantial* improvement in terms of time (maybe even as big as 10x!):

```
this took 0.07046723365783691 seconds
```

What is going on here? How can such a small change to the code make such a big difference in terms of runtime? The short answer is that those two small changes mean that we are now using a Python `set` object, rather than a list, to store the locations we've already colored in. We'll spend a lot more time with sets in an upcoming recitation; but, for now, we'll just briefly introduce sets and talk a tiny bit about the key behavior that caused this speedup.

Like lists, sets are *containers*, in that a set represents a collection of objects; and, like lists, sets are *mutable* (we can modify them after creation time by adding or removing elements). But, sets are different from lists in several ways:

- Sets can only contain certain kinds of elements.
- The elements in a set do not have an order.
- Elements can't appear more than once in the same set.

- Membership tests take roughly the same amount of time, regardless of how big the set is or what element we're looking for.

In our original represention (using a list to represent `visited`), the real culprit for the slowdown was the check for `neighbor not in visited`. When doing a "membership test" (checking to see whether an element is in the list, using the `in` keyword), Python has to scan the full list one element at a time, checking each one against the element we're looking for. And, as we visit more pixel locations in the image, `visited` gets longer and longer, so this check will get slower and slower!

The substantial speedup we see from a set here is because that membership check (`neighbor not in visited`) runs in *constant time*, meaning that it takes roughly the same amount of time, even as the set grows. For our flood-fill program, this means that, even as our `visited` set grows in size, the check for whether `neighbor` is in the `visited` set remains fast.

One word of caution before we move on: sets are absolutely the right choice of data structure in this particular case, but it can be tempting to draw too strong a conclusion from this. It's tempting to say that "sets are faster than lists" and then replace all your lists with sets! But, that won't always work. If you need your data structure to maintain the order of its elements, or if you need the ability to hold multiple equivalent objects, then a set won't work in those cases. And, for other cases where a set would work, a list may still be the right choice; it all depends on the specifics of the kinds of operations we're performing on those structures. In general, the idea is that, when we're writing programs, we have a lot of decisions to make about what is stored and how we represent it; and we want to be careful to choose the right tool for the job at hand.

5) Finding a Path Through a Maze

At this point, we have a working and relatively efficient program for performing a flood fill! Here is the code where we're leaving it off for now:

Show/Hide Line Numbers

```
1  def flood_fill(image, location, new_color):
2      # store the original color that we clicked on
3      # (recall that "*location" is equivalent to
   "location[0],location[1]")
4      original_color = get_pixel(image, *location)
5
6      # helper function to get neighboring locations of a given row, col
   coordinate
7      # (only returning in-bounds neighbors)
8      def get_neighbors(cell):
9          row, col = cell
10         potential_neighbors = [(row+1, col), (row-1, col), (row,
   col+1), (row, col-1)]
```

```
        return [
            (nr, nc)
            for nr, nc in potential_neighbors
            if 0 <= nr < get_height(image) and 0 <= nc <
get_width(image)
        ]

    to_color = [location]  # agenda: all of the cells we need to color
in
    visited = {location}  # visited set: all pixels ever added to the
agenda

    # while there are still things to color in
    # ("while to_color" is equivalent to "while len(to_color) != 0")
    while to_color:
        # remove a single (row, col) tuple from to_color and call it
this_cell
        this_cell = to_color.pop(0)

        # replace that spot with the given color
        set_pixel(image, *this_cell, new_color)

        # add each neighbor to to_color so long as it matches the color
we
        # originally clicked
        for neighbor in get_neighbors(this_cell):
            if (neighbor not in visited
                    and get_pixel(image, *neighbor) == original_color):
                to_color.append(neighbor)
                visited.add(neighbor)
```

Let's take a look at applying this to a different image, though, a maze. The following video is slowed down so that we can see the way in which our flood-fill program works its way through a maze. Here, we start from the upper left corner, and the end of the maze is the spot in the lower right.

There is something kind of cool about this. The flood fill process "explores" the maze, so to speak, and it eventually fills the whole thing. Regardless of where the goal is, the flood-fill process will also eventually reach it!

As a last little exploration for this reading, let's think about what we could change about our program in order to make it *solve* a maze (returning us a list of locations in the solution) rather than just coloring in the maze. Fortunately, it doesn't take a big change to make this happen; it turns out that our agenda-based approach will work here as well.

Instead of keeping track of single locations, we are going to keep track of *paths* (tuples of connected pixel locations), each of which will start at the start location and end at some point that we've reached in the maze.

For each step of the algorithm, we'll take a path off the agenda and try to extend it with one more point neighboring its last point, putting those new paths back on the agenda. Once we take a path off the agenda and find that it ends at the end point of the maze, we're done!

We'll still use a `visited` set here, but its usage here is a little bit nuanced. In the flood-fill example, our `visited` set kept track of *all pixels that we knew we wanted to color in*. Here, we're instead going to use it to keep track of *all locations to which we've already found a path*, i.e., all locations that have ever been the *last location* in a path.

Many pieces of our flood-fill code will now work fine, but we probably want to give this new function a new name and change a few things about its function signature (we'll change some names, but, also, since we're not coloring these things in as we go, there's no need to specify a `new_color` argument); but the goal in our mazes will be represented by a particular color. Let's also rename our agenda variable from `to_color` to `possible_paths` and make it a list of paths, where each path is a tuple of locations.

So, maybe the start of our function now looks like:

Show/Hide Line Numbers

```
1 def find_path(image, start_location, goal_color):
2     safe_color = get_pixel(image, *start_location)
3
```

```
    # helper function to get neighboring locations of a given row, col coordinate
    # (only returning in-bounds neighbors)
    def get_neighbors(cell):
        row, col = cell
        potential_neighbors = [(row+1, col), (row-1, col), (row, col+1), (row, col-1)]
        return [
            (nr, nc)
            for nr, nc in potential_neighbors
            if 0 <= nr < get_height(image) and 0 <= nc < get_width(image)
        ]

    possible_paths = [(start_location,)]  # agenda: paths we know about but have not yet tried to extend
```

And, now, our looping structure, in pseudocode, will look like:

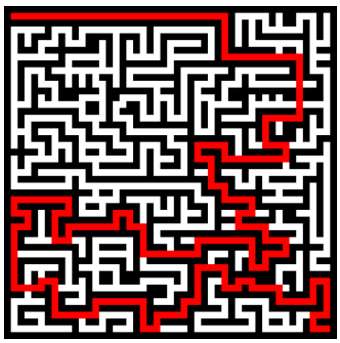
- Repeat the following:
    - Remove one path from the agenda.
    - For each of the neighbors of the *last* location in the path:
        - If it is in the visited set, skip it and move on to the next neighbor.
        - Otherwise, if it represents the goal condition (if that spot's color matches the `goal_color`), then return the path (including the goal location at the end).
        - Otherwise, if it is the same color as where we started, add it to the visited set and add the associated path (containing that neighbor) to the agenda.

  until the agenda is empty (search failed).

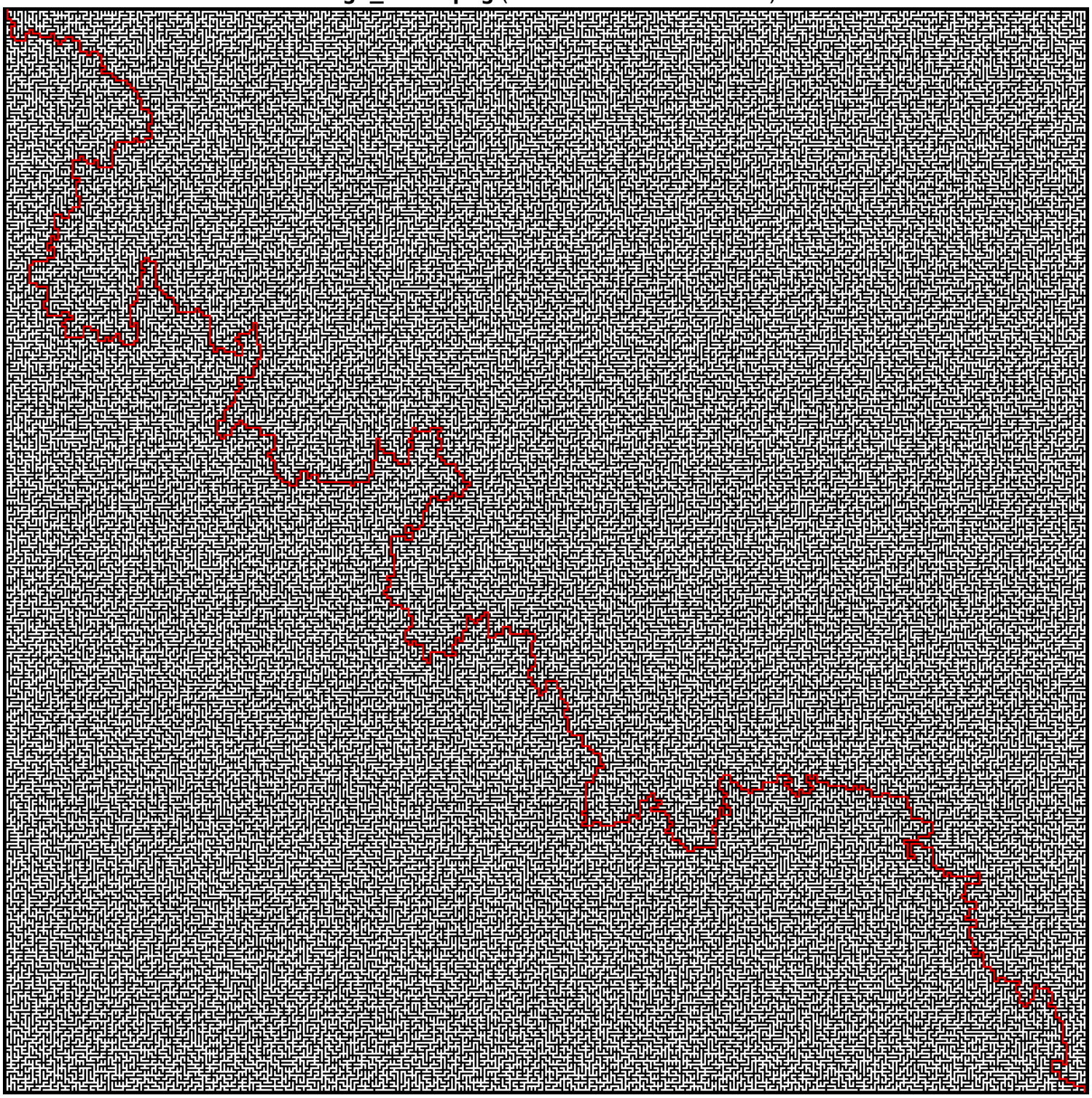
Note that this is a minor change from our flood-fill algorithm. The main difference is that we're keeping track of not only the locations we are considering but also *how we got there*.

We can also see this at work by applying it to different mazes (we've included a few in the code distribution from today). Here are two examples solved by code following this plan. These correspond to `large_maze.png` and `huge_maze.png` in the code distribution (feel free to try implementing these examples for yourself!), and in each, the start is the upper-left-most white pixel and the goal is the lower-right-most white pixel.

`large_maze.png` (293 locations in solution)



huge_maze.png (2555 locations in solution)



Here's how you can change `flood_fill.py` so that it calls `find_path` on mazes of your choice:

- find the line `IMAGE = "flood_input.png"` and change it to use the filename of a maze image, such as `"large_maze.png"` or `"huge_maze.png"`
- near the end of the file, change the call to `flood_fill` so that it calls `find_path` instead

6) Summary

As usual, we've covered a **lot** of ground in this reading along a number of different dimensions, so we'll try to recap some of the main points here.

Our main focus for this reading centered around techniques for debugging and iteratively improving a small program: following the locate/understand/fix process and using print statements and error messages to guide the locating and understanding. Although our examples here all centered around one problem, we encourage you to try to apply this same process to your own programs when unexpected things happen.

We also introduced the specific "flood-fill" algorithm and extended it to solve mazes. Both of these examples are special cases of a broadly applicable general approach called *graph search*, which is useful for solving a wide variety of real-world problems and which we will formalize more in the coming labs, recitations, and readings.

Along the way, we also introduced a few interesting features of Python, which we encourage you to experiment with as you're continuing with writing your own programs!