Pre-Lecture 1: Python Goodies, and Flood-fill Introduction

This week's lecture will be slightly different in character from last week's lecture, in that this week we will mostly be doing some live coding working on a single big exercise. As such, this set of readings will be a little bit different in terms of character, compared to last week's.

We'll use this set of readings to introduce a few interesting features of Python (some or all of which we may make use of during lecture), and then we'll also spend a bit of time introducing the problem we'll be trying to solve during this week's lecture so that we can dive right in during the lecture.

Two Cool/Useful Python Features

Before we dive in to this week's lecture 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 lecture and recitation 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.

Python Built-in: `zip`

`zip` is a nice Python built-in that allows us to easily find corresponding elements in two iterable objects. 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.

This is useful in the context of something like the following program, which subtracts corresponding elements in two lists:

```
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 6.009 lectures 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`).

Python Language Feature: List Comprehensions

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]
```

It's worth mentioning that it is never *necessary* to use this syntax, but it is a really convenient shorthand that we will use a lot in example code in 6.009 and that may be 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 i in x:
    if i < 0:
        out.append(-i)
```

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):

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`.

Check Yourself:

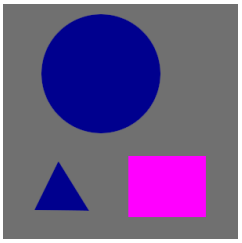
Can you use this list-comprehension idea to write an even more concise version of `subtract_lists` from above?

Show/Hide

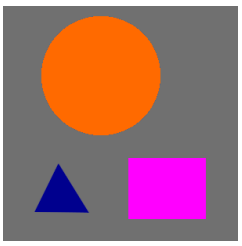
Now we'll spend a little bit of time introducing the problem we'll be working on in lecture this week.

Flood Fill

This week, we will spend most of the lecture time working on a single problem related to image processing, which we call "flood fill." This is a common operation in many image-editing programs and is sometimes referred to as a "paint-bucket" tool in that context. 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:

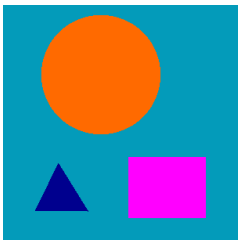


If we were to use the paint-bucket tool and click anywhere in the circle to replace it with an orange color, we would see the following as a result:



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 in lecture 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.  Returns None but mutates the original image to
    reflect the change.
```

```
    Parameters:
      * image: the image to operate on
      * location: an (x, y) tuple representing the starting location of the
                  flood-fill process
      * new_color: the replacement color
    """
    pass
```

Helper Functions

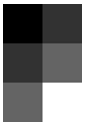
In lab 1, we will think about *image processing*, which involves manipulating the content in images. Before we can get to *manipulating* images, we first need a way to *represent* images in Python. While digital images can be represented in myriad ways, the most common has endured the test of time: a rectangular mosaic of *pixels* -- colored dots, which together make up the image. An image, therefore, can be defined by specifying a *width*, a *height*, and an array of *pixels*, each of which is a color value. This representation emerged from the early days of analog television and has survived many technology changes. While individual file formats employ different encodings, compression, and other tricks, the pixel-array representation remains central to most digital images.

To start, let's think about greyscale images alone. For this kind of image, each pixel's brightness is encoded as a single integer in the range `[0,255]` (1 byte could contain 256 different values), `0` being the deepest black, and `255` being the brightest white we can represent. The full range is shown below:

For today (and in lab 1), we'll represent an image using a Python dictionary with three keys:

- `width`: the width of the image (in pixels),
- `height`: the height of the image (in pixels), and
- `pixels`: a Python list of pixel brightnesses stored in row-major order (listing the top row left-to-right, then the next row, and so on)

For example, consider this 2×3 image (enlarged here for clarity):



This image could be encoded as the following instance:

```
i = {'height': 3, 'width': 2, 'pixels': [0, 50, 50, 100, 100, 255]}
```

Throughout the lecture (and in these notes, to start), we will make use of a few "helper" functions to make writing the code a little bit nicer. In particular, we will assume the existence of the following functions when writing our code:

- `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, x, y)`: return the color of the pixel at location (x, y) in the given image.
- `set_pixel(image, x, y, color)`: mutate the given image to replace the color at location (x, y) with the given color and return `None`.

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 lab 1.

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

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

def get_pixel(image, x, y):
    pass  # replace with your code

def set_pixel(image, x, y, color):
    pass  # replace with your code

```

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 lecture:

- Store the original color of the starting location in the image (lets 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 and color it in with our new color.
    - Add that location's neighbors to L, so long as they have color C_0.

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.

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 the forum.

A First Attempt at Code

For now, there is no need to try to turn this approach into code 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; so if something below doesn't make sense, please feel free to ask (either in-person or via the forum)!

Show/Hide

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.

This example will be our starting point for lecture, and we will spend a lot more time with it during that time. See you then!

Thanks!

Hopefully the notes above help to prepare you for Monday's lecture! Of course, if you have questions after working through the readings and/or exercises, please don't hesitate to ask!