Chapter 1 Image and code

CS101 Big Ideas

Foreshadowing: very briefly, here are a few big ideas that we'll see over the quarter. Some of these are intuitive, and some will be a surprise.

1. Algorithm

An "algorithm" is the **idea** or plan of how to compute something - e.g. the algorithm to produce a bluescreen image from two images. When you are programming, writing code, you are taking the algorithmic idea and translating it to code the computer understands.

2. Code vs. Data

Data is just the passive information, while "code" defines the active plan followed by the computer. A JPEG vs. an "app" on your phone. Both code and data both exist in the dry, mechanical domain in the computer.

3. Code is very mechanical

The computer runs whatever code it is given mechanically. That is not the same as thinking! Writing code in CS101, you'll get a real appreciation for this idea.

Followup ideas...

- Creativity and insight come from the programmer who writes the code.
- Garbage In Garbage Out - since the computer just produces mechanical transformations, putting in wrong data, you just get wrong outputs.
- Bug - A bug is when the programmer writes some code that does the wrong thing, not what they intended. The computer runs this wrong code happily, and again we just get wrong output.

4. Small code, Mass data

Often the code you write will be quite short. The computer can run the short piece or code over millions of billions of data items. This is a powerful combination - the code has some insight in it, and the computer is good at applying it massively.

5. Moore's Law

Computer hardware gets significantly cheaper and more powerful year after year. E.g. 4 years ago your phone holds 4 GB of data. Now for about the same cost it holds 16 GB (4x more). That's Moore's law.

Code-1 Introduction

CS101 explores the essential qualities of computers, how they work, what they can and cannot do, and requires not computer background at all. In this first section we'll look at the basic features of computers and get started playing with computer **code**.

Acknowledgements: thanks to Google for supporting my early research that has helped create this class. Thanks to Mark Guzdial who popularized the idea of using digital media to introduce computers.

Fundamental Equation of Computers

The fundamental equation of computers is:

Computer = Powerful + Stupid

- Powerful - look through masses of a data
  -Billions of "operations" per second
- Stupid
  -Operations are simple and mechanical
  -Nothing like "insight" or "thinking" (HAL 9000 video)
- Powerful + Stupid ... vividly experienced in our exercises
- Stupid, but very useful. How does that work?
- That's what CS101 is about
  -Visit this funny computer world, see how it works

-Understand what they can do, how made useful
-Not intimidated, computer is not some magic box
- Hidden agenda: open eyes for some, more computer science courses

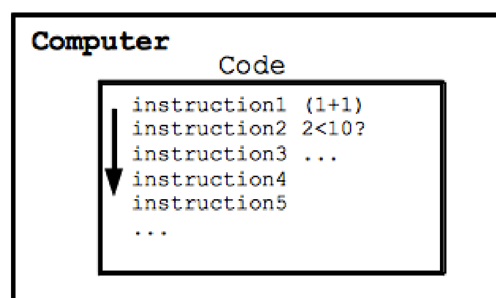
Computers are very powerful, looking through large amounts of data quickly. Computers can literally perform billions of operations per second. However, the individual "operations" that computers can perform are extremely simple and mechanical, nothing like a human thought or insight. A typical operation in the language of computers is adding two numbers together.

So although the computers are fast at what they do, the operations that they can do are extremely rigid, simple, and mechanical. Or put another way, computers are **not** like the HAL 9000 from the movie 2001: A Space Odyssey: HAL 9000 video.

If nothing else, you should not be intimidated by the computer as if it's some sort of brain. The computer is a mechanical tool which can do amazing things, but it requires a human to tell it what to do.

High Level - How Does a Computer Work?

- The computer follows a series of "code" instructions
- Each instruction is simple, mechanical
- e.g. add 2 numbers
- The computer "runs" a long series of instructions
- Purely mechanical

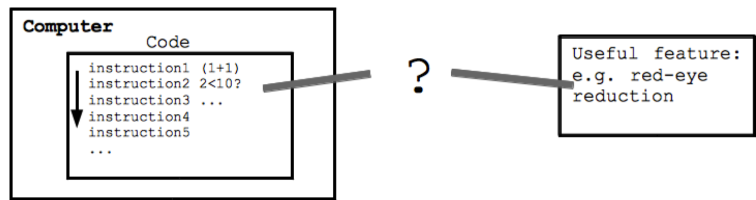


But So Many Useful Features

- Think of all the useful computer features (phone, camera)
-Email, instant messaging

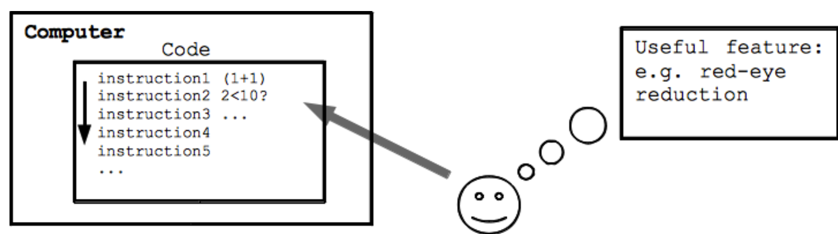
-MP3 audio
-Red-eye reduction

- If computers are so stupid... how are they so useful?
- What connects the two sides?



Programmers Make It Happen

- Programmers make it happen
- The programmer thinks up a useful feature
  -Creativity, insight about human needs, computers
- Programmers breaks down the steps, writing code for the computer
- Best features of both sides: inexpensive/fast of computer + creative insight of the programmer
- CS101 explorations: algorithms and code



Since the computer is totally mechanical and stupid -- how do they manage to do so many useful things? The gap between the computer and doing something useful is where the human programmer creates solutions. Programming is about a person using their insight about what would be useful and how it could be done, and breaking the steps down into code the computer can follow.

Code refers to the language the computer can understand. For these lectures, we'll write and run short snippets of code to understand what the essential qualities of computers, and especially their strengths and limitations.

Experimenting with code, the nature of computers will come through very clearly ... powerful in their own way, but with a limited, mechanical quality. IMHO, this mixed nature of computers is something everyone should understand, to use them well, to not be intimidated by them.

Before Coding - Patience

- We'll start with some simple coding below
- First code examples are not flashy
- Code is like lego bricks...
- -Individual pieces are super simple
- -Eventually build up great combinations
- But we have to start small

Foreshadowing

Within a few hours of lecture, we'll be doing special effects with images such as the following:

But for now we just have print()!

Patience We're going to start by learning a few programming elements, and later we'll recombine these few elements to solve many problems. These first elements are simple, so they are not much to look at on their own. Be patient, soon we'll put these elements together -- like lego bricks -- to make pretty neat projects.

Javascript Computer Langauge

- Javascript code plus some extensions just for CS101
- Our code phrases are small...
- -Just big enough to experiment with key ideas
- -Not full, professional programs

- -But big enough to show the real issues, bugs of coding

For this class, we'll use a variant of the language known as "Javascript", with some added features for this course. The Javascript language works in the web browser, so all of our experiments can live right in the browser with nothing else required. We'll look at just the of Javascript needed for our experiments, not the full language one would see using Javascript professionally. That said, Javascript is a real language and our code is real code. Our small programs show the important features of code, while keeping things fast and small.

1. First Code Example - Print

Here is code which calls the "print" function. Click the Run button below, and your computer will run this code, and the output of the code will appear to the right.

```
print(6);
print(1, 2);
```

Run executes each line once, running from top to bottom

- **print** is a function -- like a verb in the code
- Numbers within the parenthesis (...) are passed in to the print function
- Multiple values separated by commas
- **Experiments** change the code and run after each change see the new output:
  -Change a number
  -Add more numbers separated by commas inside the print(...)
  -Copy the first line and paste it in twice after the last line
  -I promise the output will get more interesting!
- Note **Narrow Syntax** not free form
  -Allowed syntax is strict and narrow
  -e.g. cannot leave out a comma

-A reflection of the inner, mechanical nature of the computer
-Don't be put off - "When in Rome..."
-We're visiting the world of the computer

- Note "print" is not a normal part of Javascript, I added it for CS101

2. Print And Strings

```
// The line below prints one number and one string
print(6, "hi");
print("hello", 2, "bye");
```

A **comment** begins with // and extends through the end of the line. A way to write notes about the code, ignored by the computer.

- A **string** is a sequence of letters written within quotes to be used as data within the code
-e.g. "hello"
-Strings work with the print function, in addition to numbers
-Strings in the computer store text, such as urls or the text of paragraphs, etc.
- **Experiments:**
-Edit the text within a string
-Add more strings separated by commas
-Add the string "print" - tricky!
-Inside the quote marks is just **data**
- Code = instructions that are Run
- Data = numbers, strings, handled by the code

Note that **print** is recognized as a function in the code vs. the "hello" string which is just passive data (like verbs and nouns). The computer ignores the comments, so they are just a way for you to write notes to yourself about what the code is doing. Comments can be use it to temporarily remove a line of code -- "commenting out" the code, by placing a "//" to its

Thinking About Syntax and Errors (today's key message!)

- Syntax -- code is structured for the computer
- Very common error -- type in code, with slight syntax problem
- Professional programmers make that sort of "error" all the time
- Fortunately, very easy to fix ... don't worry about it
- Not a reflection of some author flaw
- Just the nature of typing ideas into the mechanical computer language
- Beginners can be derailed by syntax step, thinking they are making some big error
- Exercise to inoculate you all: a bunch of typical syntax errors + fixing them
- Fixing these little errors is a small, normal step
- Note: Firefox gives you the best error messages

Syntax The syntax shown above must be rigidly followed or the code will not work: function name, parenthesis, each string has opening and closing quotes, commas separating values for a function call.

The rigidity of the syntax is a reflection of the limitations of computers .. their natural language is fixed and mechanical. This is important to absorb when working with computers, and I think this is where many people get derailed getting started with computers. You write something that any human could understand, but the computer can only understand the code where it fits the computer's mechanical syntax.

When writing for the computer, it is **very common** to make a trivial, superficial syntax mistakes in the code. The most expert programmers on earth make that sort of error all the time, and think nothing of it. The syntax errors do not reflect some flawed strategy by the author. It's just a natural step in translating our thoughts into the more mechanical language of the computer. As we'll see below, fixing these errors is very fast.

It's important to not be derailed by these little superficial-error cases. To help teach you the patterns, below we have many examples showing typical errors, so you can see what the error messages look like and see how to fix them. For each snippet of

code below, what is the error? Sometimes the error message generated by the computer points to the problem accurately, but sometimes the error message just reveals that the error has so deeply confused the computer that it cannot create an accurate error message. Firefox currently produces the most helpful error messages often pointing to the specific line with problems.

Syntax Error Examples

These syntax problems are quick to fix.

```
print("a");
prlnt(1, "b");
print(2, "c", 3);
```

```
print("a");
print(1, "b);
print(2, "c", 3);
```

```
print("a");
print(1, "b");
print(2, "c", 3;
```

```
print("a");
print(1 "b");
print(2, "c", 3);
```

```
// This one has two syntax errors
print("a");
pront(1, "b");
print(2, "c", 3;
```

You Try It

Change the code below so, when run, it produces the following output:

```
1 2 buckle
3 4 knock
```

```
print(1, "hi");
```

For the example problems shown in lecture, the solutions are available as shown below. So you can revisit the problem, practice with it, and still see the solution if you like.

Code-2 Variables and =

This is a short section, to add the idea of **variables** to the code.

- A "variable" is like a box that holds a value
  - `x = 7;`
- The above line stores the value 7 into the variable named **x**
- Later appearances of **x** in the code retrieve the value from the box
- Using = in this way is called "variable assignment"
- How we use variables in CS101:
  -Assign a value into a variable once
  -Then use that variable many times below

Code

```
x = 7;
```

7

x

```
x = 7;
print(x);
print("lucky", x);
print("x is", x);
```

- **Experiments:**
  -Try assigning (=) these values to x: 8, "hi"
- The name **x** can be anything we choose, so long as we are consistent
  -Try changing it to **xyz** throughout
- The Point: store a value once, use it on several lines, can save repetition
- Not the same as = in algebra, that means two things are equal forever
- = in code is simple -- just puts a value in a box when that line runs

Variables work as a shorthand -- we = assign a value into a variable, and then use that variable on later lines to retrieve that value. In the simplest case, this just works to avoid repeating a value: we store the value once, and then can use it many times. All computer languages have some form of variable like this, storing and retrieving values.

You Try It

Change the code below so it produces the following output. Use a variable to store the string "Aardvark" in a variable on the first line like `x = "Aardvark";`, then use the variable x on the later lines. In this way, changing just the first line to use the value "Zebra" or "Alice" or whatever changes the output of the whole program. The print() function automatically puts spaces between multiple items.

```
Aardvark Aardvark Aardvark
Here is Aardvark
Aardvark has arrived
```

Solution:

```
x = "Aardvark"
print(x, x, x);
print("Here is", x);
print(x, "has arrived");
```

Image-1 Introduction to Digital Images

In this section we'll look at how digital images work. (Hat tip to Mark Guzdial and Barbara Ericson for promoting the media computation idea of using images, sounds etc. to introduce computing.)

Digital Images

- Digital images are everywhere
- Look natural, smooth
- Behind the scenes: lots of little numbers

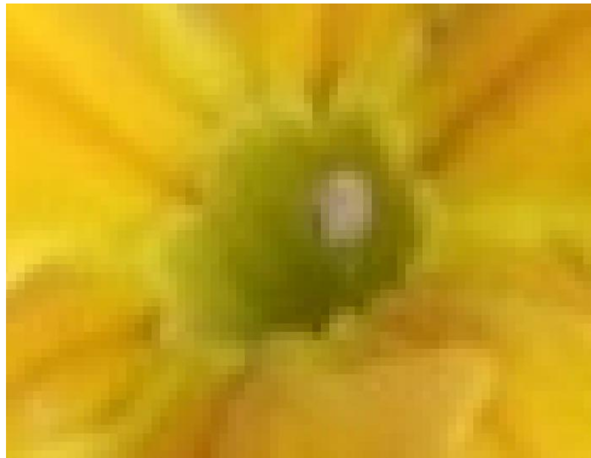
You see images on computers all the time. Here we will look behind the curtain, seeing how images are put together. What looks like a whole image to us, in the computer is a structure made of many little numbers.

Aside: I suspect that the explosion of "social media" has much to do with the increased ease and quality of digital images. They are easy to create, they are great to look at, they fit into daily life.

Here is a digital image of some yellow flowers:

Zoom In - Pixels

- Zoom in 10x on upper left flower
- The image is made of "pixels"
- Each pfixel: small, square, one color
- Perceive the whole scene, not tiny pixels
- "megapixel" is 1 million pixels
- How many pixels in an image 800 pixel wide, 600 pixels high?
  - 800 x 600 = 480,000 pixels (about 0.5 "megapixels")
- Typical digital image 5-20 megapixels

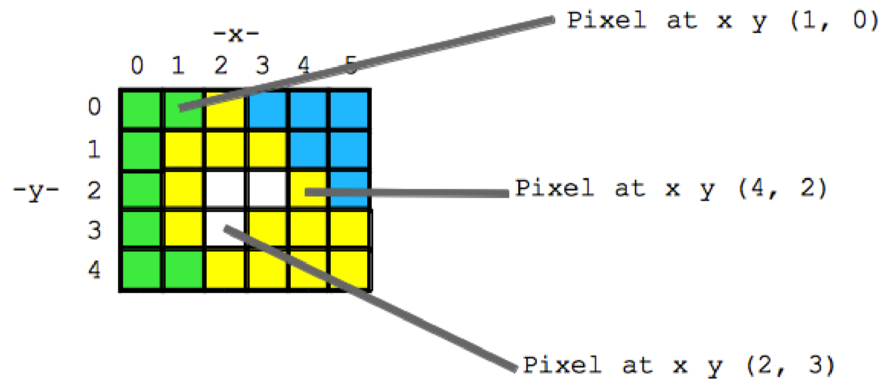


Zooming in on the upper left flower, we can see that it is actually made of many square "pixels", each showing one color.

X/Y Grid of Pixels

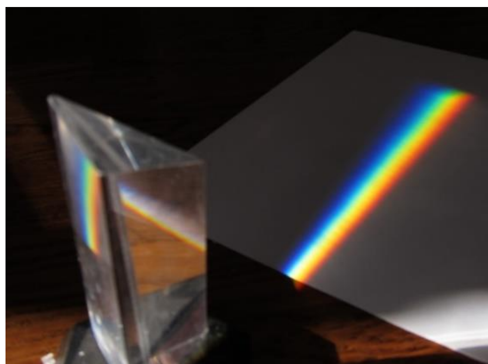
We will not work with individual x/y coordinates too much. You just need to appreciate that there is this x/y coordinate system, so that every pixel in an image has some x/y coordinate that identifies its location within the image grid.

- Pixels are organized as a x/y grid
- "origin" (0, 0) at the **upper left corner**
- X grows going to the right
- Y grows going down
- x=0, y=0 "origin" at upper left - (0, 0)
- x=1, y=0 one pixel to the right - (1, 0)
- We're not using x,y too much, but know the basic idea



History Aside - Newton's Color Prism

- Newton's famous experiment
- White light - broken up into pure colors, continuous
- Red, orange, yellow, green, blue, indigo, violet (ROY G BIV)
- Mix select colors to make other colors
- Like running the experiment backwards to make white
- Funny thing about "indigo"



Sir Isaac Newton did the famous prism experiment in 1665, showing that white light is made up of a spectrum pure colored light. Here is a picture of the experiment on my floor. White sunlight is coming in from the left into the glass triangular prism which splits

up the light. Coming out of the prism we have a continuous range of pure colors, and a few are picked out by name: red, orange, yellow, green, blue, indigo, violet (ROY G BIV).

Funny story: why add "indigo" in there, instead of just leave it at blue? Reflecting the mysticism of his time, Newton felt the number of colors should match the number of planets, which at the time was seven. It is hard to imagine how little they knew, how mysterious the operation of the world was compared to today. Of course who knows what we are ignorant of today that will be obvious in 100 years.

RGB Color Scheme - Red Green Blue

- Each pixel is one color - how to represent?
- RGB red/green/blue scheme
- Make any color with combinations of red/green/blue light
- We'll just play with the notion of color space informally
- Aside: technically color is a 3-dimensional space
  - The Newton prism shows the "hue" dimension
- Mixing lights works differently from mixing paints

How to represent the color of a pixel? The red/green/blue (RGB) scheme is one popular way of representing a color in the computer. In RGB, every color is defined as a particular combination of pure red, green, and blue light.

RGB Explorer

The best way to see how RGB works is to just play with. This RGB Explorer page which shows how any color can be made by combining red, green, and blue light.

RGB - Three Numbers

- Any color can be made by combining red/green/blue light
- Any color can be represented by three numbers
- Not just 0 and 255, intermediate values 12, 238, 39
- e.g. r:250 g:10 b:240 - purple, or just say "250 10 240"
- e.g. r:100 g:100 b:0 - dark yellow, or just say "100 100 10"

So essentially, any color can be encoded as three numbers .. one each for red, green, and blue.

Color	Red number	Green number	Blue number
red	255	0	0
purple	255	0	255
yellow	255	255	0
dark yellow	100	100	0
white	255	255	255
black	0	0	0

In RGB, a color is defined as a mixture of pure red, green, and blue lights of various strengths. Each of the red, green and blue light levels is encoded as a number in the range 0..255, with 0 meaning zero light and 255 meaning maximum light..

So for example (red=255, green=100, blue=0) is a color where red is maximum, green is medium, and blue is not present at all, resulting in a shade of orange. In this way, specifying the brightness 0..255 for the red, blue, and green color components of the pixel, any color can be formed.

Pigment Note -- you may have mixed color paints, such as adding red and green paint together. That sort of "pigment" color mixing works totally differently from the "light" mixing we have here. Light mixing is, I think, easier to follow, and in any case, is the most common way that computers store and manipulate images.

It's not required that you have an intuition about, say, what blue=137 looks like. You just need to know the most common RGB patterns we use.

You Try It Challenge

1. In the RGB Explorer, play with the sliders to make the light tan color of a cafe latte.

2. Figure out how to make orange

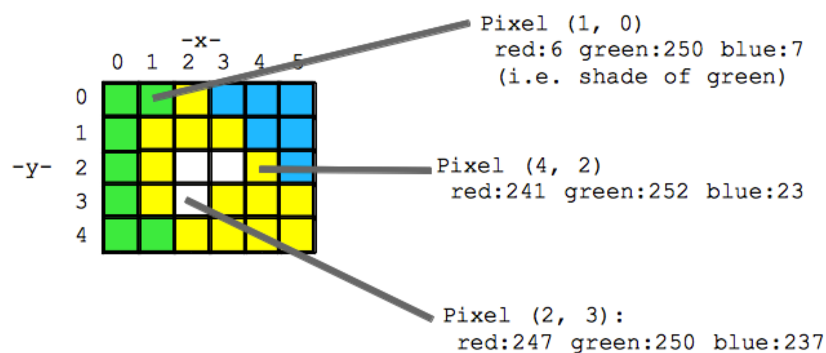
In Class Demo - Super Bright LEDs

Demo! MaxM RGB Module

Incandescent lights can pass through a filter to give colored light. In contrast, LEDs intrinsically emit light of a specific color.

Image Diagram with RGB

- Now have the complete diagram of an image
- Pixels in a grid, each identified by an x,y address
- Each pixel has 3 numbers to define its color
- Written as "red:6 green:250 blue:7"
- Or just "6 250 7"



Theme: Digital Atomization

- Start with whole image
- "Atomized" into lots of little numbers in the computer
- How do you change an image on the computer?
- Change some of the numbers → changes the image

We started with a whole image, and reduced it to a big collection of small elements. This a common theme in computer science -- what looks like an organic complicated whole, is "atomized" in the computer ... made up of a very large collection of very simple elements.

So we can start with a whole, textured digital image of something. Then break it down into small square pixels. Then each pixel breaks down to 3 numbers in the range 0..255. This is a typical computer pattern -- something whole and organic is, if you look behind the scenes, is broken down and represented as a lot of little numbers.

How to change the image? Changes to the image are done by looking at and changing the numbers that make up the image.

Image-2 Code and Images

In this section, we'll look at simple code to load and manipulate a digital image. Here we'll just manipulate one pixel at a time. In the next section, scaling up to operate on thousands of pixels at a time.

x.png

- x.png - a tiny example image
- 10 pixels wide by 10 pixels high
- PNG is an image format, like JPG

The image "x.png" is very simple - it's a very small black square with a white "x" at its center. Here is x.png:

PNG (Portable Network Graphics) is a format to store an image in a computer file, like JPG and GIF.

x.png Code Example

- Three line program
- First line stores image named "x.png" into variable named "image":
- `image = new SimpleImage("x.png");`
- `image.setZoom(20);` -- set zoom option to 20 (10, 20, ... whatever)
- `print(image);` -- print out image as usual
- Zoom: each pixel shown 20x size here

Our first image code example loads the x.png image into a variable and prints it. Run the code to see what it does. Try changing the `image.setZoom(20)` to use the number 10 or 30 instead.

```
image = new SimpleImage("x.png");
image.setZoom(20);
print(image);
```

pixel.setRed(255) Example

- Now add some pixel-manipulation lines
- First line gets pixel (0, 0) and stores it in a variable named "pixel":
- `pixel = image.getPixel(0, 0);`
- `pixel.setRed(255);` -- set red value of that pixel
- Try setRed() with different values in the range 0..255

```
image = new SimpleImage("x.png");
image.setZoom(20);

pixel = image.getPixel(0, 0);
pixel.setRed(255);

print(image);
```

Pixel Set Red/Green/Blue Functions

- `pixel.setRed(number);` -- set the red value of the pixel to be the given number (0..255)
- `pixel.setGreen(number);` -- set the green value
- `pixel.setBlue(number);` -- set the blue value

noun.verb() Pattern

- Computer code has a widely used patterns
- One, we're seeing here is the **noun.verb()** pattern
- The noun is the thing we want to operate on (e.g. a pixel)
- The verb (aka "function") operates on that noun
- e.g. the functions setRed and setZoom
- Getting used to this pattern, our code will look less weird

Aside: Image Functions Reference

For later reference, there is a separate image functions reference page which lists in one place all the image functions such as `pixel.setRed(number)` we are using here.

Experiments On Pixels (0, 0) and (1, 0)

To see how `image.getPixel(x, y)` and `pixel.setRed(number)` etc. work, we'll try the experiments listed below (use the "show" button to see the solution code).

```
image = new SimpleImage("x.png");
image.setZoom(20);

pixel = image.getPixel(0, 0);
pixel.setRed(255);

print(image);
```

Example Problems	Solution
Set pixel (0, 0) to be green.	`pixel = image.getPixel(0, 0);` `pixel.setGreen(255);`

Set pixel (0, 0) to be yellow.	`// Set red and green to 255, leave blue at 0`
	`pixel = image.getPixel(0, 0);`
	`pixel.setRed(255);`
	`pixel.setGreen(255);`

Set pixel (1, 0) to be yellow. Where is that pixel?	`pixel = image.getPixel(1, 0);`
	`pixel.setRed(255);`
	`pixel.setGreen(255);`
	`// getPixel(1, 0) retrieves the pixel`
	`// one to the right of pixel (0, 0).`

2 Pixels **Set pixel (0, 0) to green (1, 0) to red.**	`// Set (0, 0) to green`
	`pixel = image.getPixel(0, 0);`
	`pixel.setGreen(255);`
	
	`// Change pixel variable to refer to (1, 0)`
	`// then set it to red`
	`pixel = image.getPixel(1, 0);`
	`pixel.setRed(255);`

Set pixel (0, 0) to white.	`pixel = image.getPixel(0, 0);`
	`pixel.setRed(255);`
	`pixel.setGreen(255);`
	`pixel.setBlue(255);`

Set pixel (0, 0) to be dark yellow -- set the necessary colors to 150 instead of 255.	`pixel = image.getPixel(0, 0);`
	`pixel.setRed(150);`
	`pixel.setGreen(150);`

Set pixel (1, 0) to be a light, pastel red.	`pixel = image.getPixel(1, 0);`
	`pixel.setRed(255);`
	`pixel.setGreen(200);`
	`pixel.setBlue(200);`
	`// Set red at 255, green and blue equal but lower.`

Image-3 Loops

Previously, we had things like this `pixel.setRed(200);`

With one line of code, change the red value of one pixel. In this section, we'll look at the "for loop" construct, which can run a bit of code thousands of times -- a huge increase in power.

Loops - Box Analogy

- Suppose you had 5,000 cardboard boxes in a warehouse and a robot
- You want the robot to move all the boxes from one corner to another
- It's a dumb computer, so you explain how to move one box in great detail
- You want to say: repeat those same steps **for all the boxes**
- That's loops - do some steps "for all these items"

flowes.jpg

- flowers.jpg: 457 pixels wide by 360 pixels high - 164,520 pixels
- pixel.setRed(0); one pixel at a time ... not practical
- Want: we specify some code, computer runs it again and again, once for each pixel

Accessing one pixel at a time, e.g. pixel at (0, 0), then the pixel at (1, 0), etc. is not a good way to work on an image with thousands or millions of pixels. We'd like to say something like "for each pixel do this", and let the computer fiddle with the details of going through all the (x, y) values to look at each pixel once.

The very powerful "for loop" structure we'll learn here provides exactly this "for each pixel do this" feature. The loop takes a few lines of our code, and runs those lines again and again, once for each pixel in the image.

For-Loop Example 1

Run this. What does it do?

```
image = new SimpleImage("flowers.jpg");

for (pixel: image) {
  pixel.setRed(255);
  pixel.setGreen(255);
  pixel.setBlue(0);
}

print(image);
```

For each pixel, the body code sets the red, green, and blue values all to 255, 255, 0. None of the original flower data is left. All the RGB numbers are changed in the loop.

How Does That Loop Work?

- **Body** of the for-loop is the lines indented within curly braces { .. }
- Body lines are run again and again, once for each pixel
- **pixel** variable refers to a different pixel for each run of the body

For-Loop Syntax - 2 Parts

- For-loop syntax a little weird, but it's the same every time
- When we want "for every pixel do this", we'll use a for-loop
- For-loop has 2 parts
- 1. `for (pixel: image) {` - start the for-loop
- 2. "body" lines of code within curly braces { .. }
- Nice to indent the body lines, showing that they are special and go together

```
image = new SimpleImage("flowers.jpg");

for (pixel: image) {
  pixel.setRed(0);
  pixel.setGreen(0);
  pixel.setBlue(0);
}

print(image);
```

Open the for-loop "For each pixel, do the following"

"body" code lines in { .. } Lines run again and again, once for each pixel.

Education Research Aside

- If you just watch the teacher go through it, you absorb a little
- Surprisingly, if you engage to answer a question about the material you absorb a lot more
- Even if the question is easy .. just switching out of the passive mode seems to do it
- So I try to put these You Try It activities in here

Example 2 - Body Code Running Thousands of Times? You Try It

flowers-small.jpg:

- Here's an example using "flowers-small", 100 pixels wide by 79 pixels high
- 100 * 79 is 7,900 pixels total
- Body code sets each pixel to be green
- print("before"); -- before the loop
- print("after"); -- after the print(image)
- Experiment 1:
  -What do you think this will print?
  -Run it and see
- Experiment 2:
  -Add this line inside the loop: `print("inside");`
  -What do you think this will print?
  -Run it and see

```
image = new SimpleImage("flowers-small.jpg");

print("before");
for (pixel: image) {
  pixel.setRed(0);
  pixel.setGreen(255);
  pixel.setBlue(0);
}

print(image);
print("after");
```

The lines of code in the body run again and again, once for each pixel. Therefore, a line of code inside the body, inside the curly braces { }, will run thousands or millions of times. In contrast, the lines of code outside the body just run once. Inside the body, "pixel" refers to a different pixel for each run of the body.

For-Loop Example 3

Look again at flowers.jpg.Yellow is made or red + green, so we know that the yellow parts of the image have high red and green

values. So what happens if, for each pixel, we set red to 0? What are the RGB values for a typical pixel on the yellow flowers look like before this loop runs? What about after?

- Previous examples overwrote all the flower data ... not realistic
- Modify the data in flowers.jpg
- Q: What are the green leaves made of? What are the yellow flowers made of?
- Experiment
  -For each pixel, set red to 0
  -What is the body code to do this?
  -Which of RGB are high for the yellow flowers?
  -So what does setting red to 0 look like? Run it.
  -Could try it in RGB Explorer, make yellow, then set red to 0
- Theme: we discuss algorithm ideas, then you code it up for the computer

```
image = new SimpleImage("flowers.jpg");

for (pixel: image) {
  // your code here

}

print(image);
```

Solution:

```
  // your code here
  pixel.setRed(0);
```

The body code pixel.setRed(0); is run by the loop again and gain, once for each pixel in the image. Since the yellow flowers are made with red + green light, setting the red to 0 for each pixel results is

greenish flowers. The green leaves aren't changed much, since their red values were near 0 anyway.

For-Loop Example 4 - Red Channel

- Experiment or Lecture-example
- For each pixel, set green and blue to 0
- What is the body code to do this?
- The result is known as the "red channel"
- Seeing an image of just the red light areas

```
image = new SimpleImage("flowers.jpg");

for (pixel: image) {
  // your code here

}

print(image);
```

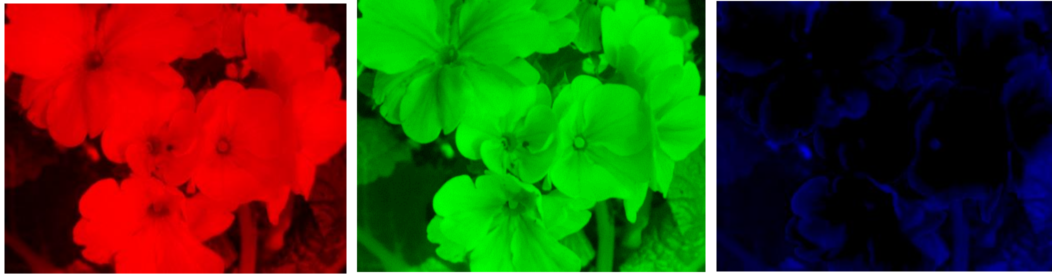
Solution:

```
  // your code here
  pixel.setGreen(0);
  pixel.setBlue(0);
```

Setting green and blue to 0 everywhere, all that is left is the area of red light that went into the original image, aka the "red channel" of the image. There are analogous green and blue channels we could look at, and the image is all three combined. The red light is most prominent for the area of yellow flowers, which makes sense as we know that yellow = red + green.

Red, Green, and Blue Channels

- The code example above computes the "red channel" image
- The red channel is just the red light of the image, with blue and green at zero
- There are analogous blue and green channel images
- The whole image is just the sum of the three channels: adding together all the light
- The three channels are shown below



For-Loop Conclusions

- A powerful feature, we specify a few lines of code, computer takes care of running them thousands of times
- Computer = powerful + stupid theme
- Code inside the loop is special: run it for every pixel
- That's why code inside the loop should be indented (optional)
- Note: Javascript does not have this feature, I added it for CS101

Image-4 Expressions

In this section, we'll look how to use expressions to greatly expand what we can do in a loop. With expressions, we'll be able to write real image manipulation code, and in particular solve some puzzles based on images.

Expressions 1 + 1

```
print(11 + 31);
```

We have seen code that "calls" a function where we pass in a value within the parenthesis, such as the value 42 passed in to the print function below.

```
print(42);
```

Instead of a plain number like 42, an "**expression**" written in the code like 11 + 13 computes the value to use. For example you could write something like this:

```
print(11 + 31);
```

When that line runs, the computer first computes the expression 11 + 31, yielding 42. Then in effect it calls print(42), passing in the computed value. Anywhere in the code where we have used a fixed number like 0 or 255 of whatever, we can instead write an expression, letting the code compute a value when that line runs.

pixel.getRed() / pixel.getGreen() / pixel.getBlue()

We have not used them until now, but there are three pixel functions that **get** the red, green or blue value out of a pixel. These will be very handy to use in expressions.

- `pixel.getRed()` -- retrieves the red value from a pixel
- `pixel.getGreen()` -- retrieves the green value
- `pixel.getBlue()` -- retrieves the blue value

Notice that calling, say, `pixel.getRed()` **has the pair of parenthesis ()** after it . The syntax requires that calling a function includes the parenthesis, even if there's nothing inside the parenthesis. A good example of the inflexible nature of computers.

Set/Get Pattern: combine pixel.setRed() and pixel.getRed()

- Suppose we want to **double** the red value of a pixel
- This is a more realistic sort of operation
- `// Doubles the pixel's red value`
- `old = pixel.getRed();`
- `pixel.setRed(old * 2);`

The `pixel.getRed()` can be combined

with `pixel.setRed(`*`number`*`)` to operate on a pixel. For example, the

above code snippet doubles the pixel's red value. The first line retrieves the red value from the pixel and stores that value in an variable named **old**. Say in this case that the red value is 50. The

second line computes `old * 2` (100 in this case), and sets that

value back into the pixel as its new red value. The net effect is to double the red value of the pixel from 50 to 100.

- The **old** variable is not necessary
- Reduce the whole get/set to one line
- `pixel.setRed(pixel.getRed() * 2);`

- How does the line above work?
- Suppose that pixel red is 120
- Q: What does running the line above do to the pixel?
- 1. Evaluates the `pixel.getRed() * 2` expression, ...
- 2. expression pixel.getRed() is 120
- 3. expression 120 * 2 is 240
- 4. calls `pixel.setRed(240);`
- Result is changing the pixel red from 120 to 240
- Whatever the red value, the line doubles it

The code `pixel.getRed() * 2` is an expression, which is whatever

the old red value was multiplied by 2. This expression is evaluated first, resulting a number such as 240. Then in effect pixel.setRed(240); is called. The setRed() etc. functions automatically limit the value set to the range 0..255. If setRed() is called with a value a greater than 255, it just uses 255, and likewise if a value less than 0 is passed in, it just uses the value 0.

Loops With Expressions - Set/Get Patterns

- Now can express **relative** color number changes
- e.g. double the red value (with set/get pattern):
  `pixel.setRed(pixel.getRed() * 2);`
- e.g. halve the red value:
  `pixel.setRed(pixel.getRed() * 0.5);`
- Bottom line, we'll use this pattern a lot:
  `pixel.setRed(pixel.getRed() * something);`

Before we could only express ideas like "set the red value to 200". Now we can express what new value we want in terms of the old value, like "triple the red value", or "set the red value to 75% of what it was".

Image Expression Example 1

- Suppose we want to change the yellow flowers to look orange
- Set the green value to 75% of its original value (i.e. * 0.75)

- What is setGreen/getGreen combination to do this?

```
image = new SimpleImage("flowers.jpg");

for (pixel: image) {
  // your code here

}

print(image);
```

Solution code:

```
  // your code here
  pixel.setGreen(pixel.getGreen() * 0.75);
```

Image Expression Example 2 - You Try It

- Set red, green, and blue each to be * 0.75 of their original values, then try 0.5 and 0.25 (algorithm)
- What is the code to do this? (code)
- What is the effect on the image?

```
image = new SimpleImage("flowers.jpg");

for (pixel: image) {
  // your code here

}

print(image);
```

Solution code:

```
  // your code here
  pixel.setRed(pixel.getRed() * 0.75);
  pixel.setGreen(pixel.getGreen() * 0.75);
  pixel.setBlue(pixel.getBlue() * 0.75);
  // Then try multipliers of 0.5 and 0.25
  // The effect is to make the image darker
(towards 0)
```

Image Expression Example 3 (optional)

- Try to improve the orange on the flowers
- Try modifying both red and green
- Change red to be 1.3 times its original value
- Change green to be 0.75 times its original value
- What is the code to do this?

```
image = new SimpleImage("flowers.jpg");

for (pixel: image) {
  // your code here

}

print(image);
```

Solution code:

```
  // your code here
  pixel.setRed(pixel.getRed() * 1.3);
  pixel.setGreen(pixel.getGreen() * 0.75);
  // The effect is to change the yellow flowers to
light orange
```

5-10-20 Image Puzzles

- 5-10-20 Puzzles
- Image where red, green, and blue divided by 5, 10, or 20
- Scale them back up by * 5, * 10, * 20 to recover the original image
- Puzzle: don't know which number goes with which color
- Experiment to figure it out
- There are only 6 possibilities:
- 5 10 20, 5 20 10, 10 5 20, 10 20 5, 20 5 10, 20 10 5
- i.e. 5 first x 2, 10 first x 2, 20 first x 2

5-10-20 Banana Puzzle

- Yellow banana, background of dark red bricks

- Bits of dark green moss between the bricks
- Use code pattern: `pixel.setRed(pixel.getRed() * 5);`
- Scale up all three colors, using factors 5, 10, 20 to fix the image

```
image = new SimpleImage("51020-banana.png");
for (pixel: image) {
  // your code here

}

print(image);
```

Solution code:

```
  // your code here
  pixel.setRed(pixel.getRed() * 20);
  pixel.setGreen(pixel.getGreen() * 5);
  pixel.setBlue(pixel.getBlue() * 10);
```

In the solution image, you will see some "banding" of the yellow of the banana. This is because the red and green channels were scaled down to a range as small as just 0, 1, 2, .. 12. With so few values, the image only represent a few different shades of yellow, and those are the bands we see if we look carefully at the banana. We will talk about "banding" more when we talk about digital media formats.

Image-5 Puzzles

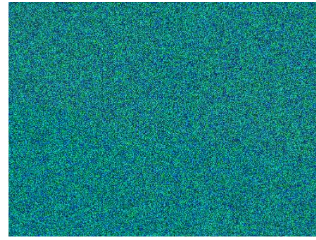
This is a short section to look at type of puzzle built out of image manipulation code.

Code To Fix the Pixels

- Backstory:
- There is an image of some unknown object
- The red/green/blue values have been distorted, hiding the real image
- Write code to fix the red/green/blue vlaues
- Recover the original image to solve the puzzle

Gold Puzzle

Here we have the "gold" puzzle image -- fix it to see the real image



Gold puzzle parameters:

- The green and blue values are all just random values in the range 0..255 ("snow" or "speckle noise")
- The data of the real image is exclusively in the red values
- In addition, the red values have all been divided by 10 (made dark)
- The green/blue snow is obscuring the real image
- Write code to recover the real image

```
image = new SimpleImage("puzzle-gold.png");

for (pixel: image) {
  // your code here

}

print(image);
```

Solution code:

```
// Strategy: zero out blue and green as they
// are just garbage noise data.
// Then scale red up by 10x to see the real
// image in red.
  // your code here
  pixel.setGreen(0);
  pixel.setBlue(0);
  pixel.setRed(pixel.getRed() * 10);
```

Seeing Red

In this case, our solution shows the image, but it's all in red. What we have here is basically a black-and-white image, but it is shown in the black-red range, rather than the usual black-white. For this section, we'll say that's good enough. We'll see how to fix the red image so it looks like a proper black-and-white image in a later section.

Image-6 Grayscale

In this section, we'll look at the structure of grayscale vs. color images, and some code to play with that difference.

Gray Among The RGB - You Try It

- Demo experiment - visit the RGB explorer
- Figure out how to make a shade of gray
- e.g. RGB values to make: dark gray, medium gray, light gray
- We'll say that these grays lack "hue"

Answer: the RGB scale is calibrated so that when a color's three red/green/blue numbers are equal, the color is a shade of gray. E.g. red=50 green=50 blue=50 is gray, without any bias towards red, green, or blue hue. If a pixel were red=75 green=50 blue=50 it would be a bit reddish, but making them all equal, it's not towards any particular hue.

Examples of gray colors in RGB:

red	green	blue	color
50	50	50	dark gray
120	120	120	medium gray
200	200	200	light gray
250	200	200	not gray, reddish
0	0	0	black (a sort of gray)
255	255	255	white (ditto)

Red Liberty Example Problem

Here is an image of the Statue of Liberty where all of the data is in the red values, so the whole image looks red (we saw this sort of image in an earlier puzzle solution). The green and blue values are all zero. This image looks quite wrong.

For this example, we'll write code to fix this image by copying the red value over to be used as the green and blue value. So for a pixel, if red is 27, set green and blue to also be 27. What is the code to do that? What will be the visual result of this?

- All green and blue values are 0
- The red values are real
- Here's what individual pixels look like:

red	green	blue
65	0	0
53	0	0
100	0	0
19	0	0
...	0	0

- Change the image to be grayscale, not just red
- Change the red/green/blue values to be all equal for each pixel.
- What change to the pixels will accomplish this?

- Algorithm: for each pixel, set the green value to be the same as the red value
- Likewise, set the blue value to be the same as the red value
- Code: you try it below

```
image = new SimpleImage("liberty-red.jpg");
for (pixel: image) {
  // your code here

}
print(image);
```

Solution code:

```
  // your code here
  // Set green and blue values to be the same as the red
value.
  pixel.setGreen(pixel.getRed());
  pixel.setBlue(pixel.getRed());
  // Usually code combines setRed() with getRed(),
  // but this code gets a value from one color
  // and sets it into another color.
```

Converting Color To Grayscale

- How to covert a regular color image to grayscale?
- Problem: for each pixel, how dark/light is it (ignoring hue)
- Choose a few pixels out of flowers.jpg, each in a row below
- Q: How to decide which pixel below is brightest? darkest?

	red	green	blue
pixel-1	200	50	50
pixel-2	0	75	75
pixel-3	100	250	250

Looking at just red or blue or green in isolation, it's hard to tell which pixel is brightest or darkest in the above table.
The **average** combines and summarizes the three values into one number 0..255. The average shows how bright the pixel is, ignoring hue: 0 = totally dark, 255=totally bright, with intermediate average values corresponding to intermediate brightnesses. More complicated brightness measures are possible, but average is simple and works fine for our purposes.

- We compute average of red/green/blue values for each pixel
- To average 3 numbers, add them up and divide by 3
- average = (red + green + blue)/3
- Average combines red/green/blue into one number
- The average measures how bright the pixel is 0..255
- Ignoring hue

	red	green	blue	average
				average = (red + green + blue) / 3
pixel-1	200	50	50	100 (medium bright)
pixel-2	0	75	75	50 (darkest)
pixel-3	100	250	250	200 (brightest)

Code To Compute Pixel Average

- Compute the average value of a pixel:
- Algorithm: add red+green+blue, then divide by 3
- Code below computes the average, stores it in a variable “avg"
- We'll use that line whenever we want to compute the average

```
  avg = (pixel.getRed() + pixel.getGreen() +
pixel.getBlue())/3;
```

Grayscale Conversion Algorithm

For this example, we'll write code to change the flowers.jpg image to grayscale, using the "average" strategy: for each pixel, compute the average of its red/green/blue values. This average number represents the brightness of the pixel 0..255. Then set the red, green, and blue values of the pixel to be that average number. The result is a grayscale version of the original color image. Once its working with flowers.jpg, try it with poppy.jpg or oranges.jpg. (Solution code available below)

- Pixel's average is effectively a "brightness" number 0..255
- Summarizes the 3 red/green/blue numbers as one number
- To change a pixel to grayscale:
  -Compute the pixel's average value
  -Set the pixel's red/green/blue values to be the average
  -e.g. red/green/blue all set to 75
- Now the pixel is gray, red/green/blue all equal

```
image = new SimpleImage("flowers.jpg");
for (pixel: image) {
  // your code here

}
print(image);
```

Solution code:

```
  // your code here
  avg = (pixel.getRed() + pixel.getGreen() +
pixel.getBlue())/3;
  pixel.setRed(avg);
  pixel.setGreen(avg);
  pixel.setBlue(avg);
  // For blue tint: pixel.setBlue(avg * 1.2);
```

Grayscale Followup Questions

- Q1: What happens if we pixel.setBlue(avg * 1.2)?
  - we get a blue tint, since blue is now bigger than the other two
- Q2: must the "avg = ... " line be inside the loop?
- Yes, it must be inside the loop
- The equals sign (=) only works when that line is run
- (Unlike traditional math meaning of =)
- Each pixel has different red/green/blue values
- We need to re-do the addition / divide-3 for each pixel

Grayscale Summary

- When red/green/blue values are equal .. shade of gray
- Average combines red/green/blue into one number
- The average measures how bright the pixel is 0..255
- Convert pixel to grayscale: set red, green, and blue to be the average
- Standard code line to compute average within loop. We'll use this line often for later problems.

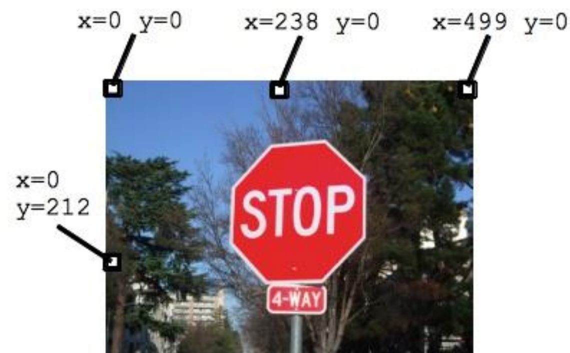
```
avg = (pixel.getRed() + pixel.getGreen() +
pixel.getBlue())/3;
```

Image-7 If Logic

- Theme 1: **Loop** bulk operation - e.g. for-loop
  - Loop body code runs again and again
- Theme 2: **Logic** test true/false - e.g. if-statement (this section)
  - Run some code only if a test is true

Our use of loops thus far have allowed us to write a little bit of code which is run for many data points. That's one big theme in computer code. The if-statement, the topic of this section, will add a second theme: the ability to write a true/false test to control if a bit of code runs or not. Combined with the loop, the if-statement will greatly expand what we can do with code.

Image X/Y Refresher



- Today we'll use the stop sign image
- 500 pixels wide, 375 pixel high
- Every pixel is identified by an x,y coordinate
- x=0 y=0 is the upper left pixel, aka (0, 0)
- X values increase going to the right
- Y values increase going down

pixel.getX() pixel.getY() image.getWidth() image.getHeight()

- Pixels x/y functions: `pixel.getX()` and `pixel.getY()`
- These retrieve the X and Y values of the pixel
- Analogous to `pixel.getRed()`
- There are also `image.getWidth()` and `image.getHeight()`
- e.g. image.getWidth() on the stop sign image yields 500
- For now, you may type in literal numbers like 500

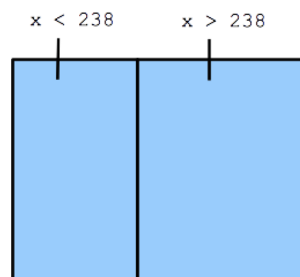
If-Statement Demo

The if-statement has a true/false test which controls if some code is run or not. Here is an example if-statement shown inside a for-loop with the "stop.jpg" image.

```
image = new SimpleImage("stop.jpg");
for (pixel: image) {
  if (pixel.getX() < 100) {
    pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(255);
  }
}
print(image);
```

- 1. `if (pixel.getX() < 100) {` - "test" inside parenthesis
- 2. Curly braces around body code after the test `{ .. }`(indented)
- How works: the loop hits every x,y. Only some x,y pass the test.
- For now just using < (less than) or > (greater than) in test
- Experiment 1: try pixel.getX() < 150 300 10
- Experiment 2: now try pixel.getY() < 100 200
- **The Flip**
- Change < to >, .e.g. `(pixel.getX() > 100)`
- Get the opposite (well, very close to the opposite)

Flip Diagram



If-Statement - You Try It

- Figure out code for the following
- stop.jpg is 500 pixels wide, 375 pixel high
- a. Set approximately the left 2/3's of the image to pure blue, just enough so none of the stop sign is visible
- b. Set a vertical stripe 20 pixels wide at the far right to blue
- c. Set a horizontal stripe 20 pixels high across the top of the image to black
- d. Set the flip of (c) to black, i.e. all except the top horizontal stripe to black

```
image = new SimpleImage("stop.jpg");
for (pixel: image) {
  if ( --your code in here-- ) {
    pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(255);
  }
}
print(image);
```

Solution code

```
// a. if (pixel.getX() < 355) {
// b. if (pixel.getX() > 480) {
// c. if (pixel.getY() < 20) {
// d. if (pixel.getY() <> 20) {

image = new SimpleImage("stop.jpg");
for (pixel: image) {
  if (pixel.getX() < 355) {
    pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(255);  // 0 here for black
  }
}
print(image);
```

Using image.getWidth() Expressions (optional)

- The "stop.jpg" image is 500 pixels wide
- Therefore to paint the left half, the test uses 250, as below
- The code uses 250 since that is 500/2
- Instead:
  - instead of 250 type in below `image.getWidth()/2`
  - i.e. an expression that computes the midpoint based on the image width
  - change it to work on "poppy.jpg" instead of "stop.jpg"
  - it just works!
- This technique is not required on these exercises, optional

```
image = new SimpleImage("stop.jpg");
for (pixel: image) {
  if ( --your code in here-- ) {
    pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(255);
  }
}
print(image);
```

Image-8 If Logic 2

If With Color

- Suppose we want to change the red sign to be blue
- Idea: check if red value is large
- For each pixel, check if (red > 160)
- Code for that test: `(pixel.getRed() > 160)`
- This is one strategy to test for red areas (needs improvement)

```
image = new SimpleImage("stop.jpg");
for (pixel: image) {
  if (pixel.getRed() > 160) {
    pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(255);
  }
}
print(image);
```

- The test will be true or false for each pixel

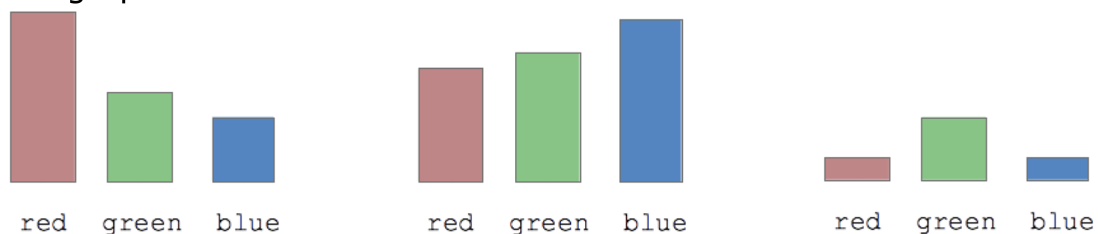
- Experiments:
- Nothing magic about 160 -- adjust it to get the result we want
- 1. Adjust the 160 value. If we change it to 220 or 250, will the if-test select more or fewer pixels than at 160? What if we set it to 100? 50?
- With a >, the 160 is like a hurdle we raise or lower
- **Restrictive:** When adjusting a test, is the goal to make it more or less restrictive?
- 2. **Flip** try flipping < > - get opposite parts of the image

The Problem With (red > 160) Test

Trying to change the stop sign to be blue, the test (red > 160) gets both white and red areas. The problem with that approach is that the red value is high in two kinds of cases -- the red part of the sign we want, but also parts of the scene that are near white. Recall that pure white is (255, 255, 255), so in whitish areas, all three values are high, so our (red > 160) test gets those too -- the white letters inside the sign are a clear example of this problem. Next we'll improve the test so it can distinguish the red and white parts of the sign.

Color Bars

Suppose we have three pixels. Each pixel has the familiar red/green/blue values. Suppose these three values are graphed as bar graphs like this:



Q: What color is the strongest for each pixel?

Look at the graphs to figure which shade dominates for each pixel. Find the tallest bar **relative to the others**. The problem with our earlier strategy was that it just looked at absolute numbers, (red > 160), failing to take into account how red relates to the other two colors.

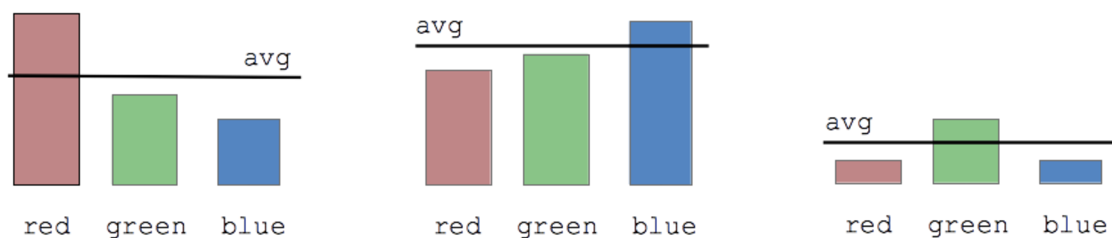
Recall Average of Pixel Color

Recall this code which, inside a loop, computes the average of the red, blue, and green values and stores that number in a variable "avg".

```
  avg = (pixel.getRed() + pixel.getGreen() +
pixel.getBlue())/3;
```

Color Bars With Average

Here are the three pixels again, but now the **avg** value is show as a line drawn across the bars.



The avg gives us a handy way to tell if a color is high relative to the others: A pixel is reddish if the red value is over the avg.

So here the first pixel is reddish and the other two are not. Or in terms of code, the test will look like `(pixel.getRed() > avg)` -- this will add the "relative to the other colors" quality that the previous test was missing.

Color Avg Test Example - Stop

- Improve the detect-red test
- Test: `(pixel.getRed() > avg * 1.1)`
- This works very well, picking out pixels with red cast
- `* 1.1` adjustment factor, more/less restrictive to get look we want
- e.g. * 1.7 makes test more restrictive
- e.g. * 0.9 makes test less restrictive
- Here 1.4 looks pretty good
- Then try the flip experiment back and forth - awesome!

```
image = new SimpleImage("stop.jpg");
for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;

  if (pixel.getRed() > avg * 1.1) {
    pixel.setRed(0);
    pixel.setGreen(0);
    pixel.setBlue(255);
  }
}
print(image);
```

This is a better way to select the red parts of the sign. For each pixel, first compute the average value for that pixel. Then compare the red value to the average to decide if the pixel is reddish. Rather than checking the literal value of the red (e.g. 160), this checks if the red is relatively large compared to the other two colors .. does the pixel lean towards red. If the test is `(pixel.getRed() > avg)` we get all the areas that have every a tiny red cast, which is not restrictive enough. The fix is to multiply the avg by some factor to make it more restrictive, like this: `(pixel.getRed() > avg * 1.1)`. The specific value, 1.1, can be tuned by trying different values, until we get the look we want. Try the values: 0.9, 1.2, 1.4, 2, 2.5. The larger the value, the higher the bar is set to detect red pixels. By adjusting the * factor, we can zero in on the look we want. Here I think 1.4 looks pretty good.

Color Avg Test You Try It - Curb

Suppose you are visiting Stanford and you park your car here, and get a parking ticket. Philosophically, they say that you are better off taking in events as they have actually happened. Nonetheless, here we'll try to fix history in code.

Challenge: write code to detect the red curb, tuning the 1.1 factor to look the best. Then (a) change the curb to medium gray red=120

green=120 blue=120. (b) change just the curb to be **grayscale**, which will look more realistic. Rather than changing the whole image to grayscale, we change just the red areas.

```
image = new SimpleImage("curb.jpg");
for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;

  if ( -- your code in here-- ) {
    pixel.setRed(120);
    pixel.setGreen(120);
    pixel.setBlue(120);
  }
}
print(image);
```

Solution code (a):

```
// This solution looks the best to me for (b).
image = new SimpleImage("curb.jpg");
for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() +
pixel.getBlue())/3;

  if (pixel.getRed() > avg * 1.1) {
    pixel.setRed(120);
    pixel.setGreen(120);
    pixel.setBlue(120);
  }
}
print(image);
```

- Tune the avg * factor to detect the red areas
- Setting to solid gray (120, 120, 120) looks a little crude
- Note that we put gray on the red plants to the right
- Our strategy selects by color, so necessarily get the red plants too
- (b) Rather than (120, 120, 120), change the red pixels to avg, making graysscale
- This looks much better, using the dark/light of the real curb
- Conclusions:
- `(pixel.getRed() > avg * 1.5)` pattern to detect pixel by color
- The body-code can do anything we wish to the detected pixel

Image-9 Bluescreen

Building on the earlier image manipulation and logic sections, here we see how to implement a movie special effect.

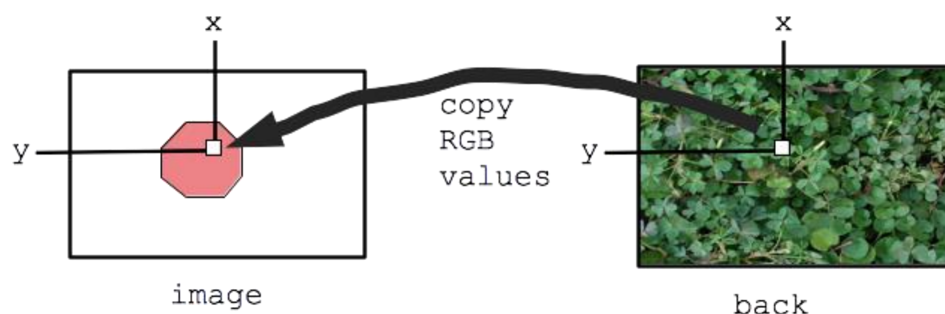
Bluescreen Special Effect

- "Bluescreen" video special effect
- Also known as Chroma Key (wikipedia)
- Video is just a series of still images, 20-60 per second
- Have **image** to modify
- Have **back** background image, pull pixels from
- Detect colored area in an image, e.g. red in stop.jpg
- For colored area, copy over pixels from back image
- Lego analogy -- same setRed() etc. we've used, arranged in a new way

Bluescreen Stop Sign Example

Say we want to change the red part of sign to look like leaves

Bluescreen Algorithm, Non-Trivial



- Two images we'll call **image** and **back**
- Detect, say, red pixels in image
- For each red pixel:
  -Consider the pixel at the same x,y in back image
  -Copy that pixel from back to image
  -What defines how a pixel looks? Three numbers.
  -Copy the RGB numbers from back pixel to image pixel
- Result: for red areas, copy over areas from back image
- Adjacent pixels maintained, so it looks good

Bluescreen Example Code

- Bluescreen plan:
- Loop over stop.jpg pixels
- Detect red pixels (avg technique * 1.5)
- We'll call this "red-high" detection
- For those pixels, copy over pixels from leaves.jpg
- For each red pixel detected:
  -Figure out x,y of pixel
  -Get pixel2 from x,y in back image
  -Copy red/green/blue from pixel2 to pixel

```
image = new SimpleImage("stop.jpg");
back = new SimpleImage("leaves.jpg");

for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;
  if (pixel.getRed() > avg * 1.5) {
    x = pixel.getX();
    y = pixel.getY();
    pixel2 = back.getPixel(x, y);  // Translate this line to English
    pixel.setRed(pixel2.getRed());
    pixel.setGreen(pixel2.getGreen());
    pixel.setBlue(pixel2.getBlue());
  }
}
print(image);
```

The above bluescreen example code detects the red part of the sign, and for those pixels, copies over the pixels from the back image. The bluescreen code has 3 key differences from the earlier examples:

1. `back = new SimpleImage("leaves.jpg");` - open a **second** image and store it in the variable "back" (thus far we've only opened one image at a time)

2. `pixel2 = back.getPixel(x, y);` - say we are looking at pixel x,y in the main image. From the **other image** get the pixel at the same x,y and store it in the variable "pixel2". We get the x and y of the current pixel in the loop with the functions pixel.getX() and pixel.getY().
3. `pixel.setRed(pixel2.getRed());` - copy the red value from pixel2 to pixel. Also do this for the other two colors. The result is that at this x,y location, the pixel is copied from the back to image.

Bluescreen Stop/Leaves Experiments - Your Try It

Try these on the above code.

1. Flip -- try replacing everything-but-the-red with leaves instead of the red, i.e. "red-low" detection
2. Sky -- try replacing the blue sky instead of the red areas of the sign. What color do you look for? Tune the * factor so only the sky is changed.
3. Shorter -- make the code a little shorter by eliminating the need for "x" and "y" variables. Put the calls to pixel.getX() and pixel.getY() right in the back.getPixel(...) call.

Solution code:

```
// flip: just change < to >

// sky: if (pixel.getBlue() > avg * 1.1) {
// This is a great example of tuning the * factor

// pixel2 in one line:
// pixel2 = back.getPixel(pixel.getX(), pixel.getY());
```

Monkey Bluescreen Example - You Try It

Now we'll do one like the movies -- film the movie start in front of a blue screen. Replace the blue background behind the monkey with pixels from moon.jpg.

Here is our monkey movie star:

Here is our background, the famous Apollo 8 photo of the earth shown rising above the moon horizon. There is a theory that this famous image helped start the environmental movement.

- Bluescreen plan:
- Loop over monkey.jpg
- Detect blue pixels, i.e. blue-high
- For those pixels: copy over moon pixels
- (opy over the earlier example code to get started

```
image = new SimpleImage("monkey.jpg");
back = new SimpleImage("moon.jpg");

for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;
  // your code here

}

print(image);
```

Solution code:

```
image = new SimpleImage("monkey.jpg");
back = new SimpleImage("moon.jpg");

for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() +
pixel.getBlue())/3;
  // your code here
  if (pixel.getBlue() > avg * 0.92) {  // detect blue-high
    pixel2 = back.getPixel(pixel.getX(), pixel.getY());
    pixel.setRed(pixel2.getRed());
    pixel.setGreen(pixel2.getGreen());
    pixel.setBlue(pixel2.getBlue());
  }
}

print(image);
```

Other Backgrounds

- Try other background images:
  -paris.jpg
  -yosemite.jpg
  -stanford.jpg
- Notice: if-logic only looks at **image** pixels
- Whatever background image we provide, it is just copied over
- The blue or whatever in the back image is ignored by our if-logic

Selecting High vs. Low

- The monkey image is especially easy to work with

- The monkey is redish and the background is blueish
- We can use red **or** blue to detect the monkey or background
- i.e. "blue-high" strategy = detect areas where blue is high
- i.e. "red-low" strategy = detect areas where red is low
- We have blue-high blue-low red-high red-low
- We have choose the best strategy depending on the image
- The body code below sets bright green on the selected pixels
- Experiments to try:
- 1. (provided) Write code for the red-high strategy to select the monkey
- 2. Change to the flip of that (red-low), selects what?
- 3. Write code and the * factor for blue-low to select the monkey
- 4. Finally, flip to blue-high, selects what?
- I like this series, reflecting a little of the perfect logic within the computer

```
image = new SimpleImage("monkey.jpg");
back = new SimpleImage("leaves.jpg");


for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;

  // red-high -> monkey
  if (pixel.getRed() > avg * 1.05) {
    pixel.setGreen(255);
  }
}
print(image);
```

Solution code:

```
// red-high -> monkey
if (pixel.getRed() > avg * 1.05) {

// flip: red-low -> background
if (pixel.getRed() < avg * 1.05) {

// blue-low -> monkey
if (pixel.getBlue() < avg * 0.95) {

// blue-high -> background
if (pixel.getBlue() > avg * 0.95) {
```

back.setAsBig(image);

Looping over a main image and then grabbing pixels from a back image, there is a problem if the back image is smaller than the main image. The code will try to access a non-existent pixel, and we get an "x/y out of bounds" error. The following line, before the loop, fixes this problem:

```
back.setAsBig(image);
```

The above line resizes the back image if necessary, so it is at least as big as the main image, which is exactly what we need for blue screening, picking pixels from the back image to place into the main image. You'll see this line in the starter code for some later problems.

Another Effect: Image Blend

Another effect we can try is blending in a "ghost" version of the monkey into the back image. We blend the colors from the two images instead of replacing entire pixels.

- Modify and print the back image, blending in the monkey
- Blend plan:
- 1. Loop over monkey image
- 2. if-logic to detect monkey with blue-low strategy
  (this is flip, previously we detected not-monkey background)
- 3. Blend in monkey: set back-pixel-red = back-pixel-red + monkey-pixel-red/2
- 4. Do it for the other 2 colors too
- 5. Print the back image
- Result is a blend of a faint monkey into the back image
- Experiment 1: play with the * factor. What does is more restrictive here? What happens when monkey-background is included?
- Experiment 2: Adjust div=2 factor, 1 2 10 20. div=2 is good to see how it works, but /10 is more artistic
- Lego analogy: same bricks as bluescreen, arranged slightly differently

```
image = new SimpleImage("monkey.jpg");
back = new SimpleImage("paris.jpg");
back.setAsBig(image);

for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;
  // your code here

}
print(back);
```

Solution code:

```
image = new SimpleImage("monkey.jpg");
back = new SimpleImage("paris.jpg");
back.setAsBig(image);

for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() +
pixel.getBlue())/3;
  // your code here
  if (pixel.getBlue() < avg * 0.9) {  // detect monkey (blue-
low)
    div = 2;  // monkey is divided by this variable below
    pixel2 = back.getPixel(pixel.getX(), pixel.getY());
    pixel2.setRed(pixel2.getRed() + pixel.getRed()/div);
    pixel2.setGreen(pixel2.getGreen() + pixel.getGreen()/div);
    pixel2.setBlue(pixel2.getBlue() + pixel.getBlue()/div);
  }
}
print(back);
```

Image-10 Advanced Bluescreen

Here we'll look at more advanced Bluescreen techniques.

Bluescreen Multi-Pass - You Try It

- Make multiple changes: change red and change blue
- Provided code detects red, copies in leaves.jpg (back)

- You Try It:
- Add a second if-statement after the first (copy/paste 1st if)
- Change to detect blue pixels, copy in yosemite.jpg pixels (back2 image)
- The second if follows the first inside the loop
- They are not inside one another, they are siblings
- They are at the same indent level
- Three pairs {..}: 1x for-loop, 2x if-statements

```
image = new SimpleImage("stop.jpg");
back = new SimpleImage("leaves.jpg");
back.setAsBig(image);
back2 = new SimpleImage("yosemite.jpg");
back2.setAsBig(image);

for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;

  if (pixel.getRed() > avg * 1.7) {  // select red (back)
    pixel2 = back.getPixel(pixel.getX(), pixel.getY());
    pixel.setRed(pixel2.getRed());
    pixel.setGreen(pixel2.getGreen());
    pixel.setBlue(pixel2.getBlue());
  }

  // Your code here


}
print(image);
```

Solution code:

```
  // Your code here
  if (pixel.getBlue() > avg * 1.2) {  // select blue (back2)
    pixel2 = back2.getPixel(pixel.getX(), pixel.getY());
    pixel.setRed(pixel2.getRed());
    pixel.setGreen(pixel2.getGreen());
    pixel.setBlue(pixel2.getBlue());
  }
```

- Experiments with red-if and blue-if
- Instead of leaves.jpg, use stanford.jpg
- Run without and then with the blue-if code
- Why does it look like that?
- Swap the order, so blue-if is first, then red-if
- Now what does it look like?
- Conclusion: red-if are just tools, we can arrange them
- They just work on whatever data we feed them

Answer:

```
When stanford is put in by red-if, the sky result is just
as blue as the other sky.
All the blue pixels are replaced by the blue-if which
follows.
This only happens if blue-if goes second.
```

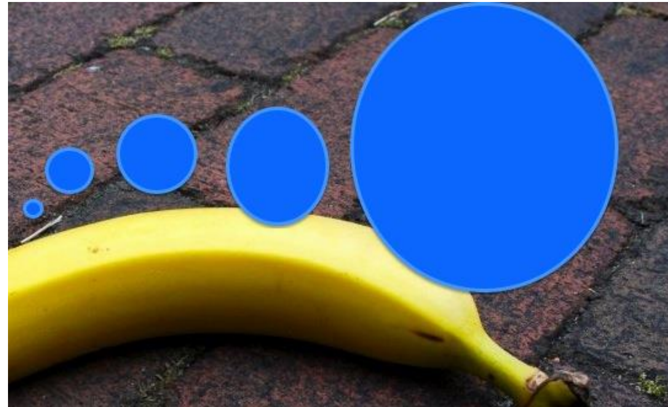
Post-Processing Paint

Here we have banana.jpg

- Suppose we want to show the banana having a dream about paris
- But we don't want to replace all the red brick
- This image is not well suited to bluescreen (banana vs. brick, too much red)
- We can "fix it in post" (Hollywood expression)
- Meaning manipulate the image in a paint program
- I created banana2.jpg as a copy to edit
- Crop the image to position the banana and the eiffel tower
- Put in blue circles artistically (demo)
- Now the code can just look for blue
- Can also use this to "fix" little areas discretely in the original image that are giving your bluescreen selection logic problems (like when you have little speckles you can't get rid of)

On the Mac, Preview has a primitive edit mode. 1. make a copy of your original file first, so you don't mess up the original. Open the copy in Preview. 2 Click the pencil to enter edit mode. 3 Click the circle to draw circles. 4 Click the colors, selecting blue and also a

fill-color of blue. Draw/arrange blue circles or whatever. Windows has a similar simple paint program.



```
image = new SimpleImage("banana2.jpg");
back = new SimpleImage("paris.jpg");
back.setAsBig(image);

for (pixel: image) {
  avg = (pixel.getRed() + pixel.getGreen() + pixel.getBlue())/3;

  if (pixel.getBlue() > avg * 1.3) {  // select blue
    pixel2 = back.getPixel(pixel.getX(), pixel.getY());
    pixel.setRed(pixel2.getRed());
    pixel.setGreen(pixel2.getGreen());
    pixel.setBlue(pixel2.getBlue());
  }

}
print(image);
```