Today: int vs float, int division, function call / parameters, start PyCharm, command line, bluescreen algorithm

These are all on HW2, due next Wed.

Two Math Systems - int and float

You would think computer code has a lot to do with numbers, and here we are.

See Python Guide: Python Math chapter.

Surprisingly, computer code generally uses two different types of numbers - "int" and "float".

int numbers

- The formal type "int" stores an integer
- e.g. `5, 23, -6`
- There is no decimal-point dot in an int
  e.g. `3.14` and `2.6e27` are float values, not int
- Mostly these work intuitively — + - * / work as you would think
- Look at int first, float second
- Math with int inputs .. produces int results
  Except with division /

Math Operators: + - * / **

- int and float support the usual operators `+ - * /`
- Bonus operator: `**` is exponentiation
  The up-hat ^ is not exponentiation
  It's bitwise xor - see CS107
- Standard order of operations: `* /` go before `+ -`
  Known as "precedence" in CS
- Otherwise evaluation goes left-right
- Fine to add parenthesis to force the order you want
- int-inputs → int-outputs (except `/` below)
- Presence of a float value makes the whole expression float

- Python int values do not have a fixed "max"
  Most languages have an int max
- Interpreter: use the up-arrow to edit previous lines - pro tip!

```
>>> 2 + 3
5
>>> 2 + 3 * 4    # Precedence
14
>>> 2 * (1 + 2) # Use parenthesis
6
>>> 10 ** 2      # 10 Squared
100
>>> 2 ** 100     # Grains of sand in universe
1267650600228229401496703205376
>>>
>>>
>>> 1 + 2 * 3     # int in, int out (/)
7
>>> 1 + 2.0 + 3  # any float -> float
6.0
```

float numbers

- "float" numbers `3.14, 1.2e23`
- float numbers have a dot "." when printed
- Surprisingly, float is not totally precise (later topic)
- A float value in a computation forces the whole thing float
- Even if the answer comes out even, a float input forces the result to be float
- Do not pass a float into `get_pixel(x, y)` - will fail with an error

```
>>> 1.5 + 3.0
4.5
>>>
>>> 1.5 * 4.0
6.0
>>>
>>> 6.02e23          # 1 mole
```

```
6.02e+23
>>> 6.02e23 * 2     # 2 moles
1.204e+24
```

Conversions `int()` `float()`

The formal names of the types are "int" and "float". As a handy Python convention, there are functions with those names that try to convert an input to that type. The `int()` function just discards the fractional part of the number.

```
>>> int(3.6)
3
>>> float(3)
3.0
```

Division / → float

- Division `/` is unique
- `a / b` returns a float value always
- Even with int inputs
- Even if the division comes out evenly
- Later lecture: there is a special `//` int-division operator that rounds down

```
>>> 7 / 2
3.5
>>> 8 / 2
4.0
```

What Does Left-to-Right Mean?

With a series of operators of the same precedence, the operators are evaluated from left to right. Consider the following expression. What to subtract from the 4?

```
>>> 4 - 2 + 1
??
```

Python starts with `4`, and does the operations one at a time, left to right - so subtracting `2`, then adding `1`, resulting in `3`.

It's the same with multiplication and division. What is the result of this epxression:

```
>>> 8 / 4 * 2
??
```

Start with `8`, then divide by `4`, then multiply by `2`, resulting in `4.0`.

In this case, the answer is float `4.0` because the division operation `/` produces a float result, even with int inputs.

Why Do We Care - int vs float?

Because of int indexing into structures.

Indexing - `get_pixel(x, y)` - int

- "Indexing" is accessing an element inside something
- Indexing is very common in code - accessing the little thing in the big thing
- Ints are naturally used for indexing
- e.g. `image.get_pixel(x, y)`
  x y numbers like 0 1 2 ..
- There is no pixel at x = 2.5
- Using a float value there raises an error
- Lots of indexing on HW2 - **int**
- The **int-in-int-out** rule works for us on hw2
  Write your expressions with int inputs, get int results

Functions - Parameter / Arguments - Two Forms

We use functions and parameters constantly - today we'll pull back the curtain on how parameters work.

We'll use the word "parameter" for a value within a function call's parenthesis. The word "argument" is another word you'll see for these.

Each function appears in the code in two different forms - the **def** form, and the **call** form. Today we will look at these two in more detail, especially for parameters.

Example: Darker Left

We'll use the darker_left() function as an example of function call and parameters.

> Darker Left

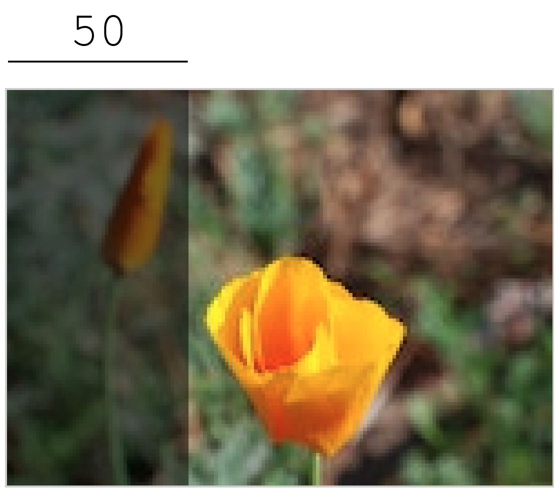
1. Def Form - Name + Parameters

Here is what the def form looks like - we see the function name and its parameters within the parenthesis. There is only one copy of the def for a function, but there can be many calls to that function.

```
def darker_left(filename, left):
        ...
```

This function has two parameters, **filename** and **left**, which are values for it to use when it runs. It loads the image in that filename, and sets a stripe down its left side to be dark, with the width of the stripe specified by the **left** parameter.

So if the function was called and **left** had the value 50, the output image would look like this:

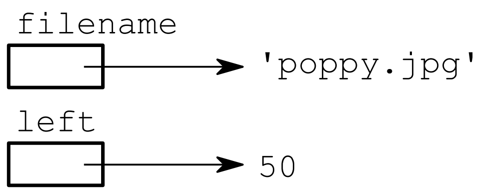


2. How To Write Code That Uses Parameters?

Look at the def again. The parameters are listed within the parenthesis.

To write the code in this function, treat each parameter like a variable that holds a value ready to use. The function code simply uses each parameter, relying on the right value being in there. So in this case `filename` and `left` have the right values in them, the code here just uses them.

The attitude here is one of trust - the code is given `filename` and `left`, and we trust that they are ready to go.



```
def darker_left(filename, left):
 # use filename, left in here
```

Write Code for darker_left()

> Darker Left

```
def darker_left(filename, left):
    # your code here
    # use filename and left
    ...
```

Q: How many pixels on the left to darken?

A: Whatever value is in the `left` parameter.

Key idea: just use the parameters, `filename` and `left` - they are already set up with the values to use. In this case, if left is 50, we want to darken a stripe 50 pixels wide. If left is 2, we darker the a stripe 2 pixels wide. Work out he for/x/range loop to hit that stripe. Note this problem does not create a separate "out" image, so that keeps it simpler.

Solution code

```
def darker_left(filename, left):
    image = SimpleImage(filename)
    for y in range(image.height):
        for x in range(left):
            pixel = image.get_pixel(x, y)
            pixel.red *= 0.5
            pixel.green *= 0.5
            pixel.blue *= 0.5
    return image
```

Summary - writing code for a function, look up at the parameters in the parenthesis. Assume those each have a value and just use them.

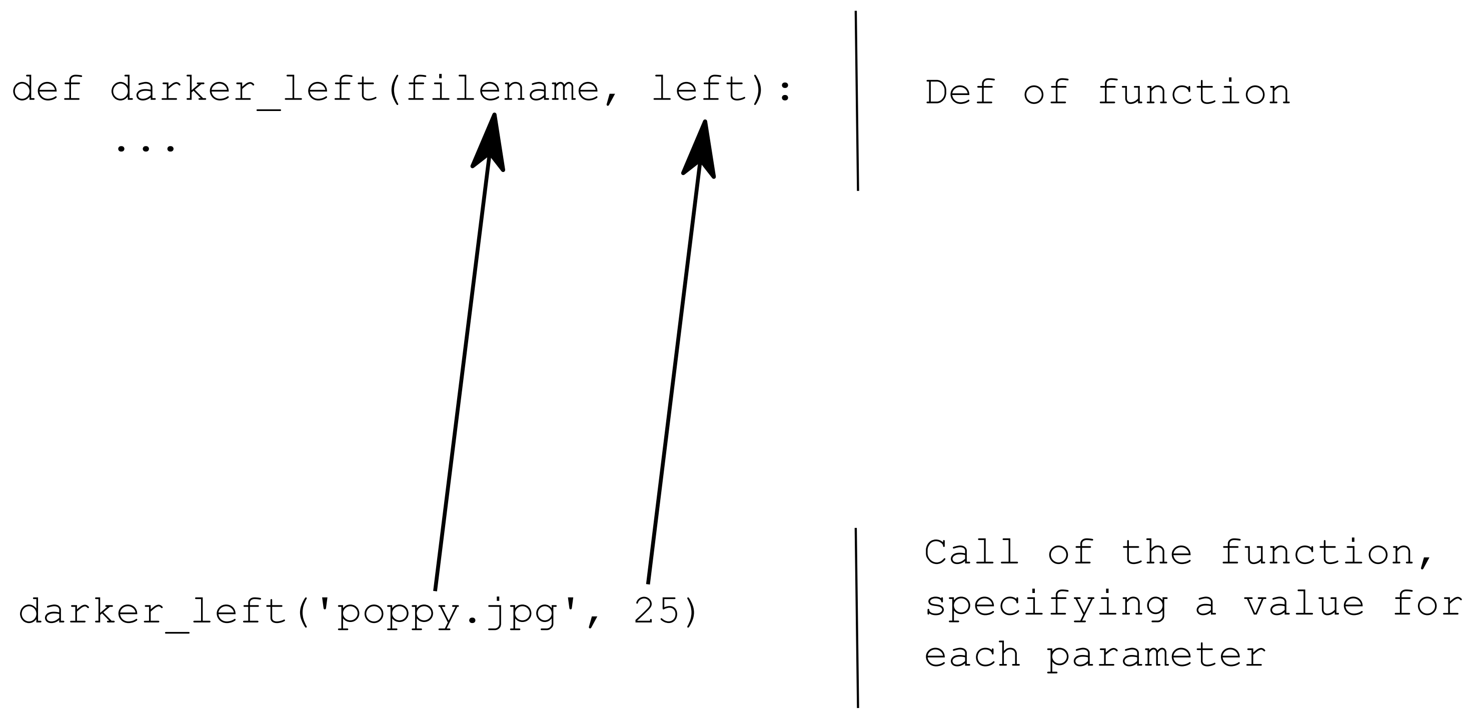
3. Where Do the Parameters Come From?

Where do the parameter values come from? The parameter values come from the function **call** and its parenthesis. The only way a function runs is when a lines calls that function, and the call will specify the values for the parameters.

Function Call Parameters

A **call** of a function is one line, calling the function by name to run it. The call specifies the values for the parameters within the parenthesis.

How does a function run? The only way a function runs is if there is a line somewhere that calls the function, and that line will have the parameter values.



```
def darker_left(filename, left):
    ...
```

Def of function

```
darker_left('poppy.jpg', 25)
```

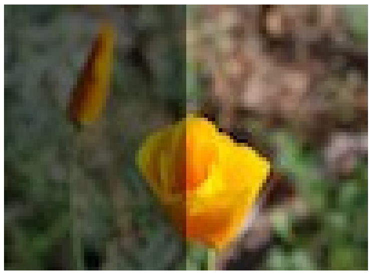
Call of the function, specifying a value for each parameter

darker_left() Example Calls

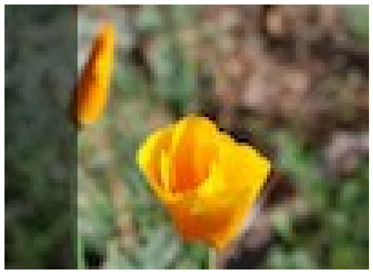
When darker_left() function is called, the parameter values to use are written within the parenthesis. In this case the **filename** parameter gives the name of the file with the image to edit, e.g. `'poppy.jpg'`. The **left** parameter specifies the number of pixels at the left side of the image which should be darkened.

Here are three separate calls to the darker_left() function, and the values for the parameters are in the parenthesis:

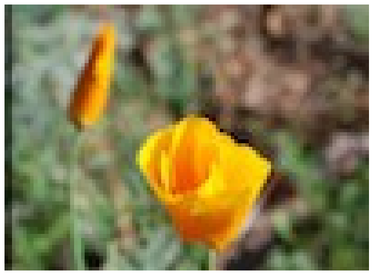
`darker_left('poppy.jpg', 50)` →



`darker_left('poppy.jpg', 20)` →



`darker_left('poppy.jpg', 2)` →



For each call to the function, we say the parameters are "passed in" to the function. So above the first calls passes in 50 for the left parameter and the last call passes in 2.

Look in the Run Menu

Look at the Run Menu on the experimental server for darker_left() carefully. For each case, you can see the syntax of the function call - the name of the function and a pair of parenthesis. Look inside the parenthesis, and you will see the parameters passed in for each run. Click the Run button with different menu selections to see it in action.

Interpreter >>> `foo(a, b)` Example

What do you see here?

```
def foo(a, b):
    ...
    ...
```

It's a function named "foo" (traditional made-up name in CS). It has two parameters, named "a" and "b".

Each parameter is for extra information the caller "passes in" to the function when calling it. What does that look like? Which value goes with "a" and which with "b"?

A call of the foo() looks like this:

```
    ....
    foo(13, 100)
    ...
```

Parameter values match up **by position** within the parenthesis. The first value goes to "a", the second to "b". It is not required that the value has the same name a the parameter.

Call `foo()` in Interpreter

Let's do a little live exercise along these lines.

The hack mode "interpreter", has a ">>>" prompt. You type a line of python code here, hit return. What you typed is sent to python to run, the results are printed after the ">>>".

> Experimental Interpreter >>>

Here we'll enter a little two-line def of foo(a, b) in the interpreter, then try calling it with various values. It's very unusual to define a function within the interpreter like this, but here's one time we'll do it. Normally we define functions inside a .py file.

The most important thing is that at the function call line, the parameter values are pulled from within the parenthesis.

The parameters match by **position**. So the first value goes to "a", the second value goes to "b". The number of parameters must be exactly 2 or there's an error, since foo() takes 2 parameters. The word "argument" is another word for a parameter, which appears in the Python error messages.

Each parameter value can be an **expression** - Python computes the value if needed, then sends that value as the parameter.

Each function has its own variables and parameters, separate from the variables and parameters in other functions, even if the variables have the same name. So an "a" inside a function is different from an "a" variable in some other function. This shows up in the last example below.

```
>>> # Define foo
>>> def foo(a, b):
        print('a:', a, 'b:', b)
>>>
>>> # Call it with 2 param values
>>> foo(1, 2)
a: 1 b: 2
>>> foo(13, 42)
a: 13 b: 42
>>>
>>> # Call with expressions - passes computed value,
e.g. 60
>>> foo(10 * 6, 33 + 2)
a: 60 b: 35
>>>
>>> # Show that number of parameters must be 2
>>> foo(12)
Error:foo() missing 1 required positional argument:
'b'
>>> foo(12, 13, 14)
Error:foo() takes 2 positional arguments but 3 were
given
>>>
>>>
>>> # Parameters do not match by name
>>> # Parameters match by position - first to "a",
second to "b"
>>> b = 10
>>> b
10
>>> foo(b, b + 1)
a: 10 b: 11
>>>
```

The Command Line

The command line is how your computer works "under the hood", and we'll use it in CS106A. Not pretty, but very powerful. We'll use it with a free program called PyCharm - see the PyCharm instructions on the course page.

For details see Python Guide: Command Line

The above guide has instructions to download this example folder: hello.zip

Unzip that folder. Open the folder in PyCharm (not the hello.py, the **folder**). In PyCharm, select "Terminal" at lower left - that's the command-line area.

First we'll type the command "date" into the terminal and hit the return key to run it. The computer runs the "date" program, and shows its output in the terminal, and then gives us another prompt to type more commands.

```
$ date
Fri Jan 14 12:09:24 PST 2022
$
```

Then try the "ls" command ("dir" on old windows). The command line runs in the context of some folder on the computer - "ls" lists the files in the folder.

Run Python Program `hello.py`

The file `hello.py` contains a Python program.

Then try running the hello.py program ("python3" on the Mac, "py" on Windows) with the command shown below. This program takes in a name on the command line, and prints a greeting to that name. It's a simple program, but it shows how programs are run and how to adjust their inputs.

```
$ python3 hello.py Alice
Hello Alice
$ python3 hello.py Bob
Hello Bob
$ python3 hello.py -n 100 Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice Alice
Alice Alice Alice Alice Alice Alice Alice Alice Alice
```

Use the **up arrow** in the command line to retrieve and then edit a previous line. We never type in the command from scratch - use the up-arrow instead. Huge time saver!

When typing at the command line, you are typing commands to your computer operating system - Mac or Windows or Linux. This is different from the ">>>" prompt where you are typing command to **Python**, but the two look similar.

Key Command Line Skills

- See the Python Guide Command Line chapter

- Obtain the command line in a folder - easiest is PyCharm "terminal" tab at window bottom
- Type commands, see output: `ls`, `pwd`, `date`, `cat` *`file`*, `cal`, `cal 2022`
  Windows has two command line systems, PyCharm terminal is the older system:
  In old Windows, use "dir" instead of "ls", "type" instead of "cat", there is no "cal" unfortunately
- Run Python program: `python3 hello.py Hermione`
- Use **up arrow** to recall previous line, edit
- Use **tab** to auto-complete file names

Image Grid Demo

Then we demo command lines from the image-grid homework. They look like this

```
$ python3 image-grid.py -channels poppy.jpg
$
$ python3 image-grid.py -grid poppy.jpg 2
$
$ python3 image-grid.py -random yosemite.jpg 3
```

Those are described in detail in the image-grid homework handout, so you'll see explanations for those when you begin the homework.

Color If Logic

> Logic Stop1

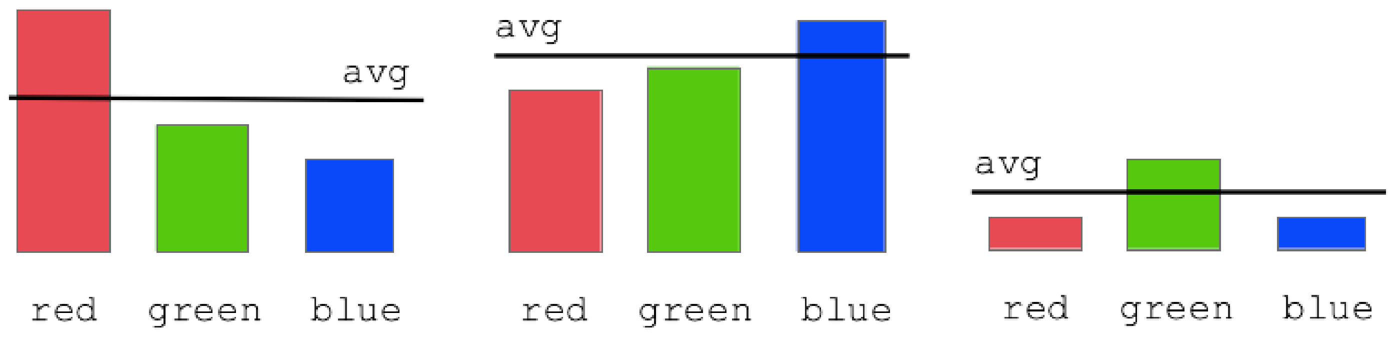
- Say we have this stop sign
- Detect **areas** of an image based on color, e.g. red
- Work out algorithm, Python code to detect a color area
- e.g. detect the red area of this stop sign
- Reminder for colors: rgb-explorer

Red Detect red >= 100

- Try on red_detect1 code example
- First try using red value with >
- Color detection - think about the "hurdle" in the if-test
- e.g. `if pixel.red >= 100:`
- Starts with hurdle value of 100
- Adjust value to get best results
- Q: To make this more selective, make 100 bigger or smaller?
- A: bigger. Look at < - think for a second to get the direction correct
- Does not work that great:
  Bright areas and red areas both have big red numbers
- redish pixel: 220, 50, 50
- white pixel: 220, 220, 220
- Best can do here is maybe hurdle of 160

Red Detect - Average - Stop2

> Logic Stop2



```
# compute average number for a pixel
average = (pixel.red + pixel.green + pixel.blue) / 3
```

- Try on red_detect2 code example
- Improvement: compare red value to **average** of red/green/blue values
- "redish pixel"
  the red value is above the average of this pixel
- "blueish pixel"
  the blue value is above the average of this pixel
- `red >= average * 1.0`
- Adjust the 1.0 "hurdle" factor by eye, try 1.0 1.1 1.2 .. find a good value
- The average technique works great to select the red stop sign pixels
- Experiment: for detected pixels, reduce only red and green, leaving blue unchanged
  result is blue stop sign
- Experiment: for detected pixels, swap red and green values

Bluescreen Algorithm

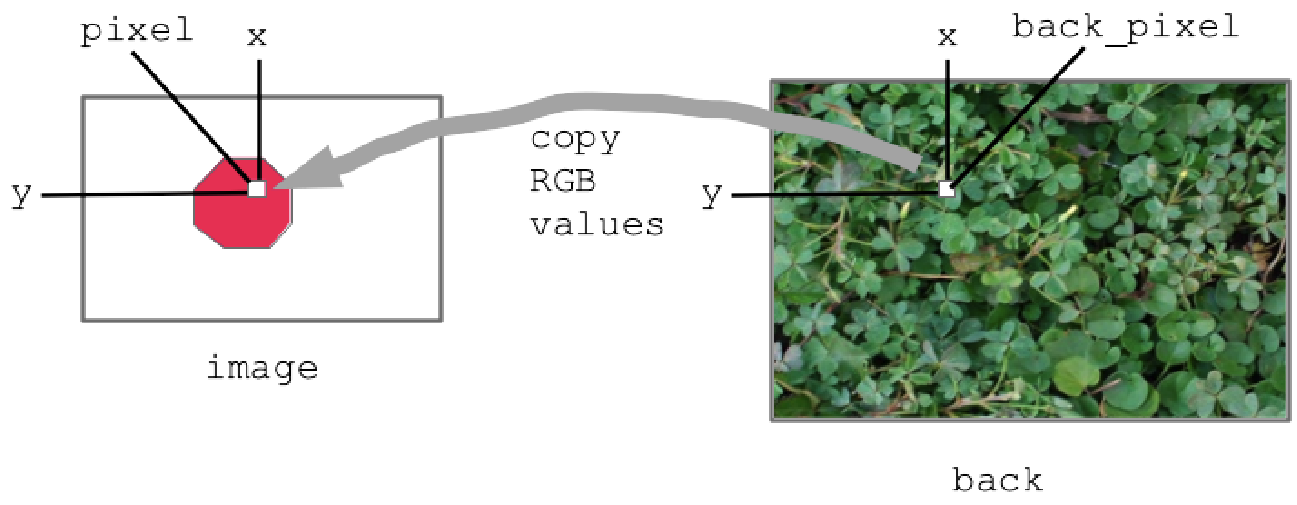
- Demo Google search: blue screen movie image
- Also known as Chroma Key (wikipedia)
- Video is just a series of still images, 20-60 per second
- This is the "front" approach, replacing colored pixels in the front image
- Have **image** with special color
- Have **back** background image
- 1. **Detect** colored area in an image, e.g. the blue area

- 2. **Replace** colored pixels with pixels from back image
- 3. The final output is the front image with the replacements done

Bluescreen Algorithm Outline

- Two images we'll call **image** and **back**
- Detect, say, red pixels in image
- For each red pixel (make a drawing)
  Consider the pixel at the same x,y in back image
  Copy that pixel from back to image
  i.e. copy RGB numbers from back pixel to image pixel
- Result: for red areas, copy over areas from back image
- Adjacent pixels in back are still adjacent in new image, so it looks right

Diagram:



Bluescreen Stop Sign Example

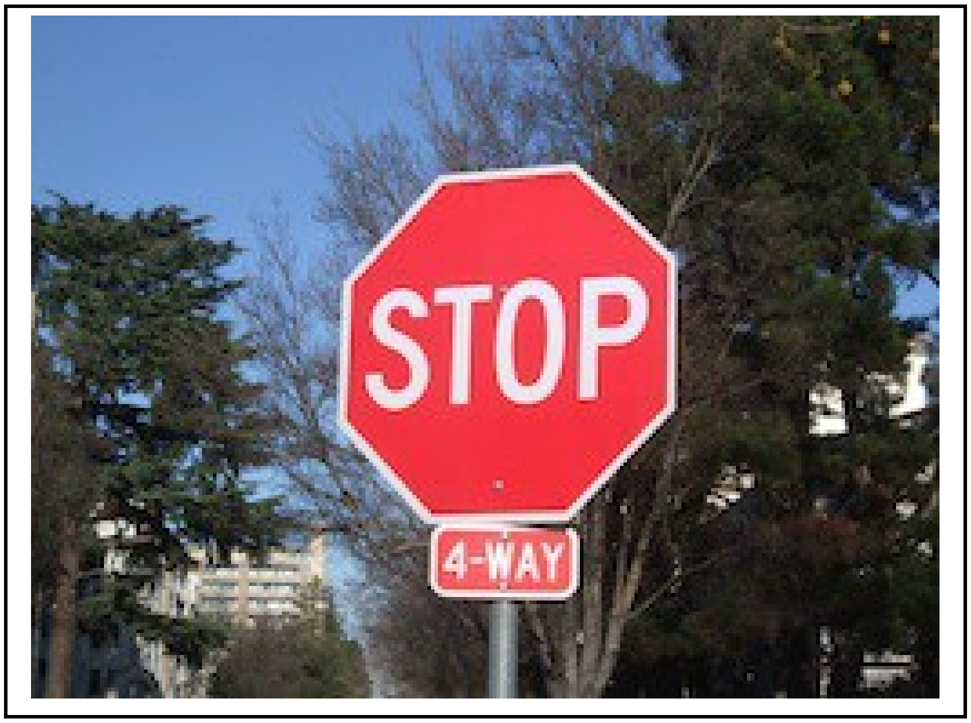
This code is complete, look at the code then run it to see.

> Bluescreen Stop Sign

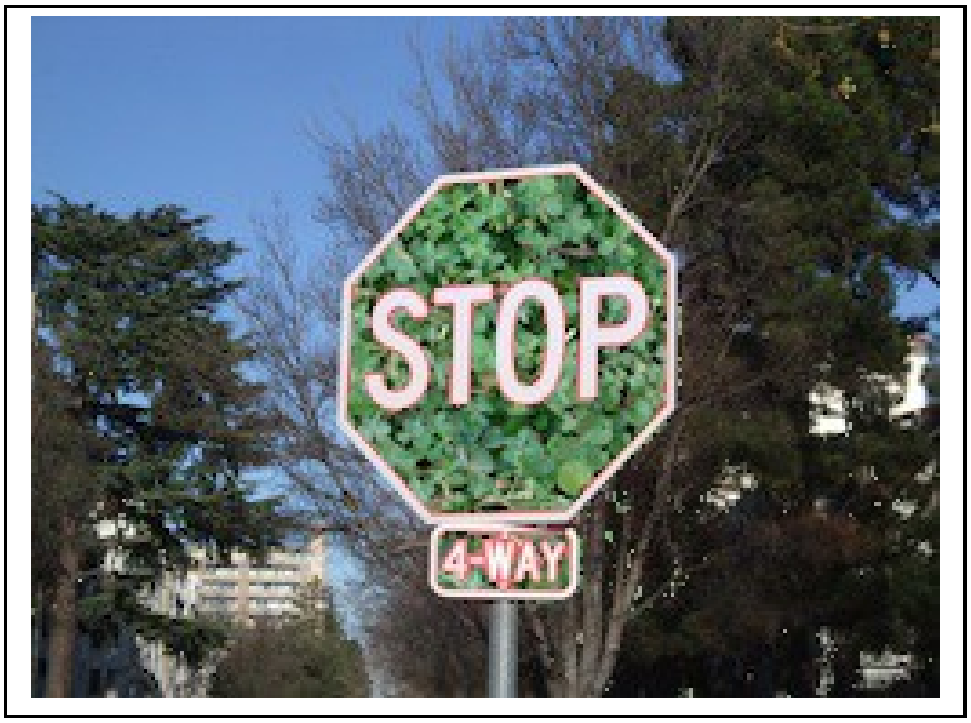
Solution code

```
def stop_leaves(front_filename, back_filename):
    """Implement stop_leaves as above."""
    image = SimpleImage(front_filename)
    back = SimpleImage(back_filename)
    for y in range(image.height):
        for x in range(image.width):
            pixel = image.get_pixel(x, y)
            average = (pixel.red + pixel.green + pixel.blue) / 3
            if pixel.red >= average * 1.4:
                # the key line:
                back_pixel = back.get_pixel(x, y)
                pixel.red = back_pixel.red
                pixel.green = back_pixel.green
                pixel.blue = back_pixel.blue
    return image
```

Before - the red stop sign before the bluescreen algorithm:



After:

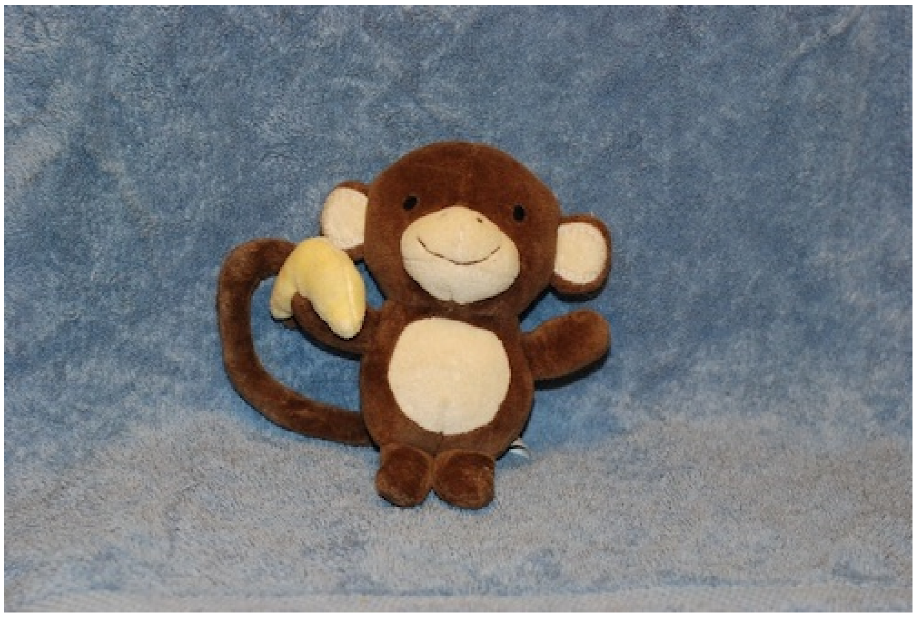


Bluescreen Monkey

> Bluescreen Monkey

A favorite old example of Nick's.

Have monkey.jpg with blue background



The famous Apollo 8 moon image. At one time the most famous image in the world. Taken as the capsule came around the moon, facing again the earth. Use this as the background.

The bluescreen code is the same as before basically. Adjust the hurdle factor to look good. Observe: the bluescreen algorithm depends on the colors in the main image. BUT it does not depend on the colors of the back image - the back image can have any colors it in. Try the stanford.jpg etc. background-cases for the monkey.

The code is complete but has a 1.5 factor to start. Adjust it, so more blue gets replaced, figuring out the right hurdle value.

This is the "front" strategy - replacing blue pixels in the front image, then the front image is the final output. There is also a "back" strategy on the HW2c handout which you have as an option.