Week 8 - Wednesday

# COMP 1800

# Last time

- What did we talk about last time?
- Namespaces
- Before that:
  - Color and images
  - **`Pixel`** class
  - **`Image`** class

# Questions?

# Assignment 6

# Function Variables

# Types

- What Python types have we talked about so far?
  - Integers
  - Floating-point values (decimals)
  - Strings
  - Booleans
  - Lists
  - Dictionaries
- Any of these types can be held in a variable

# Putting a function in a variable

- What if what we wanted to store wasn't a value but was an **action** instead?
- We can store **functions** into variables
- All you have to do is use the name of the function without the parentheses

```
import math

action = math.sqrt # no parentheses, just the name
print(math.sqrt(5)) # prints square root of 5
print(action(5)) # also prints square root of 5
```

# Why would we want to do that?

- One of the goals of computer science is reusing code that we already have
- Maybe we've written code that processes everything in a list
- But we can customize how we process it
- For example, recall the function that adds up everything in a list

# Summing everything in a list

- This function will sum everything in a list

```
def total(values):
    result = 0
    for value in values:
        result = result + value
    return result
```

# But what if I don't want to add them?

- This function multiplies everything in a list

```
def product(values):
    result = 1
    for value in values:
        result = result * value
    return result
```

- There are two differences from the previous slide:
  - The starting value of **`result`** is 1 here instead of 0
  - We multiply instead of add each value

# We can make a function that does anything

- This function will apply any function (called **`action`**) to everything in the list, with a given starting value

```
def process(values, action, starting):
    result = starting
    for value in values:
        result = action(result, value)
    return result
```

# Let's make a few actions

- These functions are functions we can use with process
- One adds two numbers, and the other multiplies them

```
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

# Using our actions

- Now we can call **process** with the actions we defined

```
numbers = [3, 4, 9, 2, 1, 7]
total = process(numbers, add, 0) # starts at 0
product = process(numbers, multiply, 1) # starts at 1
```

- We can even use a built-in function like **max**

```
largest = process(numbers, max, numbers[0])
```

# Applications in Image Processing

# To use Pixel

- To create a custom color:

```
color = Pixel(255,165,0) # orange
green = color.getGreen()
```

- Create colors using **Pixel** to specify **RGB** values
- Get individual values using:
  - **getRed()**
  - **getGreen()**
  - **getBlue()**

# Image methods

| Method | Use |
|---|---|
| `FileImage(file)` | Creates an **Image** object from a file name |
| `EmptyImage(width, height)` | Creates a blank **Image** of size **width** by **height** |
| `getWidth()` | Return the width of the image |
| `getHeight()` | Return the height of the image |
| `getPixel(x, y)` | Return the **Pixel** which is the color at (**x**,**y**) |
| `setPixel(x, y, pixel)` | Set the **Pixel** object at (**x**,**y**) to **pixel** |
| `save(file)` | Save the **Image** to the file with the given file name |

# Pixel mapper

- The book observes that many operations need to visit every column and row of an image and transform the pixels in that location
- To generalize this process, we can make a pixel mapper function that takes a function parameter that says how each pixel should be transformed
- This function will:
  - Get the width and height of the current image
  - Make a new image that's the same size
  - Visit every column of the old image
    - Visit every row in that column
      - Put a new pixel into the new image by transforming the old pixel
  - Return the new image

# Pixel mapper function

```
def pixelMapper(image, transform):
    width = image.getWidth()
    height = image.getHeight()
    newImage = EmptyImage(width, height)
    for x in range(image.getWidth()):
        for y in range(image.getHeight()):
            pixel = image.getPixel(x, y)
            newPixel = transform(pixel)
            newImage.setPixel(x, y, newPixel)
    return newImage
```

# Using the pixel mapper

- In order to use the pixel mapper, we have to write function that will transform pixel values
- Here's one that makes the photo negative of a pixel:

```
def negative(pixel):
    red = pixel.getRed()
    green = pixel.getGreen()
    blue = pixel.getBlue()
    return Pixel(255 - red, 255 - green, 255 - blue)
```

# Making a negative image

- Once we have both the **`pixelMapper`** and **`negative`** functions, we can use them together to create a negative version of any image

```
newImage = pixelMapper(image, negative)
```

# Brightening

- Let's write a function we can use with **`pixelMapper`** that will multiply the red, green, and blue values of a pixel by a factor of two
- If the value gets larger than 255, limit it to 255

```
def brighten(pixel)
```

# Grayscale

- Let's write a function we can use with **`pixelMapper`** that will create a grayscale version of a pixel
- Remember that a color is grayscale if its red, green, and blue values are all the same
- We can find out what that value as follows:
  - $value = .3 \cdot red + .59 \cdot green + .11 \cdot blue$
- Then, the final color is $(value, value, value)$

```
def grayscale(pixel)
```

# Wacky colors

- Colors will look strange if the red, green, and blue components get swapped around
- Let's write a function that will make:
  - The new red the old green
  - The new green the old blue
  - The new blue the old red

```
def wacky(pixel)
```

# Applying our changes

- With our functions written, we can apply all three of them individually to make an image that is negative, brightened, and then with wacky colors

```
image = pixelMapper(image, negative)
Image = pixelMapper(image, brighten)
image = pixelMapper(image, wacky)
```

# Limitations

- As powerful as this technique is, it can only change a single pixel at a time
- It can't change the size or shape of the image, since it depends on the new image being the same size as the old one
- It has no memory of past pixels and can't predict future ones

# Quiz

# Upcoming

# Next time…

- Work time for Assignment 6
- Cryptanalysis on Monday

# Reminders

- Read 8.2 and 8.3 for Monday
- **Finish Assignment 6**
  - **Due Friday before midnight!**