Manipulating Pixels by Range and More on Functions

Remember that pixels are in a matrix

- Matrices have two dimensions: A height and a width
- We can reference any element in the matrix with (x,y) or (horizontal, vertical)
  - We refer to those coordinates as *index numbers* or *indices*
- We sometimes want to know *where* a pixel is, and **getPixels** doesn't let us know that
  - Not to mention the bug that leaves out the first row and column

Tuning our color replacement

- If you want to get more of Barb's hair, just increasing the threshold doesn't work
  - Wood behind becomes within the threshold value
- How could we do it better?
  - Lower our threshold, but then miss some of the hair
  - Work only within a range…

Introducing the function range

- Range returns a sequence between its first two inputs, possibly using a third input as the increment

```
>>> print range(1,4)
[1, 2, 3]
>>> print range(-1,3)
[-1, 0, 1, 2]
>>> print range(1,10,2)
[1, 3, 5, 7, 9]
```

That thing in [] is a sequence

```
>>> a=[1,2,3]
>>> print a
[1, 2, 3]
>>> a = a + 4
An attempt was made to call a function with a parameter of an invalid type
>>> a = a + [4]
>>> print a
[1, 2, 3, 4]
>>> a[0]
1
```

We can assign names to sequences, print them, add sequences, and access individual pieces of them.

We can also use **for** loops to process each element of a sequence.

We can use *range* to generate index numbers

- We'll do this by working the range from 1 to the height, and 1 to the width
- But we'll need more than one loop.
  - Each for loop can only change one variable,
    and we need two for a matrix

Working the pixels by number

- To use **range**, we'll have to use *nested loops*
  - One to walk the width, the other to walk the height

```
def increaseRed2(picture):
  for x in range(1, getWidth(picture)):
    for y in range(1, getHeight(picture)):
      px = getPixel(picture,x,y)
      value = getRed(px)
      setRed(px, value*1.1)
```

Bug Alert:
Be sure to watch your blocks carefully!

Missing any pixels?

What's going on here?

The first time through the first loop, x is the name for 1.

We'll be processing the first column of pixels in the picture.

```
def increaseRed2(picture):
  for x in range(1,getWidth(picture)):
    for y in range(1,getHeight(picture)):
      px = getPixel(picture,x,y)
      value = getRed(px)
      setRed(px,value*1.1)
```

Now, the inner loop

Next, we set y to 1. We're now going to process each of the pixels in column 1.

```
def increaseRed2(picture):
  for x in range(1,getWidth(picture)):
    for y in range(1,getHeight(picture)):
      px = getPixel(picture,x,y)
      value = getRed(px)
      setRed(px,value*1.1)
```

Process a pixel

With x = 1 and y = 1, we get the leftmost pixel and increase its red by 10%

```
def increaseRed2(picture):
  for x in range(1,getWidth(picture)):
    for y in range(1,getHeight(picture)):
      px = getPixel(picture,x,y)
      value = getRed(px)
      setRed(px,value*1.1)
```

Next pixel

Next we set y to 2 (next value in the sequence *range(1,getHeight(picture))*

```
def increaseRed2(picture):
  for x in range(1, getWidth(picture)):
    for y in range(1, getHeight(picture)):
      px = getPixel(picture,x,y)
      value = getRed(px)
      setRed(px,value*1.1)
```

Process pixel (1,2)

x is still 1, and now y is 2, so increase the red for pixel (1,2)

```
def increaseRed2(picture):
  for x in range(1, getWidth(picture)):
    for y in range(1, getHeight(picture)):
      px = getPixel(picture, x, y)
      value = getRed(px)
      setRed(px,value*1.1)
```

We continue along this way, with y taking on every value from 1 to the height of the picture.

Finally, next column

Now that we're done with the loop for y, we get back to the for loop for x.

x now takes on the value 2, and we go back to the y loop to process all the pixels in the column x=2.

```
def increaseRed2(picture):
  for x in range(1, getWidth(picture)):
    for y in range(1, getHeight(picture)):
      px = getPixel(picture, x, y)
      value = getRed(px)
      setRed(px,value*1.1)
```

Replacing colors in a range

Get the range using MediaTools

```
def turnRedInRange():
  brown = makeColor(57,16,8)
  file=r"C\Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
  for x in range(70,168):
    for y in range(56,190):
      px=getPixel(picture,x,y)
      color = getColor(px)
      if distance(color,brown)<50.0:
        redness=getRed(px)*1.5
        setRed(px,redness)
  show(picture)
  return(picture)
```

Walking this code

- Like last time:
  - Don't need input
  - same color we want to change
  - same file
- make a picture

```
def turnRedInRange():
  brown = makeColor(57,16,8)
  file=r"C\Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
  for x in range(70,168):
    for y in range(56,190):
      px=getPixel(picture,x,y)
      color = getColor(px)
      if distance(color,brown)<50.0:
        redness=getRed(px)*1.5
        setRed(px,redness)
  show(picture)
  return(picture)
```

The nested loop

- Used MediaTools to find the rectangle where most of the hair is that we want to change

```
def turnRedInRange():
  brown = makeColor(57,16,8)
  file=r"C\Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
  for x in range(70,168):
    for y in range(56,190):
      px=getPixel(picture,x,y)
      color = getColor(px)
      if distance(color,brown)<50.0:
        redness=getRed(px)*1.5
        setRed(px,redness)
  show(picture)
  return(picture)
```

Scanning for brown hair

- We're looking for a close-match on hair color, and increasing the redness

```
def turnRedInRange():
  brown = makeColor(57,16,8)
  file=r"C\Documents\mediasources\barbara.jpg"
  picture=makePicture(file)
  for x in range(70,168):
    for y in range(56,190):
      px=getPixel(picture,x,y)
      color = getColor(px)
      if distance(color, brown) < 50.0:
        redness=getRed(px)*1.5
        setRed(px,redness)
  show(picture)
  return(picture)
```

Similar to scanning whole picture

We could raise threshold now. (Why?...)

Could we do this without nested loops?

- Yes, but only with a complicated **if** statement
- Moral: Nested loops are common for 2D data

```
def turnRedInRange2():
  brown = makeColor(57,16,8)
  file=r"C:\Documents \mediasources\barbara.jpg"
  picture=makePicture(file)
  for p in getPixels(picture):
    x = getX(p)
    y = getY(p)
    if x >= 70 and x < 168:
      if y >=56 and y < 190:
        color = getColor(p)
        if distance(color,brown)<100.0:
          redness=getRed(p)*2.0
          setRed(p,redness)
  show(picture)
  return picture
```

Review and more on Functions

- How can we reuse variable names like **picture** in both a function and in the Command Area?
- Why do we write the functions like this? Would other ways be just as good?
- Is there such a thing as a better or worse function?
- Why don't we just build in calls to **pickAFile** and **makePicture?**

One and only one thing

- We write functions as we do to make them *general* and *reusable*
  - Programmers hate to have to rewrite something they've written before
  - They write functions in a general way so that they can be used in many circumstances.
- What makes a function *general* and thus *reusable*?
  - A reusable function does *One and Only One Thing*

Compare these two programs

```
def makeSunset(picture):
    for p in getPixels(picture):
        value=getBlue(p)
        setBlue(p, value*0.7)
        value=getGreen(p)
        setGreen(p, value*0.7)
```

Yes, they do exactly the same thing!

makeSunset(somepict) has the same effect in both cases

```
def makeSunset(picture):
    reduceBlue(picture)
    reduceGreen(picture)

def reduceBlue(picture):
    for p in getPixels(picture):
        value=getBlue(p)
        setBlue(p, value*0.7)

def reduceGreen(picture):
    for p in getPixels(picture):
        value = getGreen(p)
        setGreen(p, value*0.7)
```

Observations on the new makeSunset

- It's normal to have more than one function in the same Program Area (and file)
- makeSunset in this one is somewhat easier to read.
  - It's clear what it does "reduceBlue" and "reduceGreen"

```
def makeSunset(picture):
    reduceBlue(picture)
    reduceGreen(picture)

def reduceBlue(picture):
    for p in getPixels(picture):
        value = getBlue(p)
        setBlue(p, value*0.7)

def reduceGreen(picture):
    for p in getPixels(picture):
        value = getGreen(p)
        setGreen(p, value*0.7)
```

Programs are read by people, not computers!

Considering variations

- We can only do this because **reduceBlue** and **reduceGreen**, do *one and only one thing*.
- If we put **pickAFile** and **makePicture** in them, we'd have to pick a file twice (better be the same file), make the picture—then save the picture so that the next one could get it!

```
def makeSunset(picture):
    reduceBlue(picture)
    reduceGreen(picture)

def reduceBlue(picture):
    for p in getPixels(picture):
        value = getBlue(p)
        setBlue(p, value*0.7)

def reduceGreen(picture):
    for p in getPixels(picture):
        value = getGreen(p)
        setGreen(p, value*0.7)
```

Does makeSunset do *one and only one thing*?

- Yes, but it's a higher-level, *more abstract* thing.
  - It's built on lower-level *one and only one thing*
- We call this *hierarchical decomposition.*
  - You have some *thing* that you want the computer to do?
  - Redefine that *thing* in terms of smaller *things*
  - Repeat until you know how to write the smaller things
  - Then write the larger things in terms of the smaller things.

What happens when we use a function

- When we type in the Command Area

```
>>> makeSunset(picture)
```

Whatever object that is in the *Command Area* variable **picture** becomes the value of the *placeholder (input) variable* **picture** in

```
def makeSunset(picture):
    reduceBlue(picture)
    reduceGreen(picture)
```

makeSunset's picture is then passed as input to **reduceBlue** and **reduceGreen**, but their input variables are completely different from **makeSunset**'s picture.

- For the life of the functions, they are the same *values (picture objects*)

Names have contexts

- In natural language, the same word has different meanings depending on *context*.
  - Time flies like an arrow
  - Fruit flies like a banana
- A function is its *own* context.
  - Input variables (*placeholders*) take on the value of the input values *only for the life of the function*
    - Only while it's executing
  - Variables defined within a function also only exist within the context of that function
  - The context of a function is also called its *scope*

Parameters are placeholders

- Think of the input variable, i.e. parameter, as a placeholder
  - It takes the place of the input object
- During the time that the function is executing, the placeholder variable *stands for* the input object.
- When we modify the placeholder by changing its pixels with **setRed**, we actually change the input object.

Input variables as placeholders (example)

- Imagine we have a wedding computer

```
def marry(husband, wife):
    sayVows(husband)
    sayVows(wife)
    pronounce(husband, wife)
    kiss(husband, wife)
```

```
>> marry("Tom Cruise","Katie Holmes")
```

```
def sayVows(speaker):
    print "I, " + speaker + " blah blah"

def pronounce(man, woman):
    print "I now pronounce you…"

def kiss(p1, p2):
    if p1 == p2:
        print "narcissism!"
    if p1 <> p2:
        print p1 + " kisses " + p2
```

Variables within functions *stay* within functions

- The variable **value** in **decreaseRed** is created *within* the scope of **decreaseRed**
  - That means that it only exists while decreaseRed is executing
- If we tried to *print value* after running decreaseRed, it would work *ONLY* if we already had a variable defined in the Command Area
  - The name *value* within *decreaseRed* doesn't exist outside of that function
  - We call that a *local* variable

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value = getRed(p)
    setRed(p, value*0.5)
```

Writing *real* functions

- Functions in the mathematics sense take input and usually return *output.*
  - Like ord(character) or makePicture(file)
- What if you create something inside a function that you *do* want to get back to the Command Area?
  - You can **return** it

```
def computeAverage(num1, num2, num3):
  ave = (num1 + num2 + num3) / 3
  return ave


>> x = computeAverage(10,20,30)
```

Consider these two functions

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value = getRed(p)
    setRed(p, value*0.5)
```

```
def decreaseRed(picture, amount):
  for p in getPixels(picture):
    value = getRed(p)
    setRed(p, value * amount)
```

- It is common to have *multiple* inputs to a function.
- The new **decreaseRed** now takes an input of the multiplier for the red value.
  - **decreaseRed(picture, 0.5)** would do the same thing
  - **decreaseRed(picture, 1.25)** would *increase* red 25%

Names are important

- This function should probably be called **changeRed** because that's what it does.
- Is it more general?
  - Yes.
- But is it the one and only one thing that you need done?
  - If not, then it may be less understandable.
  - You can be *too* general

```
def decreaseRed(picture, amount):
  for p in getPixels(picture):
    value = getRed(p)
    setRed(p, value*amount)
```

```
def changeRed(picture, amount):
  for p in getPixels(picture):
    value = getRed(p)
    setRed(p, value * amount)
```

Always make the program easy to understand *first*

- Write your functions so that *you* can understand them *first*
  - **Get your program *running***
- ***ONLY THEN*** should you try to make them better
  - Make them more understandable to other people
    - Another programmer (or you in six months) may not remember or be thinking about increase/decrease functions
  - Make them more efficient
    - The new version of **makeSunset** i.e. the one with **reduceBlue** and **reduceGreen**) takes twice as long as the first version, because it changes all the pixels *twice*
    - But it's easier to understand and to get working in the first place

Removing "Red Eye"

- When the flash of the camera catches the eye just right (especially with light colored eyes), we get bounce back from the back of the retina.
- This results in "red eye"
- We can replace the "red" with a color of our choosing.
- Find *where* the eyes are (x, y) using MediaTools

Removing Red Eye

```
def removeRedEye(pic, startX, startY, endX, endY,
    replacementColor):
  red = makeColor(255, 0, 0)
  for x in range(startX, endX):
    for y in range(startY, endY):
      currentPixel = getPixel(pic, x, y)
      if (distance(red, getColor(currentPixel)) < 165):
        setColor(currentPixel, replacementColor)
```

By specifying bounds of eye as parameters makes this work on any picture

Why use a range? Because we don't want to replace her red dress!

What we're doing here:

- Within the rectangle of pixels (startX, startY) to (endX, endY)
- Find pixels close to red, then replace them with a new color *replacementColor*

"Fixing" it: Changing red to black

removeRedEye(jenny, 109, 91, 202, 107, makeColor(0,0,0))

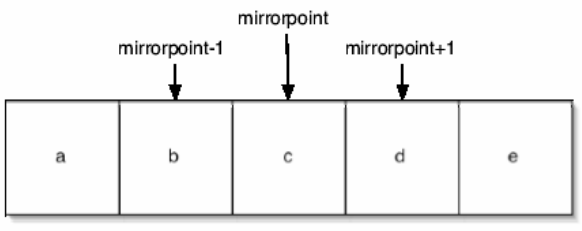
- Jenny's eyes are actually not black
  - could fix that
- Eye are also not mono-color
  - A better function would handle *gradations* of red and replace with *gradations* of the correct eye color

If you know where the pixels are: Mirroring

- Imagine a mirror horizontally across the picture,
  or vertically
- What would we see?
- How do generate that digitally?
  - We simply *copy* the colors of pixels from one place to another

Mirroring a picture

- Slicing a picture down the middle and sticking a mirror on the slice
- Do it by using a loop to measure an *offset*
  - The index variable is actually measuring an offset from the *mirror point*
- Then reference to either side of the mirror point using the offse



Recipe for mirroring

```
def mirrorVertical(source):
    mirrorpoint = int(getWidth(source) / 2)
    for y in range(1, getHeight(source)):
        for xOffset in range(1, mirrorpoint):
            pright = getPixel(source, xOffset + mirrorpoint, y)
            pleft = getPixel(source, mirrorpoint - xOffset, y)
            c = getColor(pleft)
            setColor(pright, c)
```

How does it work?

- Compute the half-way horizontal index
- The y value travels the height of the picture
- The xOffset value is an *offset*
  - **It's not actually an index**
  - **It's the amount to add or subtract**
- We copy the color at mirrorpoint - offset to mirrorpoint + offset

int converts value in parens to integer (2.5 becomes 2)

```
def mirrorVertical(source):
    mirrorpoint = int(getWidth(source) / 2)
    for y in range(1, getHeight(source)):
        for xOffset in range(1, mirrorpoint):
            pright = getPixel(source, xOffset + mirrorpoint, y)
            pleft = getPixel(source, mirrorpoint - xOffset, y)
            c = getColor(pleft)
            setColor(pright, c)
```

Can we do this with a horizontal mirror?

```
def mirrorHorizontal(source):
    mirrorpoint = int(getHeight(source) / 2)
    for yOffset in range(1, mirrorpoint):
        for x in range(1, getWidth(source)):
            pbottom = getPixel(source, x, yOffset + mirrorpoint)
            ptop = getPixel(source, x , mirrorpoint - yOffset)
            setColor(pbottom, getColor(ptop))
```

Of course!

What if we wanted to copy bottom to top?

- Very simple: Swap the **order of pixels** in the bottom line

```
def mirrorHorizontal(source):
    mirrorpoint = int(getHeight(source) / 2)
    for yOffset in range(1, mirrorpoint):
        for x in range(1, getWidth(source)):
            pbottom = getPixel(source, x, yOffset + mirrorpoint)
            ptop = getPixel(source, x , mirrorpoint - yOffset)
            setColor(ptop, getColor(pbottom))
```

Set color this way, instead of this

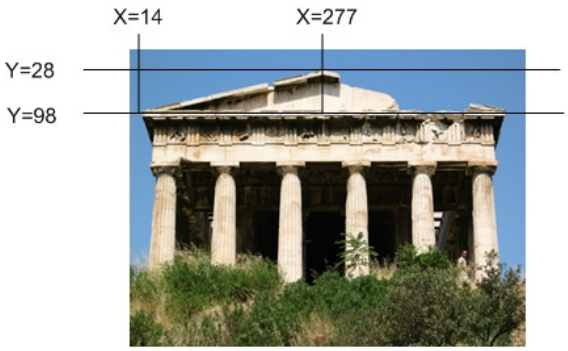
```
setColor(pbottom, getColor(ptop))
```

Doing correction with mirroring

- Mirroring can be used to create interesting effects, but it can also be used to create realistic effects.
- Consider this image that M.G. took on a trip to Athens, Greece.
  - Can we "repair" the temple by mirroring the complete part onto the broken part?

Figuring out where to mirror

- Use MediaTools to find the mirror point and the range that we want to copy



Writing a function for specific file

- The function to mirror the temple needs to work for one and only one file.
- But we still don't want to write out the whole path.
  - setMediaPath() allows us to pick a directory where our media will be stored.
  - getMediaPath(filename) will *generate* the entire path for us to the filename *in the media directory*
  - *THIS ONLY WORKS WHEN WE'RE ACCESSING FILES IN THE MEDIA DIRECTORY AND WHERE WE HAVE SET THE PATH FIRST DURING OUR SESSION WITH JES!*

Program to mirror the temple

```
def mirrorTemple():
  source = makePicture(getMediaPath("temple.jpg"))
  mirrorpoint = 277
  lengthToCopy = mirrorpoint - 14
  for x in range(1, lengthToCopy):
    for y in range(28, 98):
      p1 = getPixel(source, mirrorpoint - x, y)
      p2 = getPixel(source, mirrorpoint + x, y)
      setColor(p2, getColor(p1))
  return source
```

Did it really work?

- It clearly did the mirroring, but that doesn't create a 100% realistic image.
- Check out the shadows: Which direction is the sun coming from?