More About Formatting
Image Processing in Python

CS 8: Introduction to Computer Science
Lecture #14

Ziad Matni
Dept. of Computer Science, UCSB

Administrative

Homework assignment #7 is due ~~Thursday (6/1)~~ Tuesday (6/6)

Project #2 is due ~~Tuesday, 6/6~~ Thursday (6/8)

NEW Lab assignment #7 is due on Friday (6/2)

Administrative

- 4 MORE CLASSES TO GO! ☺

** = Due date has changed*

M	T	W	Th	F
5/29	**5/30** **LECTURE 14**	**5/31** LAB 7 issued	**6/1** **LECTURE 15** HW8 issued	**6/2** LAB 7 due
6/5	**6/6** **LECTURE 16** HW7 due*	**6/7** Work on your Project2 in lab	**6/8** **REVIEW** HW8 due Project2 due*	**6/9** *Last day of Spring classes at UCSB*
6/12	**6/13**	**6/14**	**6/15** **FINAL EXAM at 4PM**	**6/16**

Formatting Strings in Python

- Recall the "%" method: called a **conversion specifier**
- Example:

```
>>> print("These %d bottles of beer on the wall" % 99)
'These 99 bottles of beer on the wall'

>>> n = 4
>>> cost = 0.5
>>> print ("%d items, each for $%f3.2" % (n, cost))
'4 items, each for $0.50'
```

- Also `%f`, `%e`, `%g` for float; `%s` for string; and more – **see Tables 5.2 and 5.3 (p. 162) in textbook**
  - Can specify field width, left or right justify, other

New Way: Using `.format`

- Similar ideas, different syntax:
  `template.format(p0,p1,...,k0=v0,k1=v1,...)`
  - **`template`** is a string with ***conversion specifiers*** enclosed in **curly braces**
  - **`p0, p1,`** etc… are ***positional arguments*** and the **`k=v`** pairs are ***keyword arguments***

```
>>> "{1} has ${0:.2f}".format(42,'Jo')
'Jo has $42.00'
```

  - All same conversion specifiers as old way

New Way: Using `.format`

- Keyword arguments are handy, esp. if lots of args

```
>>> "{0} is {age}".format('Ed',age=20)
'Ed is 20'
```

- More information can be found at:

http://www.python-course.eu/python3_formatted_output.php

(but, alas, not in our text book)

THIS WEBSITE MATERIAL IS REQUIRED READING

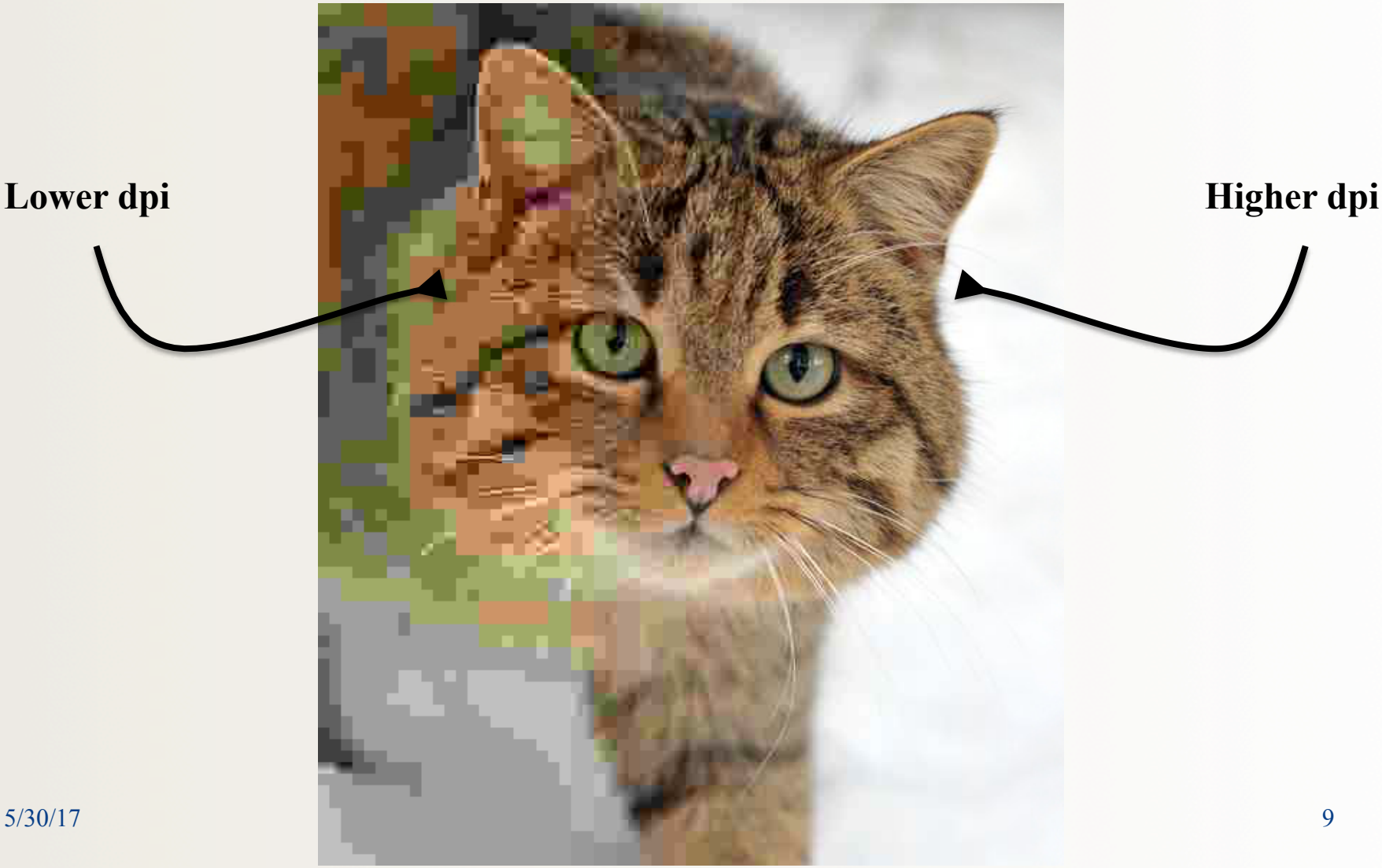
Starting Chapter 6

Digital Images on Computers

- Two types of images: **raster** vs. **vector**

- **Raster** (a.k.a "bit-map") images
  - Most picture formats from photos, paint/shop programs
  - Typically **JPEG** (**.jpg**, **.jpeg**) types
  - Made of a finite number of **pixels** (or **dots**)
    - Quality of picture is measured in **dots per inch** (**dpi**)
    - Close-ups look blurry or "pixelated"
  - The higher the resolution, the more pixels are needed
    - More pixels mean larger file sizes to store the image
  - Raster images are a great choice for photographic pictures

JPEG Example with Different Quality Settings



Digital Images on Computers

Vector (a.k.a "object-based") images

- Most picture formats that come from drawing programs
- Typically **SVG** (**.svg**) types
- **Not** pixel representation – uses mathematical formulae to represent shapes
  - Close-ups or pull-backs look smooth and clean
- Resolution is always good
  - File size is constant (usually small)
- Great for **logos**, **simple representations** of real objects
- Isn't very good for exact photographic representations

Examples of Raster vs Vector

Raster (bit-map)

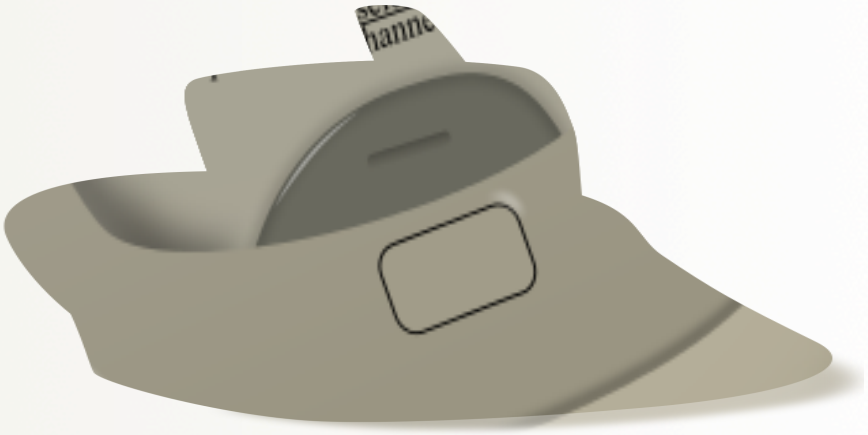
Vector

Same Examples (zoomed in)

Raster (bit-map)

Shows "pixilation"

Vector



Shows perfect reproduction

Indexed Colors in Images

- Colors on a monitor are represented by the **RGB scheme**
  - **256** variations on **each of** Red, Green, and Blue palates
  - Mixing gives a full palate of colors (per projected, not reflected light)
  - Giving you a combination of over **16 million colors**

- Are there **more** than 16 million colors in the real world?

Indexed Colors in Images

Q: Are there more than 16 million colors in the real world?

A: Yes! (well, probably, not that *I* can tell…)

A fixed scheme, like RGB, is necessary because:

1. **It puts an upper limit**
   (on colors, on file sizes, on time to render pictures onto a screen, etc…)

2. **It accommodates display technologies**
   (they're really advanced, but they're not limitless in their capabilities!)

3. It is **good enough** for 99.99% of computer (esp. Web) users!

The RGB Scale

- 256 settings for Red ➔ 8 bits
- 256 settings for Green ➔ 8 bits
- 256 settings for Blue ➔ 8 bits
  - Why?

- 1 bit = 2 combinations (0 or 1)
- 2 bits = 4 combinations (00, 01, 10, or 11)
- N bits = 2^N combinations

- RGB has 24 bits to use to define a "color"
  - 2^{24} is approximately 16 million…

The number of bits used to describe a color pallet

exponentially raises the number of colors used in a computer graphic

Wikipedia.com

Image Processing with the `cImage` Module

- Textbook's `cImage` module processes **raster** data
- Designed to work with `.gif` and `.ppm` formats only
  - Can install a library for `.jpg` format, but not available in lab
- Chapter 6 uses objects of the module's `Pixel`, `FileImage`, `EmptyImage` and `ImageWin` classes

A `Pixel` class

- A way to manage the color of one pixel

- A `color` = *amounts* of (`red`, `green`, `blue`)
  - When coded by the RGB color model
  - Range of each part: 0-255

```
whitePixel = cImage.Pixel(255,255,255)
blackPixel = cImage.Pixel(0,0,0)
purplePixel = cImage.Pixel(255,0,255)
yellowPixel = cImage.Pixel(255,255,0)
```

The "mixes" don't always work like, say, mixing paints do

- Methods: `getRed()`, `setBlue(value)`, …

Image Classes in `cImage`: `EmptyImage` and `FileImage`

- Technically both subclasses of `AbstractImage` – so objects have exactly the same features
  - **Create new**: `cImage.EmptyImage(cols, rows)`
  - Or **use existing**: `cImage.FileImage(filename)`

- Really just a way to manage a set of pixels, organized by rows and columns
  - **`x`** denotes the column – leftmost `x` is 0
  - **`y`** denotes the row – topmost `y` is 0

- Methods: `getWidth()`, `getHeight()`, `getPixel(x, y)`, `setPixel(x, y, pixel)`, `save(filename)`, … and `draw(window)`

`ImageWin` class

- A window frame that displays itself on-screen
  - And lets an image draw (itself) inside

```
window = cImage.ImageWin(title, width,height)
image.draw(window)
```

- Mostly just used to hold images, but also has some methods of its own
  - e.g., `getMouse()` – returns `(x,y)` tuple where mouse is clicked (in window, not necessarily same as image)
  - `exitOnClick()` – closes window and exits program on mouse click (like `turtle.screen` feature)

</LECTURE>