Review of concepts

• What is a variable?
  • Storage space to keep data used in our programs
  • Variables have a name
  • And also a type
  • Example:
    • Age = 19
    • CourseTitle = “CS140”
  • In this course we will use variables to store numbers, strings (text) and also media, like pictures and sounds
  • Our textbook refers to variables as names
Review of concepts

• What is a function?
  • A piece of code that performs a task
    • Think of it as a “command”
    • Example: `abs()` is a function that calculates the absolute value of a number
  • Sometimes functions need information from us
    • `abs()` needs a number
  • A parameter is a placeholder for the information a function needs
    • `abs()` needs a number: `abs(n)` -- n is a parameter
An argument is the real piece of data we give to a function to perform its task

- \text{abs}(-85) \quad -85 \text{ is an argument}
- x = 34
- \text{abs}(x) \quad x \text{ is an argument}

Sometimes a function returns a value

- The return value can be stored in another variable
- \text{abs}() \text{ returns a number}
- i = \text{abs}(-85) \quad \textbf{abs} \text{ returns 85, which is stored in } i
Review of concepts

- In Python there are pre-defined functions for us to use
  - Example: abs(), ord()
- JES has also a set of pre-defined functions to help us manipulate media:
  - makePicture(), makeSound(), pickAFile()
- Functions can be called from the Command Area
  - >>> file = pickAFile()
  - >>> x = abs(-89)
- Functions can also be called from the Program Area
Review of concepts

- We can write our own functions
- In fact, a program in Python is nothing else but a collection of functions written by the programmer
  - To create a function, use the command `def`
  - Then, the **name of the function**, and the names for the input values between parentheses (these names are the **parameters**)
    - If the function doesn’t take any parameters, then just put the empty parentheses `()`
  - End the line with a colon `:`
- The **body** of the function is indented
Review of concepts

- Statements that are indented the same, are in the same block.
- Statements that are in the same block as where the cursor is are enclosed in a blue box.
Command Area vs. Program Area

- These are two different domains
- We can create a variable in the command area and another one in the program area with the same name, however these two variables are not the same, each one belongs to a different domain.
- For example:
  - Joe Smith from Seattle, WA
  - Joe Smith from Philadelphia, PA
  - Same name, different domains, definitely two different people
- The same idea applies to functions
  - You can have variables with the same names in two different functions
  - Each variable is a different object, with a different domain
Chapter 3
We perceive light different from how it actually is

- Color is light
- Light is continuous
- Visible light is in the wavelengths between 370 and 730 nanometers
  - That’s 0.00000037 and 0.00000073 meters
- But we perceive light with color sensors in our eyes, that peak around 425 nm (blue), 550 nm (green), and 560 nm (red).
  - Our brain figures out what a particular color is based on the feedback from these three sensors
Luminance

- We perceive **borders** of things, **motion**, and **depth**, via *luminance*
  - Luminance is *not* the amount of light, but our **perception** of the amount of light: how dark/light things are.
  - We see blue as “darker” than red, even if there is same amount of light.
- Much of our luminance perception is based on comparison with the surroundings, not raw values.

Luminance is actually **color blind**.

Completely different parts of the brain process luminance and color.
Digitizing pictures as bunches of little dots

- The lack of resolution in human vision is what makes it possible to digitize pictures.
- We digitize pictures by breaking them into lots and lots of little dots
- Enough dots and it looks like a continuous whole to our eye
- Each picture element (dot) is referred to as a pixel
Pixels

- Pixels are *picture elements*
  - Each pixel object knows its *color*
  - It also knows where it is in its *picture* (its position)

When we zoom the picture to 500%, we can see individual pixels.
A Picture is a *matrix* of pixels

- A picture is encoded as a **matrix** of pixels
- It’s not a continuous line of elements, that is, an **array**
- A picture has **two dimensions**: Width and Height
- We need a two-dimensional array: a **matrix**
Referencing elements in a matrix

- We talk about positions in a matrix as \((x,y)\), or (horizontal, vertical)
- Element \((0,1)\) in the matrix at left is the value 9 and element \((2,0)\) is 13
- The position of a pixel is encoded with a pair of numbers in the form \((x, y)\)
Encoding color

• Each pixel knows its color

• Lots of encodings for color
  • Printers use CMYK: Cyan, Magenta, Yellow, and black.
  • Others use HSB (Hue, Saturation, and Brightness), also called HSV (Hue, Saturation, and Value)

• We’ll use the most common for computers
  • RGB: Red, Green, Blue
  • The color or each pixel is made of a little bit of red, a little bit of green and a little bit of blue
Encoding RGB

- Each component (red, green, and blue) is encoded as a single byte
  - 1 byte = 8 bits
  - 1 bit can only take one of these two values: 1 or 0
  - $2^8$ possible values (00000000 to 11111111), in the range 0 to 255
- Colors go from (0,0,0) to (255,255,255)
  - If all three components are the same, the color is in grayscale
    - (200,200,200) at (3,1)
  - (0,0,0) is black
  - (255,255,255) is white
How much can we encode in 8 bits?

- Let’s walk it through.
  - If we have one bit, we can represent two patterns: 0 and 1.
  - If we have two bits, we can represent four patterns: 00, 01, 10, and 11.
  - If we have three bits, we can represent eight patterns: 000, 001, 010, 011, 100, 101, 110, 111

- General rule: In $n$ bits, we can have $2^n$ patterns
  - In 8 bits, we can have $2^8$ patterns, or 256
  - If we make one pattern 0, then the highest value we can represent is $2^8-1$, or 255
What’s a “picture”?

• An encoding that represents an image
  • Knows its height and width
    • Via `getHeight()` and `getWidth()`
  • Knows its filename
  • Knows its `window` if it’s opened
    • via `show()` `explore()` and `repaint()`
• Can be saved into a file
  • `writePictureTo()`
  • We use this function to save changes made to a picture
Reminder: Manipulating Pictures

```python
>>> file = pickAFile()
>>> print file
C:\MediaSources\arch.jpg
>>> picture = makePicture(file)
>>> show(picture)
>>> print picture
Picture, filename C:\MediaSources\arch.jpg height 480 width 360
>>> explore(picture)
>>> repaint (picture)
>>> w = getWidth(picture)
>>> h = getHeight(picture)
>>> print w
360
>>> print h
480
```
Manipulating pixels

\texttt{getPixel(picture, x, y)} gets a single pixel at position (x, y)

\texttt{getPixels(picture)} gets all of the pixels in a matrix and lines them up:

\begin{table}[h]
\begin{tabular}{c|c|c}
0 & 1 & 2 & 3 \\
\hline
0 & & & \\
1 & & & \\
\end{tabular}
\end{table}
What can we do with a pixel?

- We can find out the color channels of a pixel as well as change these values
  
  - `getRed`, `getGreen`, and `getBlue` are functions that take a pixel as input and return a value between 0 and 255
  
  - `setRed`, `setGreen`, and `setBlue` are functions that take a pixel as input and a value between 0 and 255, then change the value of the color channel to the given integer value

- We can also find out the position of the pixel in the picture, that is, its x and y coordinates
  
  - `getX` and `getY` are functions that take a pixel as input and return the values of the x and y coordinates
Manipulating pixels

```python
g>>> pixel = getPixel (picture, 13, 56)
>>> print pixel
Pixel red=241 green=199 blue=161
>>> print getX(pixel)
13
>>> print getY(pixel)
56
>>> print getRed(pixel)
241
>>> print getGreen(pixel)
199
>>> print getBlue(pixel)
161
>>> setRed (pixel, 255)
>>> setGreen (pixel, 255)
>>> setBlue (pixel, 255)
```
We can also get, set, and make Colors

- **getColor** takes a pixel as input and returns a Color object with the color at that pixel
- **setColor** takes a pixel as input *and* a Color, then sets the pixel to that color
- **makeColor** takes red, green, and blue values (in that order) between 0 and 255, and returns a Color object
- **pickAColor** lets you use a color chooser and returns the chosen color
- We also have functions that can **makeLighter** and **makeDarker** an input color
Manipulating Colors

```python
>>> print getColor (pixel)
color r=255 g=255 b=255

>>> color = makeColor (0, 100, 34)

>>> print color
color r=0 g=100 b=34

>>> color2 = pickAColor()

>>> print color2
color r=96 g=145 b=128

>>> setColor (pixel, color)

>>> print (pixel)
Pixel red=0 green=100 blue=34

>>> makeLighter (color2)
Color(137, 207, 182)

>>> print color
color r=0 g=100 b=34

>>> makeDarker(color)
Color(0, 70, 23)
```
How do you find out what RGB values you have? And where?

- From the top menu: Media Tools ➔ Picture Tool
- Call the explore() function

The MediaTools menu knows what variables you have in the Command Area that contain pictures.
Comments

- Comments are used in programming to
  - Communicate with other programmers
  - Clarify pieces of code
  - Label sections of our program
- Comments are ignored by the interpreter and do not execute or perform any instructions
- A good program always includes comments
  - Header comment at the top of the program
    - Explains the purpose of the program, lists the programmer’s name, and the date the program was created/modified
  - Comment on the purpose of each function you write
  - Comment tricky/interesting lines of code
Comments

- In Python we start comments with the `#` sign
- Python ignores from `#` through the rest of the line
- If you start a line with `#` the whole line is ignored
- Comments can start any where in a line, not just the beginning of the line
- Example:
  
  ```
  #this is an example of a comment
  ```

- Why do we want lines to be ignored?
  - To be able to leave notes to ourselves or someone else about how the program works
Decreasing the red in a picture

- Problem: To decrease the red in a picture
- What we need: One picture, name it \texttt{pict}
- Step 1: Get \texttt{all} the pixels of \texttt{pict}.
- Step 2 for each pixel \texttt{p} in the set of pixels:
  - Get the value of the red of pixel \texttt{p}, and set it to 50\% of its original value
Decreasing the red in a picture

- Let’s take a closer look at Step 2
- Step 2 for each pixel $p$ in the set of pixels:
  - Get the value of the red of pixel $p$, and set it to 50% of its original value
- If the picture we are working with is 78 by 94 pixels, then there is a total of 7332 pixels (and this is a small picture!)
- We would have to write this step a total of 7332 times
- Not very efficient → there is a better way to do this
  - Use loops: programming structures that allow us to repeat a block of code as many times as needed
The for loop

- **for** is the name of the command
- An **index variable** is used to hold each element of a sequence
- The keyword **in**
- A function that generates a **sequence**
  - The index variable will be the name for one value in the sequence, each time through the loop
- A colon (":" )
- And a **block** (the indented lines of code)

```python
for index in sequence:
    Block of code
```
Example of a for loop

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

p is the index variable

allPixels is the sequence or collection of pixels to be manipulated

The loop takes one pixel at the time, gets its red value and then changes it to 50% of the original value.
What happens when a for loop is executed

- The *index variable* is set to an item in the *sequence*
- The block is executed
  - The index variable is often used inside the block
- Then execution *comes back* to the *for* statement, where the index variable gets set to the next item in the sequence
- Repeat until every value in the sequence was used.
Example 2: Creating a negative

- Let’s think it through
  - R,G,B go from 0 to 255
  - Let’s say Red is 10. That’s very light red.
    - What’s the opposite? LOTS of Red!
  - The negative of that would be 245: 255-10
- So, for each pixel, if we negate each color component in creating a new color, we negate the whole picture.
  - 255 – currentRed, 255 – currentGreen, 255 - currentBlue
Example 3: Converting to grayscale

- We know that if red = green = blue, we get grey.
- What we need is a value representing the darkness of the color, the *luminance*.
- There are lots of ways of getting it, but one way that works reasonably well is very simple—just take the average of the three color channels for every pixel:

\[
\frac{(red + green + blue)}{3}
\]
Can we get back to the original picture?

- **NO**: We’ve lost information
  - We no longer know what the ratios are between the reds, the greens, and the blues
  - We no longer know any particular value.
So that’s not really the *best* grayscale

- In reality, we don’t perceive red, green, and blue as *equal* in their amount of luminance: How bright (or non-bright) something is.

  - We tend to see blue as “darker” and red as “brighter”
  - Even if, physically, the same amount of light is coming off of each
Building a better greyscale

- We’ll weight red, green, and blue based on how light we perceive them to be, based on laboratory experiments.

```python
def greyScaleNew(picture):
    for px in getPixels(picture):
        newRed = getRed(px) * 0.299
        newGreen = getGreen(px) * 0.587
        newBlue = getBlue(px) * 0.114
        luminance = newRed + newGreen + newBlue
        setColor(px, makeColor(luminance, luminance, luminance))
```

- Note that: $0.299 + 0.587 + 0.114 = 1$
Functions: One and only one thing

• When your write functions, try to make them *general* and *reusable*
  • Programmers hate to have to re-write 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*
  • A reusable function will likely have parameters
  • A reusable function may return a value