Announcements

- Project 2 due next Monday
- Next Tuesday is review session;
- Midterm 1 on Wed., EE 129, 8:00 – 9:30pm
- Project 3 to be posted Oct. 3 (next Wed)
- Preparing for the Midterm:
  - Review Chapters 3-6 of Part 1 and chapters 8-9 of Part 2 of the textbook. Pay attention to the examples and exercises
  - Review the lecture slides, especially the clicker questions. If in doubt, run the programs

Python Boot Camp!...?

- FAQ
- Quick review of Libraries
- Sequences
- Index into a sequence
  - [ ] notation
- Slicing and other operations

Function calls as conditions?

- We have seen a lot of conditionals like:
  - if x == 3: or if y: or if (x<3) and (y%3 ==0):

- But what about something like this?

  if myFun(x):

  This is equivalent to writing:

  z = myFun(x):

  if z:

  So it is just fine

Libraries

- What is the difference between:
  1. import *library*
  2. from *library* import *

- Both provide you with a mechanism to utilize additional functionality in your program
  - Version 1 requires referencing library functions using the object notation: <library>.<function>(<parameters>)
    ```
    import math
    math.sqrt(x)
    ```
  - Version 2 obligates you to use the function name without library reference:
    ```
    from math import *
    sqrt(x)
    ```
  - If you mix the two Python throws an error

Libraries

```
>>> from math import *
>>> math.sqrt(4)
Traceback (most recent call last):
  File "<pyshell#153>", line 1, in <module>
    math.sqrt(4)
NameError: name 'math' is not defined


>>> import graphics
 >>> win = GraphWin()
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
   win = GraphWin()
NameError: name 'GraphWin' is not defined
```

Returns

- If a function does not specify a value to return it returns a special python value: "None"
- Just because a function *has* a return statement in it, does *NOT* mean it will return a value in every case

```
>>> def example(x):
        if x < 10:
            print(x)
        else:
            return(30)
>>> z = example(10)
>>> print(z)
30
>>> z = example(9)
9
>>> print(z)
None
```

Sequences in Python

- So far, we know of three types of sequences in Python
  - Strings: "Hello World"
  - Ranges: range(10)
  - Lists: [0,1,2,3,4,5] list(range(10))

Sequences

- **Range:** stores multiple *integers* consecutively in memory
- **String:** stores multiple *characters* consecutively in memory
- **List:** stores multiple *elements* consecutively in memory
- These structures provide means to access individual values.
- **Ranges, Lists** and **Strings** are indexed from 0 up

Indices (plural of index)

- Indices provide us a quick mechanism for accessing a *given* element that is contained within a sequence

[ ] Notation

- a[k] : gives a name to the k^{th} element of a list
- a =   "Sally"
  - a[k]   is equal to the k+1 character of Sally

- a = list(range(0, 10))
  - a[k] is equal to the k+1 number in the range of 0 to 9

Lists: Examples

- a = list(range(0, 10))
- print(a) [0,1,2,3,4,5,6,7,8,9]
- print(a[3]) 3
- print(a) [0,1,2,3,4,5,6,7,8,9]

Lets Make it More Concrete

```
a = 10
b = range(0,5)
c = "Sally"
```

	a	10
b[0]	b	0
b[1]		1
b[2]		2
b[3]		3
b[4]		4
c[0]	c	S
c[1]		a

….

Negative Indices

- What happens if we use a negative index?
  - Do we get an error?

```
x = range(10)
print(x[-1])      ← this will print 9
print(x[-10])     ← this will print 0
print(x[-11])     ← Error!
```

```
>>> print(x[-11])
Traceback (most recent call last):
  File "<pyshell#173>", line 1, in <module>
    print(x[-11])
IndexError: range object index out of range
```

Lets Make it More Concrete

```
a = 10
b = range(0,5)
c = "Sally"
```

	a	10
b[-5]	b	0
b[-4]		1
b[-3]		2
b[-2]		3
b[-1]		4
c[-5]	c	S
c[-4]		a

….

Lists: Examples

- a = list(range(0, 10))
- print(a) [0,1,2,3,4,5,6,7,8,9]
- print(a[-3]) 7
- print(a) [0,1,2,3,4,5,6,7,8,9]

Lists

- The **[ ]** can be used to *index* into an list, range, or string. For example:

```
i = 0
x = list(range(0,10))
while i < 10 :
    print (x[i])
    i = i + 1
```

```
i = 0
x = range(0,10)
while i < 10 :
    print (x[i])
    i = i + 1
```

Strings

- The **[ ]** can be used in the same way on a string. For example:

```
i = 0
x = “This is a string”
while i < 16 :
    print (x[i])
    i = i + 1
```

The len() function

- The len function gives you the "length" or number of elements in a sequence
- Strings: number of characters in the string
- Ranges: number of integers in the range
- Lists: number of elements in the list

```
>>> len(range(10))
10
>>> len([0,1,2,3,4,5])
6
>>> len("this is a string")
16
```

Defensive Coding

- These three examples suffer from the same defect!
  - The while loop is *hard coded!*

```
i = 0
x = list(range(0,10))
while i < 10 :
    print (x[i])
    i = i + 1
```

```
i = 0
x = "This is a string"
while i < 17 :
    print (x[i])
    i = i + 1
```

ERROR!

The len function

- A better way to write the previous code:

```
i = 0
x = “This is a string”
while i < len(x):
    print (x[i])
    i = i + 1
```

Clicker Question: Are these two functions equivalent?

```
def printByCharacter(str)
  i = 0
  while i < len(str):
      print (str[i])
      i = i + 1
```

```
def printByCharacter(str)
  i = 0
  while i < 16:
      print (str[i])
      i = i + 1
```

A: yes

B: no

Why is this important?

- We want to write general purpose functions

```
def printByCharacter(str)
  i = 0
  while i < len(str):
      print (str[i])
      i = i + 1
```

Typical indexing mistakes

- Undershooting the bounds
  - a = “hello” a[-6]
- Overshooting the bounds
  - a = “hello” a[5]
- Off by one
  - a[0] vs a[1]
  - By convention we use 0-based indexing
    - a= “hello”
    - print(a[0])
    - print(a[1])

Homework

- Study for the exam!
- Work on Project 2

Python Boot Camp

- String Slicing
- Lists
  - Heterogeneous vs homogenous
  - Assignment to lists allowed
  - Lists containing other sequences

CQ: Are these programs equivalent?

```
i = 0
x = "This is a string"
while i < len(x):
    print (x[i])
    i = i + 1
```

```
x = "This is a string"
for y in x:
    print (y)
```

A: yes

B: no

What is going on here?

```
x = "This is a string"
for y in x:
    print (y)
```

x	T
	h
	i
	s
	i

….

Under the hood we are doing something similar to:

y = x[j]

CQ: Are these programs equivalent?

```
i = 0
x = "This is a string"
while i < len(x):
    print (x[i])
    i = i + 1
```

```
x = "This is a string"
i = 0 – len(x)
while i < 0:
    print (x[i])
    i = i + 1
```

A: yes

B: no

Slicing

- In addition to selecting a single value from an array or string the [ ] can be used to select values in a special range.

```
x = "This is a string"
print (x[0])
print (x[0:5])
print (x[:3])
print (x[3:])
print (x[-1:])
print (x[:-1])
```

Slicing

x = *"This is a string"*

print (x[0])

T

print (x[0:5])

This

print (x[:3])

Thi

print (x[3:])

s is a string

print (x[-1:])

g

print (x[:-1])

This is a strin

Lists

- We can also store more complex elements into an list. For example, consider these two cases:

```
x = “ABCD”
y = [“A”,”B”,”C”,”D”]
print (x)
ABCD
print (y)
['A', 'B', 'C', 'D']
```

Lists

- y is an example of a list of strings. Each element is a string. We could expand it as follows:

```
y = [“ABCD”, ”BCD”, ”CD”, ”D”]
```

- As you can see each element can be a different length. They can also be different types:

```
y = [“ABCD”, [1,2,3] , ”CD”, ”D”]
```

Lists

- Suppose we wanted to extract the value 3

```
y = [“ABCD”, [1,2,3] , ”CD”, ”D”]
y[1][2]
```

- The first set of **[ ]** get the array in position 1 of y. The second **[ ]** is selecting the element in position 2 of that array. This is equiv. to:

```
z = y[1]
z[2]
```

Assigning to Lists

- The [ ] syntax not only allows us to access a given element, it lets us access that memory location
  - Namely, we can assign to that location
    - b[2] = 100
    - print(b[2])
    - b[2] = b[2] – 50
    - print(b[2])

	a	10
b[0]	b	0
b[1]		1
b[2]		2
b[3]		3
b[4]		4

Strings are Immutable

- What do we mean by immutable?
  - We cannot assign to strings like we do to lists

```
i = 0
x = "This is a string"
x[i] = 'b'
```

Ranges are Immutable

- What do we mean by immutable?
  - We cannot assign to strings like we do to lists

```
i = 0
x = range(10)
x[i] = ‘b’
```

Operations on Lists

- Just like we can *concatenate* strings we can concatenate lists
  - print ([1, 2, 3] + [4, 5, 6])
  - Will print: [1, 2, 3, 4, 5, 6]

- Just like we can *slice* strings we can also slice lists
  - b = [1, 2, 3, 4, 5, 6]
  - print (b[2:5])
  - Will print [3, 4, 5]

Advanced List Operations

- We once again use the object.method() syntax
  - This time the list is the object
  - Notice the list type supports *different* methods from the string type
- c = [1, 2, 3, 4, 5]
- c.append(6)
  - Results in c having an additional element:
    - [1, 2, 3, 4, 5, 6]

Announcements

- Amazon will update Part 1 of our e-Textbook. Instructions are on the homepage.
- Next Tuesday is review session;
- Midterm 1 on Wed., EE 129, 8:00 – 9:30pm
- Project 3 to be posted Oct. 3 (next Wed)
- Preparing for the Midterm:
  - Review Chapters 3-6 of Part 1 and chapters 8-9 of Part 2 of the textbook. Pay attention to the examples and exercises
  - Review the lecture slides, especially the clicker questions. If in doubt, run the programs

CQ:Are these programs equivalent?

1

```
b =
[ ‘h’ ,’ e’ ,’ l’ ,’ l
’ ,’ o’ ]
def myFun(l):
    l.append(6)
    return l
print(myFun(b))
```

2

```
b =
[ ‘h’ ,’ e’ ,’ l’ ,’ l’ ,’
o’ ]
def myFun(l):
    l +[6]
    return l
print(myFun(b))
```

A: yes

B: no

What can we do to make them equivalent?

```
b = [ 'h' ,' e' ,' l' ,' l' ,' o' ]
def myFun(l):
    l = l +[6]
    return l
print(myFun(b))
```

- Now program 2 will print the same as program 1
  - But what about the value of b after the function?

Advanced List Operations

- L = [0, 1, 2]
- L.extend([4, 5, 6])
  - print(L) will print: [0, 1, 2, 4, 5, 6]
- L.extend(["Hello"])
  - print(L) will print: [0, 1, 2, 4, 5, 6, "hello"]
- L.insert(0, "a")
  - print(L) will print: ["a" , 0, 1, 2, 4, 5, 6, "hello"]
- L.insert(2, "a")
  - print(L) will print: ["a" , 0, "a" , 1, 2, 4, 5, 6, "hello"]

A special case for insert

- L = [0, 1, 2]
- L.insert(len(L), 3)
  - print (L)   will print [0, 1, 2, 3]
- L.insert(3000, 4)
  - print (L)   will print [0, 1, 2, 3, 4]
- Insert also works with negative indices
  - Try it and see what you get!

CQ:Are these programs equivalent?

1

```
b =
[ ‘h’ ,’ e’ ,’ l’ ,’ l
’ ,’ o’ ]
b.insert(len(b), “w” )
print(b)
```

2

```
b =
[ ‘h’ ,’ e’ ,’ l’ ,’ l’ ,’
o’ ]
b.append( “w” )
print(b)
```

A: yes

B: no

Advanced List Operations

- L = [0, 1, 2, 0]
- L.reverse()
  - print(L)   will print: [0, 2, 1, 0]
- L.remove(0)
  - print(L)   will print: [2, 1, 0]
- L.remove(0)
  - print(L)   will print: [2, 1]
- print (L.index(2))   will print 0

Why are Lists useful?

- They provide a mechanism for creating a collection of items

```
def doubleList(b):
    i = 0
    while i < len(b):
        b[i] = 2 * b[i]
        i = i +1
    return (b)

print(doubleList([1,2,3]))
```

Why lists are useful

- We can encode other structures, for instance arrays
- That is

[[1,2,3], [4,5,6]]

- would encode

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

- Can it also encode

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

???

Applications & Projects 3, 4

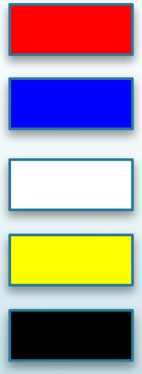
- Image as rectangular raster
- Pixels and color channels
- Who needs 16M colors anyway?
- Changing pixels infrastructure
  - Coordinate system
  - Black and white case
- Turtle tracks
- Instruction loop of a conceptual machine
- Abstractions

Image Basics

- An image is a rectangle of color dots called pixels
- In B&W images, the pixels are either black, or white, or a shade of grey.
- We can make an image by
  - Building a rectangle of pixels, called a raster
  - Assigning to each pixel a color value
- .gif, .jpg, etc. are just encodings

Color Values

- All colors in computer images are a combination of red, green and blue, the color channels
  - Each color channel is encoded as a number 0..255
    - 0 means the color is absent,
    - 255 the color is at maximum brightness
    - Gray means R=G=B
    - All other values are shades of brightness of the color
  - Example:
    - R,G,B = 255,0,0
    - R,G,B = 0,0,255
    - R,G,B = 255,255,255
    - R,G,B = 255,255,0
    - R,G,B = 0,0,0



Pixels and Coordinates

- To make or change a picture, we need to assign values to the pixels
- We can enumerate the pixels using Cartesian coordinates (column, row):
  - V = getPixel(c,r) would deliver the three components
  - setPixel(c,r,V) would set the three components to V
  - But what is V?

0,2	1,2	2,2
0,1	1,1	2,1
0,0	1,0	2,0

Pixel Values

- For project 4, we restrict to black and white, as it simplifies our data structures:
  - setPixelToBlack(c,r)
  - setPixelToWhite(c,r)
  - setAllPixelsToWhite()
  - isPixelWhite(c,r)
  - isPixelBlack(c,r)

Pixel Values

- For project 3 we deal with RGB triples, allowing each color channel to be in the full range of 0..255
- Here, we will manipulate pixel values of real images
- Example: make a color picture B&W:
  - Each pixel value is averaged and the average assigned to each RGB field
  - Color image pixel (128,205,33) would be replaced with (122,122,122), because (128+205+33)/3 == 122

Does it matter?

- If some area is colored (r,g,b) and an adjacent area is colored (r+3,g-4,b+7) can you tell the difference?

(156,132,200) V. (159,128,207)

Color value in binary

- 0..255 equals 8 bits:
  - (156, 132, 200) is (**10011100,10000100,11001000**)
  - (159, 128, 207) is (**10011111,10000000,11001111**)
- Conjecture: **the 4 low-order bits do not matter**
  - So only 4 bits matter? Try it!
  - We can use the low order bits for clandestine purpose:
    - Encode the high order 4 bits of another picture as the low-order 4 bits of this picture
    - Use the 4 low-order bits for other things, e.g. watermarking

Things to try

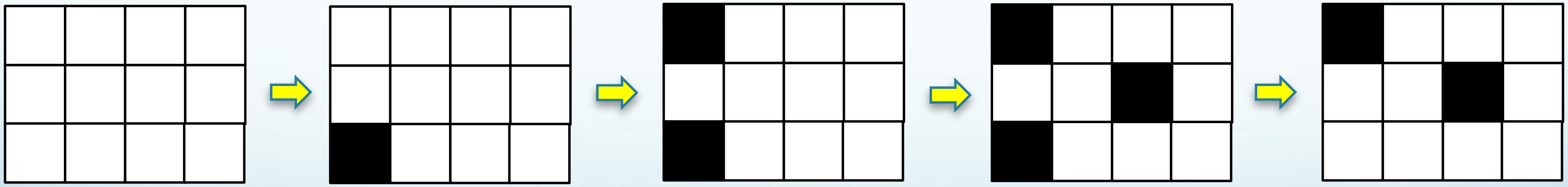
- Set the 4 low-order bits to 0
- Hide one picture in another
- Extract the hidden picture
- Put text into the picture
- …

Making a b&w picture w/o gray

- Need to create the pixel rectangle:
  - Canvas functions:
    - makeWhiteImage(width,height)
    - destroyImage()
  - Pixel assignment and test functions
    - setPixelToBlack(c,r)
    - setPixelToWhite(c,r)
    - setAllPixelsToWhite()
    - isPixelWhite(c,r)
    - isPixelBlack(c,r)

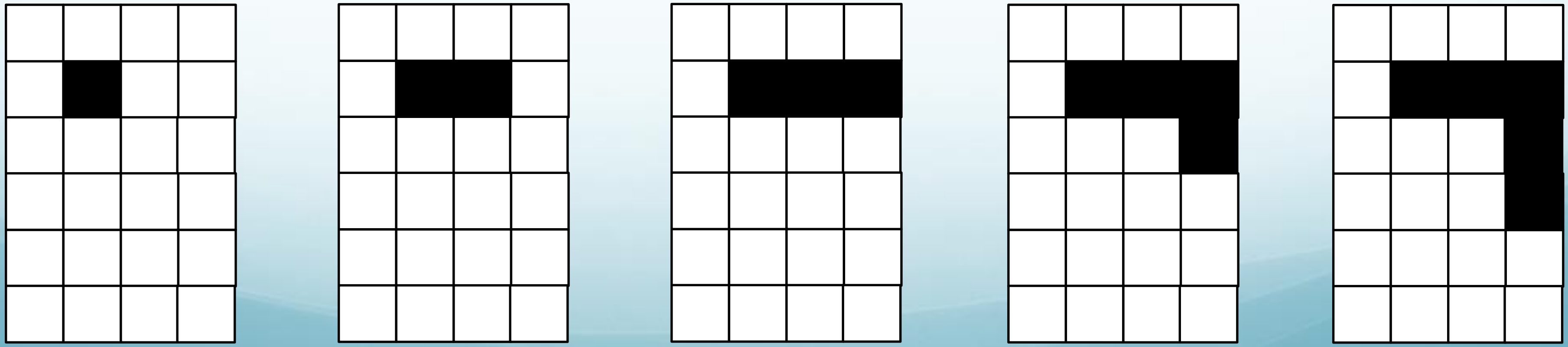
Example

- makeWhiteImage(4,3)
  setPixelToBlack(0,0)
  setPixelToBlack(0,2)
  setPixelToBlack(2,1)
  setPixelToWhite(0,0)



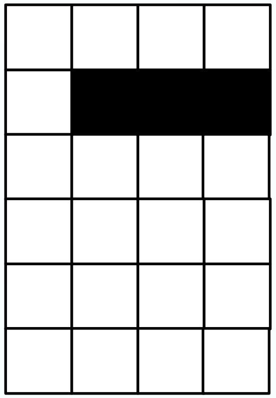
Turtle Metaphor

- Turtle moves across canvas, pixel by pixel, starts somewhere
- Where the turtle is, it leaves a black pixel
- Turtle can move N, S, E, W
- Tell the turtle:
  - Where to start, here at position (1,4)
  - Where to move
- Moves could be encoded as a string: "EESS"



Boundaries

- What should happen if the turtle wants to move East but is at the border?
  - Could throw an error
  - Could stay put at border
  - Could move virtually off canvas
  - Example: "EEE"



- If we stay put, then turtle ignores going off raster
  - Then "EESS" is equivalent to "EEESS"

Turtle Algorithm

- Recall the "read book" algorithm…
- Turtle algorithm:
  1. Get width, height of image; create white canvas
  2. Get pos_x, pos_y of turtle
  3. Make pixel (pos_x, pos_y) black
  4. Get string S encoding turtle moves
  5. While there remain characters of S not yet processed:
  6. move turtle according to next character
  7. make new pixel black

Parallels

- Characters of S are like machine instruction
- The "machine" is the infrastructure of marking turtle squares black and keeping the turtle on the rectangle of pixels, the canvas
- The instructions manipulating pixels are the machine instructions…
  - But that machine has a low level of abstraction
- CS is all about abstractions and conceptual machines