#7: Producing Output

SAMS PROGRAMMING C

Housekeeping

Quiz1 Grading – partial credit, median 100, average 76.7

Overall Grade Calculation – hw1 + hw2 + hw3 + hw4 + hw5 + 2*quiz1 + 2*quiz2

- participation used in edge cases

Reminder: **hard coding test cases is not allowed, and will not receive any points!**

Also: **you must write all of your own code on the homeworks!** Collaboration is okay, but writing code for someone else or accepting someone else's code is not.

Finally: **it's okay if you sometimes can't finish all the problems on a homework or quiz!** Just do your best and submit whatever you get done. We're here to learn, not to score perfect grades.

Review from Last Week

Use **indexing and slicing** on strings while writing functions

Utilize **lists** as data structures when writing programs

Understand the difference between **mutable** and **immutable** datatypes

Today's Learning Goals

Understand how **aliasing** works with lists and other mutable objects

Build **lists of multiple dimensions**

Use the **tkinter library** to build graphics in Python

More About Lists

Reminder: Lists are Mutable

Last week, we learned that lists are **mutable**- the values in them can be changed directly.

This is possible because we aren't actually storing the list value directly inside its variable. Instead, the variable contains a **reference** to the list. We can change the list while leaving the reference the same.

This is why we can call `lst.append(item)` without needing to set the call equal to a value!

Side Effects...

The mutable nature of lists has an important side effect- **copying variables works differently than with non-mutable values.**

Normally, if I make a copy of a string or number, the copy is disconnected from the original value:

```
a = "foo"

b = a

a = a + "foo"

print(a, b)
```

Side Effects...

The mutable nature of lists has an important side effect- **copying variables works differently than with non-mutable values.**

But if I copy a list, we'll get an unexpected result...

```
a = [1, 2, 3]
b = a
a.append(4)
print(a, b)
```

Aliasing

List copying is broken because our variables are **aliased**. They both store the same reference, where each reference points to the same list. Changing the list doesn't change the reference.

We can make what's going on clearer by **visualizing the code's execution**.

Example here: https://goo.gl/aZwfcW

Exercise: Aliasing

Predict what the following code will print. When we run the code, check your results.

```
x = [ "a", "b", "c" ]
y = x
s = "foo"
x.append(s)
s = s + "bar"
print("x", x)
print("y", y)
print("s", s)
```

Function Call Aliasing

When we call a list in a function, the parameter is an alias of the originally provided list. This lets us write functions that are **destructive**- they change the provided value instead of returning.

```
def doubleValues(lst):
    for i in range(len(lst)):
        lst[i] = lst[i] * 2
a = [1, 2, 3]
print("before", a)
print("result", doubleValues(a))
print("after", a)
```

Destructive vs. Non-Destructive

If we want to make a list function that is **non-destructive**, we make and return a new list instead.

```
def doubleValues(lst):
    result = [ ]
    for i in range(len(lst)):
        result.append(lst[i] * 2)
    return result
a = [1, 2, 3]
print("before", a)
print("result", doubleValues(a))
print("after", a)
```

Built-in Functions

Built-in list functions can be destructive or nondestructive.

Need to add a single element?

`lst.append(item)` is destructive; `lst = lst + [item]` is non-destructive.

Need to remove the last element?

`lst.pop()` is destructive; `lst = lst[:-1]` is non-destructive.

If you aren't sure whether a function is destructive or nondestructive, pay attention to whether it changes the list and whether it requires an assignment.

Exercise: Lists in Functions

We want to write a function `replace(lst, oldItem, newItem)` that replaces all instances of `oldItem` in the list with `newItem`.

How would we implement this function destructively?

How would we implement it non-destructively?

Multi-dimensional Lists

Reminder: lists can hold any type of data. That includes more lists!

We often talk about creating **two-dimensional lists**. These are just lists that contain one-dimensional (regular) lists. They're useful for storing information that comes in grids (think pixels, game boards, spreadsheets...)

```
grid = [ ["city", "state"],
         ["Pittsburgh", "PA"],
         ["Baltimore", "MD"],
         ["New Orleans", "LA"] ]
```

2D List Indexing

When indexing into a multi-dimensional list, you index from the **outside in**.

With 2D lists, we refer to **rows** (the inner lists) and **columns** (the indices within the list). This lets us treat 2D lists like data tables.

```
lst = [ [ "row 0 col 0", "row 0 col 1" ],
        [ "row 1 col 0", "row 1 col 1" ] ]
print(lst[1]) # [ "row 1 col 0", "row 1 col 1" ]
print(lst[1][0]) # "row 1 col 0"
```

2D List Iteration

Likewise, when iterating over a multi-dimensional list, we use **multiple nested loops**. For 2D lists, we first iterate over the rows (the inner lists), then the columns (the elements).

```
lst = [ [ "a", "b" ], [ "c", "d" ] ]
for row in range(len(lst)):
    for col in range(len(lst[row])):
        print(lst[row][col])
```

Exercise: 2D Lists

We want to write a function that takes a 2D list of one-character strings and a word, and returns the [row, col] index of the starting character of the word if that word can be found in an adjacent string of characters in the list, or None otherwise.

Basically, we want to write a word search solver!

Testing wordSearch

```
def testWordSearch():
    board = [ [ 'd', 'o', 'g' ],
              [ 't', 'a', 'c' ],
              [ 'o', 'a', 't' ],
              [ 'u', 'r', 'k' ] ]
    print(wordSearch(board, "dog")) # [0, 0]
    print(wordSearch(board, "cat")) # [1, 2]
    print(wordSearch(board, "tad")) # [2, 2]
    print(wordSearch(board, "cow")) # None
```

Graphics!

Tkinter Canvas

In Python, we can draw graphics on the screen using many different modules. We'll use Tkinter in class because it's built-in, but there are other options for outside of class (turtle, pygame, Panda3D...)

Tkinter creates a new window on the screen and puts a **Canvas** into that window. We'll call methods on that canvas in order to draw on it.

NOTE: Tkinter will not work on most online editors. You'll need to run it locally on the computer.

NOTE 2: Tkinter documentation can be found at http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/canvas.html

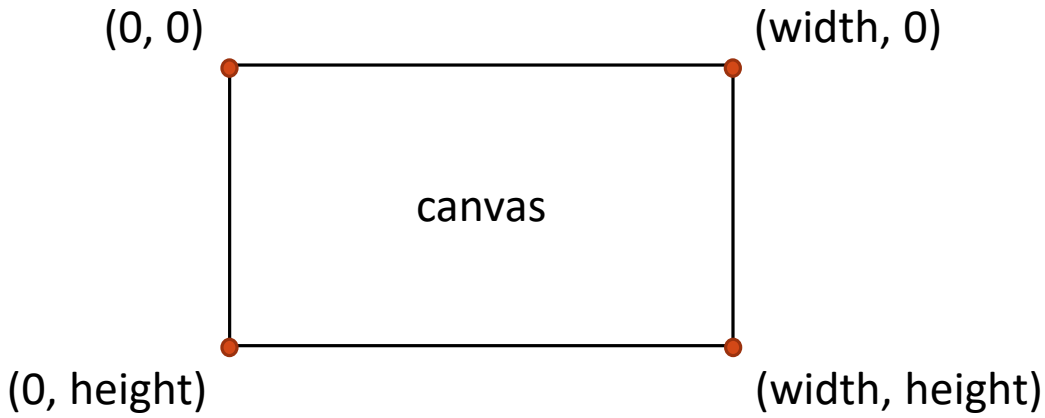
Tkinter starter code

```
from tkinter import *


def draw(canvas, width, height):
    pass # replace with your drawing code!


def runDrawing(width=300, height=300):
    root = Tk()
    canvas = Canvas(root, width=width, height=height)
    canvas.configure(bd=0, highlightthickness=0)
    canvas.pack()
    draw(canvas, width, height)
    root.mainloop()
    print("bye!")


runDrawing(400, 200)
```

Coordinates on the Canvas

You can think of the canvas (or any image) as a 2D grid of pixels, where each pixel can be filled with a dot of color. This grid has a pre-set **width** and **height;** the number of pixels from left to right and the number of pixels from top to bottom.

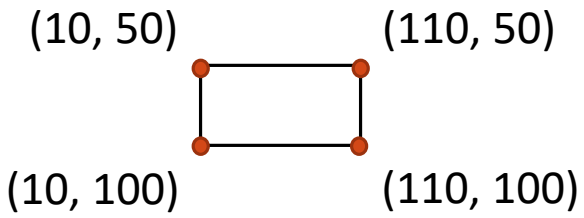
We can refer to pixels on the canvas by their (x, y) coordinates. However, these coordinates are different from coordinates on normal graphs- they start at the **top left corner** of the canvas.



Drawing a rectangle

To draw a rectangle, we use the method **create_rectangle**. This method takes four required parameters: the x and y coordinates of the top-left corner, and the x and y coordinates of the bottom-right corner. The rectangle will then be drawn between those two points.

```
canvas.create_rectangle(10, 50, 110, 100)
```



Changing the rectangle

We can also add many **optional parameters** to the rectangle method to change the rectangle's appearance. You can include as many as you want- just put them after the coordinates.

```
canvas.create_rectangle(10, 50, 110, 100, fill="yellow") # makes rectangle yellow
canvas.create_rectangle(10, 50, 110, 100, outline="red") # makes border red
canvas.create_rectangle(10, 50, 110, 100, width=5) # makes border 5 pixels wide
canvas.create_rectangle(10, 50, 110, 100, width=0) # removes border
```

Drawing multiple shapes

If we draw more than one shape, the shapes can overlap! Shapes which are drawn later are drawn on top.

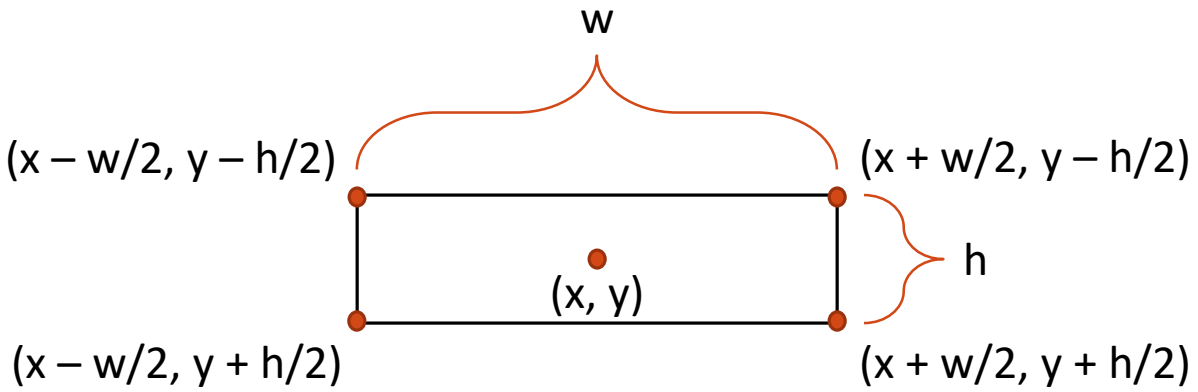
```
def draw(canvas, width, height):
    canvas.create_rectangle(  0,   0, 150, 150, fill="yellow")
    canvas.create_rectangle(100,  50, 250, 100, fill="orange", width=5)
    canvas.create_rectangle( 50, 100, 150, 200, fill="green",
                                                outline="red", width=3)
    canvas.create_rectangle(125,  25, 175, 190, fill="purple", width=0)
```

Calculating the center

Often we want to draw shapes based on a center point, a shape width, and a shape height.

To do this, we need to calculate the left, top, right, and bottom coordinates using this information.

```
centerX, centerY = 200, 200

rectWidth, rectHeight = 300, 80

left = centerX - rectWidth/2

top = centerY - rectHeight/2

right = centerX + rectWidth/2

bottom = centerY + rectHeight/2

canvas.create_rectangle(left, top, right, bottom)
```



Adjusting Size Based on Window Size

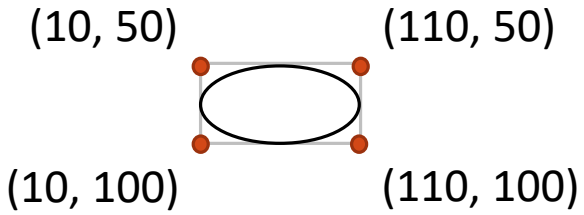
We know the window's height and width in draw() (as they are provided as parameters), so we can draw objects *proportionally* to the window, so they resize appropriately.

```
def draw(canvas, width, height, img):
    squareSize = min(width, height)
    canvas.create_rectangle(width/2 - squareSize/2,
                            height/2 - squareSize/2,
                            width/2 + squareSize/2,
                            height/2 + squareSize/2,
                            fill="red")
```

Drawing an oval

Of course, we can draw more shapes than just rectangles. First, to draw an oval, use **create_oval**. This function uses the same parameters as create_rectangle, where the coordinates mark the oval's **bounding box**. create_oval has the same optional parameters as create_rectangle.

```
canvas.create_oval(10, 50, 110, 100)
```



Drawing a polygon

To draw a polygon with **create_polygon**, we specify the coordinates of **each of the polygon's points, in perimeter order**. The polygon can have as many points as needed, but will need at least three points to appear.

```
canvas.create_polygon(10, 10, 50, 150, 100, 50)
```

You can use the normal optional parameters with polygons, and you can also create a cool curved shape by setting `smooth=1`.

```
canvas.create_polygon(10, 10, 50, 150, 100, 50, smooth=1)
```

Drawing a line

Drawing a line with **create_line** is like drawing a polygon- include the coordinate for each point on the line where the direction changes. However, lines are allowed to only have two points.

```
canvas.create_line(10, 50, 200, 150, fill="blue")
canvas.create_line(10, 10, 50, 150, 100, 50, fill="red")
```

Lines can also be smoothed (though you should use **create_arc** to draw arcs), but do not have an outline color. Lines can also be given an arrow on the first point (arrow=FIRST), last point (arrow=LAST), or both (arrow=BOTH).

```
canvas.create_line(10, 50, 200, 150, arrow=BOTH)
```

Drawing text

To write text in the canvas, we use **create_text**. This takes the x, y coordinate of the center point of the text and can have the following optional parameters:

```
canvas.create_text(100, 100, text="Hello World!") # the text to be displayed. Not really optional...


canvas.create_text(100, 100, text="Hello World!", fill="red") # text color
canvas.create_text(100, 100, text="Hello World!", font="Arial 30 bold") # text font, size, and type
canvas.create_text(100, 100, text="Hello World!", anchor=NW) # anchors are used to specify where the
# coordinate is w.r.t the text. The default is CENTER; you can also use NW, N, NE, E, SE, S, SW, W
canvas.create_text(100, 100, text="Hello World!", width=50) # by default, text is all on one line.
# If width is set, the text length will be restricted & will automatically break into multiple lines.
```

Drawing images

If we want to use a pre-made image in Tkinter, we can load one in as a PhotoImage. This can be created with:

```
img = PhotoImage(file="sample.gif")
```

We can resize the image if needed, using **subsample** to make it smaller and **zoom** to make it bigger.

```
img = img.subsample(5) # make the image 5 times smaller
```

```
img = img.zoom(2) # make the image twice as large
```

Unfortunately, PhotoImages can only be .pgm, .ppm, and .gif files. For more filetypes, use the external module PIL.

Drawing images

Once you've created an image, you can draw it with **create_image**. This method takes the x, y coordinates of the image and can have other optional parameters...

```
# the image to be displayed. not really optional...
canvas.create_image(200, 100, image=imageVar)
# the anchor point of the coordinate. Same as for text, default CENTER
canvas.create_image(200, 100, anchor=N)
```

NOTE: images take a while to load, so they must be created **in the runDrawing function**, not in draw(). They should be initialized after root=Tk() but before draw().

Problem-solving with Graphics

Now we have all the building blocks we need to make cool images!

Let's start by drawing flags: http://flagpedia.net/

Today's Learning Goals

Understand how **aliasing** works with lists and other mutable objects

Build **lists of multiple dimensions**

Use the **tkinter library** to build graphics in Python