#2: Graphics and Algorithmic Thinking

SAMS SENIOR CS TRACK

Last time

Learned what programming is

Used data values, variables, and functions to build programs

Today's Learning Goals

Use the tkinter library to construct graphics with programming

Solve unfamiliar problems using algorithmic thinking

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

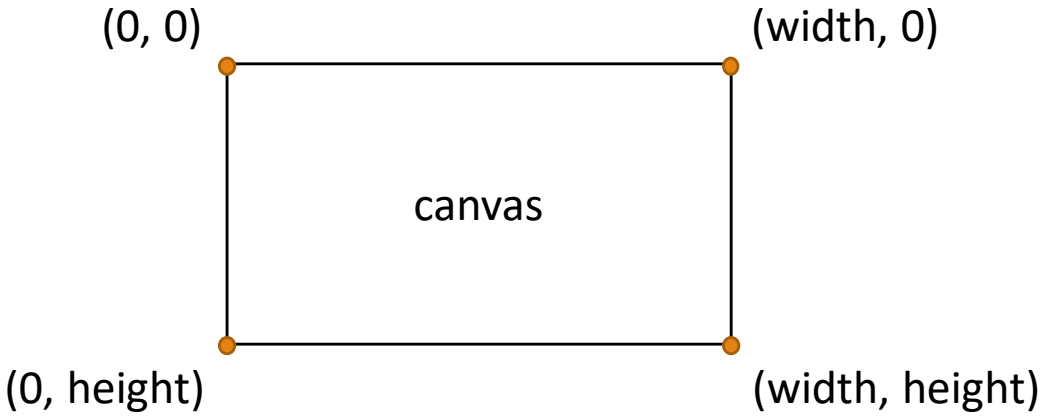
Tkinter starter code

```
from tkinter import *


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


def runDrawing(width=300, height=300):
    root = Tk()
    root.resizable(width=False, height=False) # prevents resizing window
    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 two-dimensional 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.

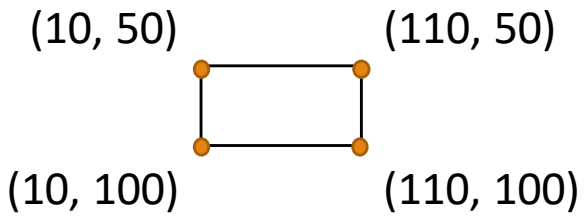
We can refer to pixels on the canvas by their (x, y) coordinates. However, these coordinates are different from coordinates on normal graphs- they start at the **top left corner** of the canvas.



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.

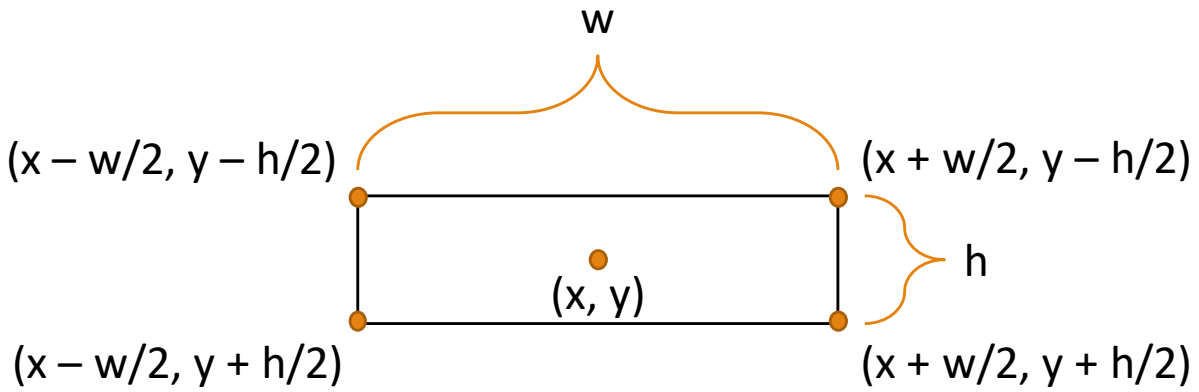
```
def draw(canvas, width, height):
    canvas.create_rectangle(  0,   0, 150, 150, fill="yellow")
    canvas.create_rectangle(100,  50, 250, 100, fill="orange", width=5)
    canvas.create_rectangle( 50, 100, 150, 200, fill="green",
                                                outline="red", width=3)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

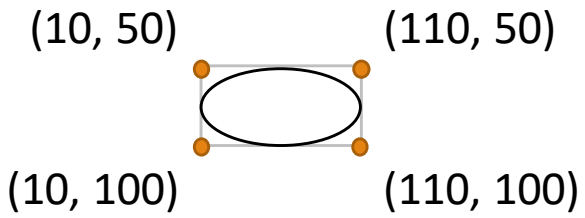
We know the window's height and width in draw() (as they are provided as parameters), so we can draw objects *proportionally* to the window, so they resize appropriately.

```
def draw(canvas, width, height):
    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, image=imageVar, 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().

Coding with Graphics

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

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

Building Blocks

We now have most of the building blocks of basic programs: numbers, text, Booleans, variables, functions, and graphics.

Next, let's talk about how we figure out which blocks to use and when...

Algorithmic Thinking

Problem Solving

Programming in general involves determining **how to solve problems with algorithms**. An algorithm is a defined process that accomplishes some task. When we write Python code, we are encoding the process into a language the computer can understand.

Doing problem solving in programming can be broken down into the following steps:

1. Understand the problem
2. Devise a plan
3. Carry out the plan
4. Review your work

Understanding the Problem

The first step of problem solving is to **thoroughly understand the problem**. This sounds simple, but can be tricky. If you thoroughly understand a problem, you should understand what the function will do on *any given input*. This will involve carefully reading the WHOLE prompt.

Example: "Write the function eggCartons(eggs) that takes an integer number of eggs and returns the number of egg cartons needed to hold that many eggs."

What should eggCartons(eggs) return when given the value...

- 12
- 13
- 1
- 11
- 32
- 0

Devising a Plan

Once you understand a problem, you need to devise an algorithm or plan for how to solve it. Depending on the prompt, you may need to **translate a provided algorithm**, **apply a known algorithm pattern**, or **generate a new algorithm**.

Translate: the prompt will describe an algorithm in regular language, and you will only need to translate it into Python. For example, "write a function that computes the distance between two points". You don't need to invent a distance algorithm- one already exists!

Apply: the prompt may be similar to a problem you've seen before; for example, "draw a flag with four vertical stripes". Then you can apply the previous code pattern and modify it as necessary.

Generating New Algorithms

The hardest problems are those in which the algorithm to solve the problem is not immediately familiar. In these cases, you will need to invent an algorithm yourself.

The best way to learn how to generate algorithms is to **practice**, as each problem has its own individual quirks. However, there are three helpful approaches you can use when generating algorithms: **induction, top-down design,** and **human-computer**.

Induction

In **induction**, you investigate several pairs of inputs and outputs to attempt to find a pattern. That pattern can then be generalized into an algorithm.

Example: Write the function nearestBusStop(street) that takes a non-negative int street number, and returns the nearest bus stop to the given street, where buses stop every 8th street, including street 0, and ties go to the lower street.

For example, the nearest bus stop to 12th street is 8th street, and the nearest bus stop to 13th street is 16th street.

Look at where the output changes based on the input, graph the results, and write the program.

4 – 0

5 – 8

8 – 8

12 – 8

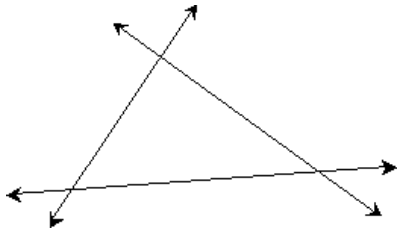
13 – 16

Top-down design

In **top-down design**, you attempt to simplify the problem by breaking it down into multiple easier problems. You can then write code using **helper functions** that solves the main problem, then write each helper function individually to complete the work.

Example: Write the function threeLinesArea(m1, b1, m2, b2, m3, b3), that takes the slopes and intercepts of three lines and returns the area of the triangle that the three lines form when they intersect.

This initially seems difficult, because there isn't an immediate solution. However, we can make it simpler by breaking it down into parts and solving each part separately.



First: if we know the lengths of the three sides of the formed triangle, we can use Heron's Formula to calculate the area. So we need to find the lengths of the three sides.

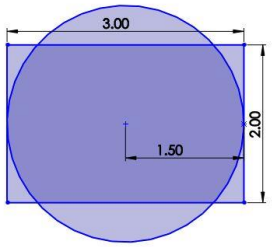
Second: if we know the coordinates of the two ends of each side, we can use the distance formula to compute the length. So we need to find the coordinates of the three points of the triangle.

Third: We can find the three points where the lines intersect by setting their x's and y's equal and solving the equation. Then we're done!

Human Computer

In **human-computer**, you solve a problem by walking through an example yourself, and taking note of what approach you personally use to find the answer. We often already have systematic approaches that we don't actively think of while doing actions!

Example: Write the function rectangularPegRoundHole(r, w, h), which returns True if a rectangular peg with width w and height h can pass through a round hole with radius r, and False otherwise.



Imagine trying to fit a rectangular object into a round hole. How can you tell if the peg will never be able to fit?

You check the longest part of the rectangle- the diagonal! Now you just need to calculate the length of that diagonal in code.

Carrying out the Plan

Once we've come up with a working algorithmic plan, we need to **carry out the plan** by translating it into code. As you become fluent in coding and Python, you'll be able to go straight from idea to code, but when you're starting out, this might be difficult!

If you're having trouble, it can help to start by **writing the algorithm as a series of steps in plain language**, either on paper or in comments. Your written algorithm should be clear and complete enough that another person could carry it out, and should distinctly label any information that needs to be remembered (that information will become variables later on).

Once your written algorithm is complete, you can translate **one step at a time** into code. If you don't know how to do a certain step in Python, you can check the course notes or ask a TA to find the appropriate programming tool, then experiment with that tool in the interpreter until you understand it.

Reviewing your work

Finally, once your program is written, you need to **review your work** by running and testing your code. This isn't like spot-checking your work on a written assignment, because the computer can already tell you when something is wrong!

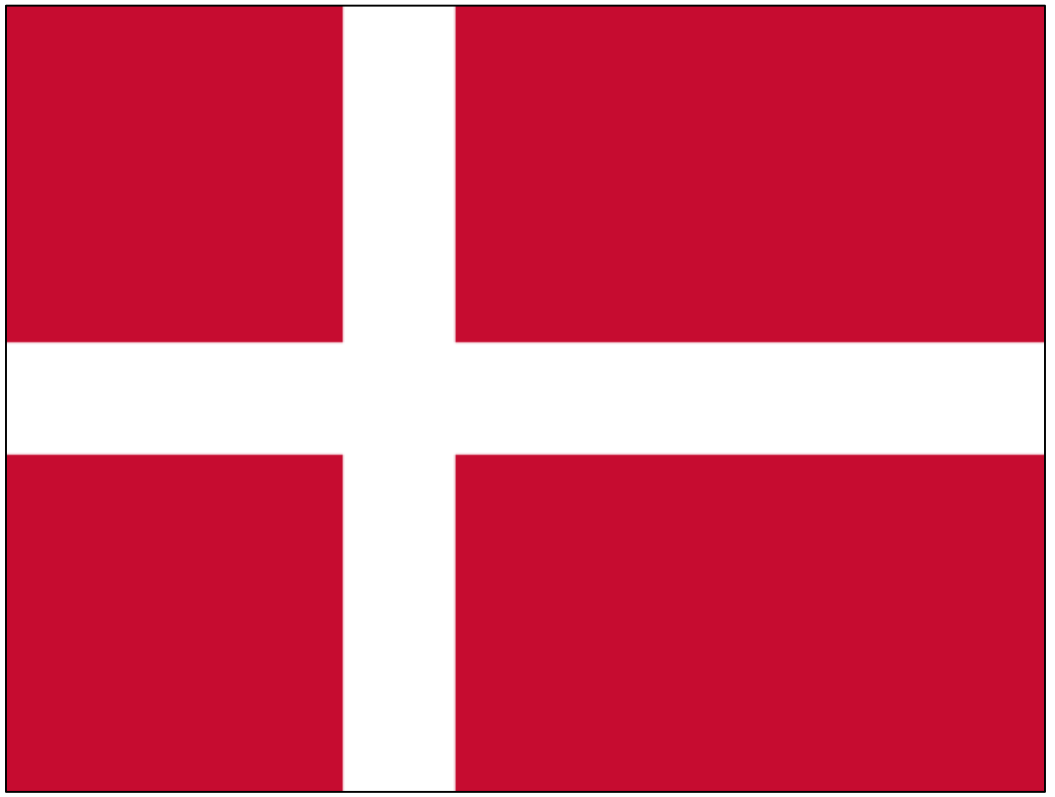
Remember the input-output pairs we considered in Step 1? We can now turn those into **test cases**, then run the program on the test cases to make sure it's working appropriately. When a test case fails, we can use **debugging** to determine what's going wrong. Once all the test cases are passing, you should read your program one more time to make sure it makes sense, then submit it for final review.

We'll go over testing and debugging in more depth next week. Let's practice problem solving...

Example – Flag of Denmark

How would we go about programming the Flag of Denmark?

Use the strategies we just discussed to figure out an algorithmic plan!



Today's Learning Goals

Use the tkinter library to construct graphics with programming

Solve unfamiliar problems using algorithmic thinking