#9: Interactive Applications

SAMS PROGRAMMING C

Review from Last Week

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

Build **lists of multiple dimensions**

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

Today's Learning Goals

Create interactive programs with **input/output streams** and **files**

Create interactive graphic programs with **keyboard and mouse events**

Create animations with **time-based events**

Input & Output

When we interact with a computer, we provide the computer with **input**, and it responds with **output**.

This input-output cycle is at the center of all interactive applications. The application must be able to process and respond to human input, just as human users must be able to understand and respond to computer output.

Computer Output

We've already used one mode of computer output: we've had the computer **print** values to the screen.

`print()` sends information to `sys.stdout` (system standard out). This shows up in the interpreter in Pyzo.

Computer Input

So far, to provide input, we've directly typed values into the interpreter or into our files. But we can ask for input during programs as well...

`result = input(message)` displays the given message, then waits for the user to type a response. When the user presses Enter, the typed message is stored as a string in `result`.

```
name = input("Who are you?")
print("Hello " + name + "!")
```

Files as Input/Output

We can also process input and output using files on the computer! Specifically, we can **read** text from files and **write** text back into them. First, we need to create a File object. We do this using the **open** command.

```
f = open(filename, mode)
```

When we're done with a file, remember to always **close** it! Otherwise, we won't be able to open it again later!

```
f.close()
```

Reading/Writing From Files

When opening a file object, if we set the mode to "r" (read), we can read text from it; if we set it to "w" (write), we can write text to it.

```
f = open(filename, "r")
text = f.read()
f.close()
```

```
f = open(filename, "w")
f.write(text)
f.close()
```

Managing errors

When we start allowing users to input values, we can get errors that are the user's fault, not ours. To handle possible user error, we use **error-catching** with **try** and **except**.

```
try:
    <possible breaking code>
except:
    <what to do if the code breaks>
```

Error Catching Example

Say we want to write a program that multiplies the inputted number by 2. We need to make sure to handle the case where the user gives us a non-number!

```
try:
    result = int(input("Enter a number: "))
    print("Answer: ", result * 2)
except:
    print("You have to enter a number!")
```

Adding some randomness...

Finally, it can be helpful to allow for some random behavior when making applications interactive. We can approximate randomness using the **random** library.

https://docs.python.org/3.6/library/random.html

To choose a random number in the range [x, y]:

```
num = random.randint(x, y)
```

To select a random element from a list lst:

```
item = random.choice(lst)
```

Example

Let's program an interactive guessing game!

Interactive Graphics

Just as we can make interactive text applications, we can also make interactive graphic applications!

We primarily interact with graphical applications using the **mouse/trackpad** and the **keyboard**.

New Graphics Starter Code

See https://www.cs.cmu.edu/~112/notes/notes-animations-part2.html#starter-code

Note: you won't be responsible for the run() function. That's just setting everything up for you.

Storing information in data

First, note that draw() has been replaced by redrawAll. Instead of being called once, redrawAll will be called **over and over again**, replacing the picture on the canvas each time it is called.

Instead of width and height, we're given **data**. data will store all the information we need to access in the graphic. In fact, it already stores the width and the height!

```
print(data.width, data.height)
```

To add new information to data, we just say:

```
data.varName = value
```

Handling Mouse events

A mouse event involves two pieces of information: the **x** and **y** coordinates where the mouse/trackpad was clicked on the canvas.

That information is passed along in the **event** parameter, as **event.x** and **event.y**.

```
print(event.x, event.y)
```

We can store that information in **data** to modify things in **redrawAll**!

Storing values over time

When we want to update a value in an interactive graphic over time, we have to give that variable an initial value. This is done in the **init** function, which is called just once, at the very beginning.

Note: to keep track of variables, they have to be stored in data! Otherwise, they're just local variables!

Example: moving a circle

```
def init(data):
    data.currentX = data.width/2
    data.currentY = data.height/2

def mousePressed(event, data):
    data.currentX = event.x
    data.currentY = event.y

def redrawAll(canvas, data):
    canvas.create_oval(data.currentX - 50, data.currentY - 50,
                       data.currentX + 50, data.currentY + 50,
                       fill="lavender")
```

Example: clicking a button

```
def init(data):
    data.buttonX, data.buttonY = data.width/2, data.height/2
    data.buttonSize = 50
    data.buttonClicked = False

def mousePressed(event, data):
    if (data.buttonX - data.buttonSize <= event.x <= data.buttonX + data.buttonSize) and \
       (data.buttonY - data.buttonSize <= event.y <= data.buttonY + data.buttonSize):
        data.buttonClicked = not data.buttonClicked

def redrawAll(canvas, data):
    color = "purple" if data.buttonClicked else "gray"
    canvas.create_rectangle(data.buttonX - data.buttonSize, data.buttonY - data.buttonSize,
                            data.buttonX + data.buttonSize, data.buttonY + data.buttonSize,
                            fill=color)
```

Handling Keyboard events

A keyboard event involves one piece of information: which key is typed.

We can get that key as a single character with **event.char**. Some keys don't have single-character representations, though; for those, we can find special representations in **event.keysym**.

```
print(event.char, event.keysym)
```

Example: displaying typed characters

```
def init(data):
    data.curChar = ""
    data.curKeysym = ""

def keyPressed(event, data):
    data.curChar = event.char
    data.curKeysym = event.keysym

def redrawAll(canvas, data):
    canvas.create_text(data.width/2, data.height/2, font="Arial 32 bold",
                       text=data.curChar + "\n" + data.curKeysym)
```

Example: moving with arrow keys

```
def init(data):
    data.circleX = data.width/2
    data.circleY = data.height/2

def keyPressed(event, data):
    if event.keysym == "Up":        data.circleY -= 20
    elif event.keysym == "Down":    data.circleY += 20
    elif event.keysym == "Left":    data.circleX -= 20
    elif event.keysym == "Right":   data.circleX += 20

def redrawAll(canvas, data):
    canvas.create_oval(data.circleX - 50, data.circleY - 50,
                       data.circleX + 50, data.circleY + 50, fill="salmon")
```

Animation: Changing Data Over Time

Animation is the process of making graphics look like they are moving by changing them slightly as time passes. We can create animations in tkinter too!

We simulate time passing using **timerFired**. This function only takes one parameter, **data**. It gets called every **data.timerDelay** milliseconds; by changing data in timerFired, we can make the data change continuously over time!

Example: tracking time passed

```
def init(data):
    data.timerDelay = 1000 # one second
    data.timeCount = 0

def keyPressed(event, data):
    if event.keysym == "Up":        data.timerDelay *= 2
    elif event.keysym == "Down":    data.timerDelay //= 2

def timerFired(data):
    data.timeCount += 1

def redrawAll(canvas, data):
    s = "Time Passed: " + str(data.timeCount) + "\n" + \
        "timerDelay: " + str(data.timerDelay)
    canvas.create_text(data.width/2, data.height/2, font="Arial 32 bold", text=s)
```

Example: moving shape

```
def init(data):
    data.boxXSpeed = 10
    data.boxX = 10
    data.boxY = data.height/2

def timerFired(data):
    data.boxX += data.boxXSpeed

def redrawAll(canvas, data):
    canvas.create_rectangle(data.boxX - 20, data.boxY - 20,
                            data.boxX + 20, data.boxY + 20, fill="green")
```

Putting it all together

To make a full interactive application or game, we just need to combine all the necessary functions and data!

In these applications and games, we'll often need to store **game state** in data. This will let us keep track of what's currently going on behind the scenes.

Note: never modify the game state in redrawAll! This can lead to nasty, unexpected behaviors. Only modify state in init, keyPressed, mousePressed, and timerFired.

Example: Memory Game

Let's program a memory game!

Game state: a **2D list** holding the values of the cards, a **1D list** holding (row, col) indices of cards that have been flipped, and a **timer**.

Events: clicking to flip a card, pairs of cards flipping back over after time has passed.

Display: all of the cards, either showing their value or grayed out.

Today's Learning Goals

Create interactive programs with **input/output streams**

Create interactive graphic programs with **keyboard and mouse events**

Create animations with **time-based events**