#9: Interaction & Events

SAMS SENIOR CS TRACK

Last Time

Used **strings**, **lists**, and **objects** to hold and modify data.

Understood the difference between **mutable** and **immutable** datatypes, and when **aliasing** occurs.

Today's Learning Goals

Create **interaction** with the computer through use of **input** and **output**.

Use the **Model-View-Controller** framework to organize complex applications.

Capture **events** and use them to make interactive graphics.

Interaction

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 + "!")
```

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:
    <possibly-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. (We'll talk about this in more depth next week).

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 number guessing game!

I'm thinking of a number between 1 and 10. Or rather, the computer is...

Events

Interaction in 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**.

Organizing an Application

As we start to build more complex programs, we'll need to decide how to organize the programs in advance. It's too hard to approach coding something as complex as Tetris or Mario otherwise.

In interactive graphics, we do this using a framework called **MVC: Model-View-Controller**

Model-View-Controller

When you decide to code a complex application, think about how it will change, then consider:

Model – what parts of the application change over time? We'll need to store this data so that we can update it whenever we need to.

View – what parts of the application need to be drawn on the screen? We'll need to refresh the graphics every time the model changes.

Controller – what causes the model to change? And how does it change as a result? We'll need to capture these **events** and change the model appropriately.

MVC Example

How would we break the game Sudoku down according to the MVC structure?

Game: http://www.logicgamesonline.com/sudoku/

Model:

View:

Controller:

New Graphics Starter Code

See http://krivers.net/15112-s19/notes/notes-animations-part1.html#starter-code

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

Model

We need to access the model in both the view and the controllers. Therefore, we'll create a simple object that can be provided to all functions. This object, called **data**, will alias across functions. When we update it in the controller, it will also update in the view!

We need to put all the necessary information in data at the start of the program. To do this, we write a function `init(data)` which is only called once, at the very beginning. In init, we can add new model variables with:

```
data.varName = value
```

Note that data starts with the width and height already stored in it.

View

We'll update the view in the function `redrawAll(canvas, data)`. This is just like the draw() functions we've already written, except that it will be called **over and over again**, every time the controllers change a value in data.

By calling redrawAll multiple times and changing the values in data, we can make the graphics appear to change, or move, based on our interaction!

Event Loop

Your computer is always waiting to capture a variety of **events** that can happen from different input sources. (Pressing a key, moving the trackpad, clicking the mouse, plugging in the power...)

The computer can then forward some of those events to our program, so we can interpret them. In interactive graphics, we'll ask the computer to forward **mouse click** events and **keyboard press** events.

Our program will listen for those forwarded events and will call special **controller functions** when it receives them. It will also update the view every time a change occurs.

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**.

```
def mousePressed(event, data):
    print(event.x, event.y)
```

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

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**.

```
def keyPressed(event, data):
    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")
```

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 and the controllers.

Today's Learning Goals

Create **interaction** with the computer through use of **input** and **output**.

Use the **Model-View-Controller** framework to organize complex applications.

Capture **events** and use them to make interactive graphics.