#3-3: User Interaction

CS SCHOLARS – PROGRAMMING

Learning Goals

Create interactive programs using **text-based interaction** to process user input

Create interactive programs using **event-based interaction** to support user interaction

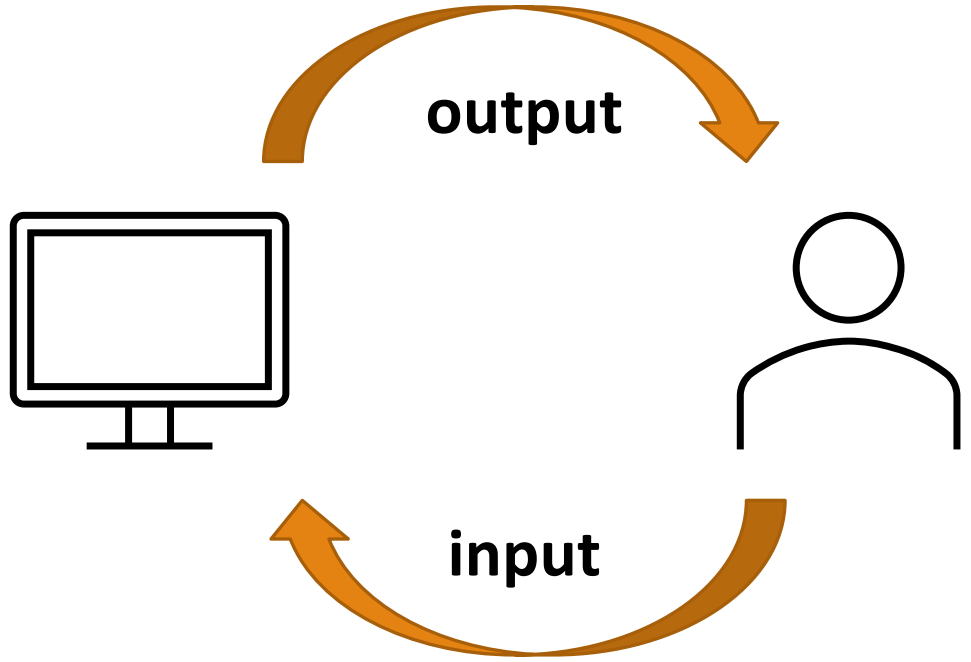
Text-Based Interaction

Interacting with the User

Now we have everything that we need to build programs that can authentically interact with the person using them (the 'user').

When we build an interactive program, we have to take **input** from the user, process it in the system, and use it to produce **output** that will be shown to the user in some way.

This process repeats indefinitely until some goal is met.



Text-based Interaction

One way to process input and create output is via **text-based interaction**.

Get all user input through the `input` function we learned previously. Produce all output as text displayed via the `print` function to the screen.

To repeat the procedure, use a loop. Since we usually do not know exactly how many repetitions will be needed, a `while` loop is often used (with the loop control variable based on the user's input).

Example: Data Entry

Let's write a simple program that gets multiple inputs from the user and process them like a data stream.

We'll need to give the user a way to signal that they're done entering numbers. This can by done with a special input, like the string `'q'`.

Note that a new input is collected and a new output is shown in every iteration of the loop.

```
numbers = []

value = input("Enter a number, or q to quit:")

while value != "q":

    num = int(value)

    numbers.append(num)

    print("Current numbers:", numbers)

    value = input("Enter a number, or q to quit:")

print("Total sum:", sum(numbers))
```

Validating Input

We may sometimes need to **validate** user input, to make sure it matches the requirements.

For example, in the data entry program, we might want to ensure that the input actually is a number, and a positive number, before accepting it.

It often helps to move processing user input into a **helper function**, to avoid overcomplicating the original function.

```
def getUserInput():
    while True: # returning will stop loop
        value = input("Enter a number, or q to quit:")
        if value == "q":
            return value
        elif value.isnumeric():
            value = int(value)
            if value > 0:
                return value

def processData():
    numbers = []
    value = getUserInput()
    while value != "q":
        numbers.append(value)
        print("Current numbers:", numbers)
        value = getUserInput()
    print("Total sum:", sum(numbers))
```

Activity: reorderList

You do: write a function that takes a list of strings and asks the user to enter a new order for the strings, one value at a time. For example, if we call `reorderList(["a", "b", "c", "d"])`, the user could enter `c`, then `a`, then `d`, then `b`, and the function would return the list `[ "c", "a", "d", "b" ]`.

Make sure your program has clear requests for input, understandable output, a well-managed interaction loop, and validates the user input!

Event-Based Interaction

Event-Based Interaction

Most programs that you interact with use much more than just text! They may let you use the **mouse** to interact with the screen and the **keyboard** to provide other inputs (like directions with the arrow keys). The program then probably displays results visually, not just through text.

That's what we'll work on next – how to write a program that can capture more complex user events and produce more complex output. We'll do this with an **interaction framework** that uses Tkinter to display graphics and accept user mouse and keyboard input.

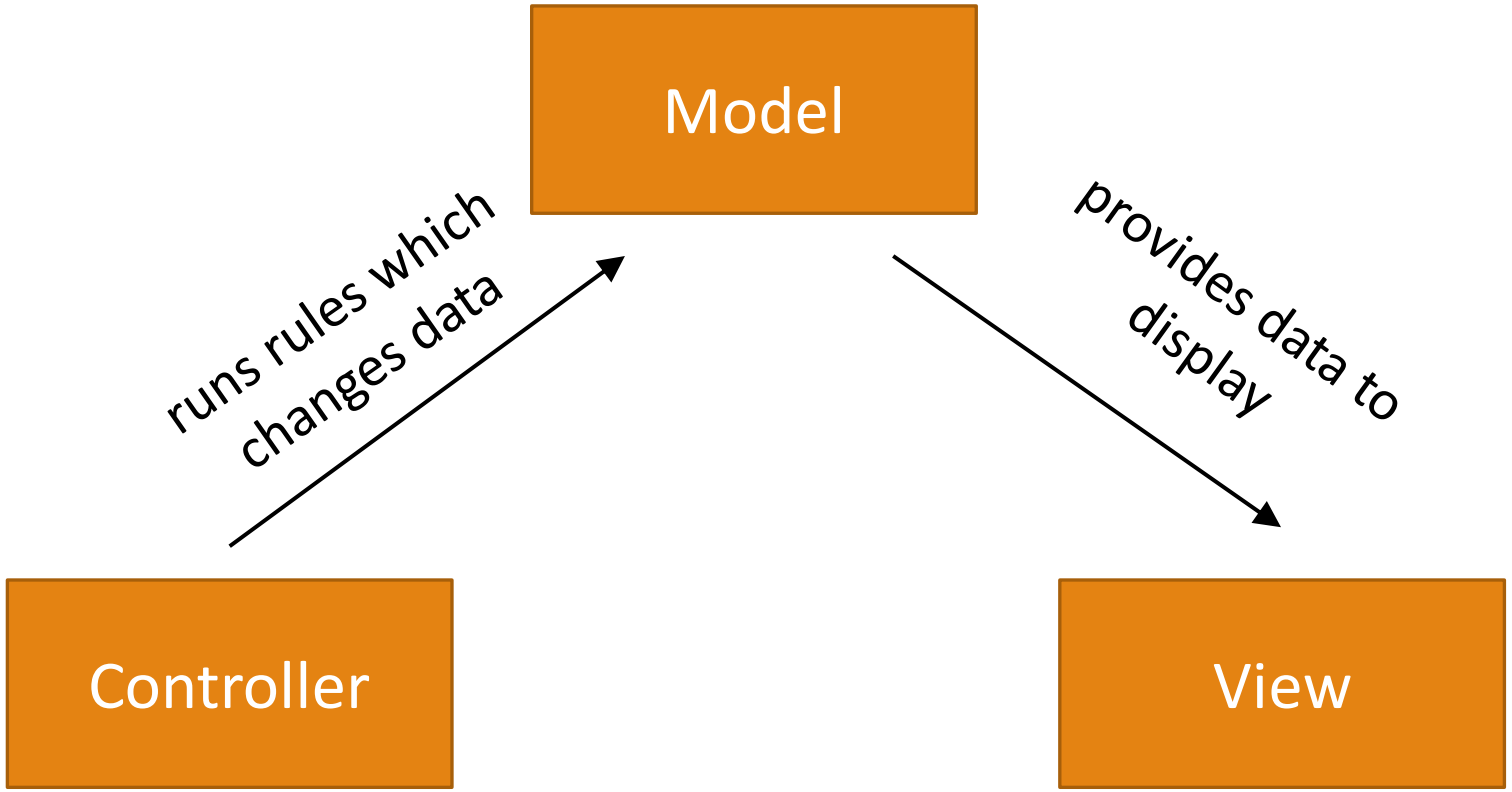
Interaction Parts in Code

Our interaction code will be composed of three parts:

- A **model** which stores the core data used in the interaction in a shared data structure
- Event **controllers** which run rules that update the model components when user events occur
- A graphical **view** which repeatedly displays the current state of the model

All three of these parts will be organized in an **interaction framework** which sets up the initial model, manages the controllers, and updates the view as needed. That framework is provided for you, but you'll need to fill in the parts!

Model, View, Controller



Making the Components

We need to be able to pass the whole model around the code as a single variable. We'll do this by creating an **object** called `data` and adding components to that object.

These components will act just like variables; the only difference is that we'll use `data.componentName` instead of `componentName` by itself. It's similar to when we import a library or call a method on a list. For example, to store information about a circle that represents some part of the model, we could set:

```
data.x = 200
data.y = 200
data.r = 50
```

By storing all the components in one structure we can pass the same structure around to all the functions we write as a single parameter. This structure will be **mutable** (like lists!), so we'll be able to update it directly in the rule functions, then display the updated data in the view function.

Displaying the Model

To display the whole model, we'll use Tkinter to draw graphics that represent the components visually. By referring to component values in `data` in the view function, we can make graphics that change alongside the model.

For example, if `data.x = 200`, `data.y = 200`, and `data.r = 50`, we could draw a circle with:

```
canvas.create_oval(data.x – data.r, data.y – data.r,
                   data.x + data.r, data.y + data.r)
```

We'll erase and re-draw the graphics window every time the rules of the program run. If we change the components a little bit at a time, this makes the display appear to update smoothly.

Running the Rules

We can run the either when a **mouse event** happens, or when a **keyboard event** happens.

When you take an action on your computer, a **signal** is sent from the computer hardware to any programs that are currently running. That signal has information about the type of the event (key press vs. mouse click), plus any additional information that might be useful (which key was pressed).

Once this signal is received, it triggers a function. If that function changes the model's components in `data`, this will simulate the model changing over time!

```
data.x = data.x + 5 # move the circle to the right on mouse click
```

Interaction Functions

Now we have everything we need for the **interaction framework**. You can find starter code for the framework linked on the course website. For each interactive program you make, start with the starter code, then update four functions to build a simple simulation:

- `init(data)` makes the original components. `data` is the model object
- `keyPressed(data, event)` and `mousePressed(data, event)` run the rules to update `data`. `event` holds information about the event.
- `redrawAll(canvas, data)` displays the model. `canvas` is a Tkinter canvas

This is different from the code we're used to because the functions **work together** instead of running in a sequential order.

Simple Example – Color-Changing Ball

Let's start with a simple example. Say we want to draw a circle and have the color of the circle change every time the user clicks the mouse.

The **model** should track any values that might change. In this case, that's the **color** of the circle. Set an initial component value in `init`.

The **rules** should describe how the model changes over time. In this case, we **change the color** in the shared data model with every call to `mousePressed`.

The **view** should draw a circle in the middle of the window and set its color based on the color in the model. This is done in `redrawAll`.

Simple Example Code

```
def init(data):
    # put variables in data here
    data.color = "red"

def redrawAll(canvas, data):
    # (200, 200) is center point
    # make sure to reference data for the parts that change!
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data.color)

def mousePressed(data, event):
    import random
    # Let's pick a color randomly!
    newColor = random.choice(["red", "orange", "yellow",
                              "green", "blue", "purple"])
    data.color = newColor # update data to change the model
```

Activity: Make the circle grow

You do: open the interaction starter code and copy in the functions from the previous slide. Run the code to make sure it works, then modify the code in the three functions so that the circle **grows larger** each time a key is typed.

Hint: you'll need to add one **component** to the model, the thing that is changing. You should change that component in `keyPressed` and access it while drawing the circle in `redrawAll`.

keyPressed Events

We can use the `event` parameter in the rules functions to create more personalized interaction!

In `keyPressed`, the `event` parameter contains two values we can access with a `.` (like string or list methods and the `data` components):

- `event.char` is a string that holds the character pressed
- `event.keysym` is a string that holds the 'name' of the character, for characters we can't show in a string (e.g., Enter or BackSpace)

If we want to draw the last-pressed character in the middle of the screen, for example, we would store that character in `data`, then draw it in `redrawAll`:

```
def keyPressed(event, data):
    data.text = event.char
```

Example Key Event

```
def init(data):
    data.color = "red"
    data.tmp = "" # need to hold partial strings

def redrawAll(canvas, data):
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data.color)

def keyPressed(event, data):
     # build up a color string one char at a time until user presses Return
     if event.keysym != "Return":
        data.tmp += event.char
     else:
        # move the color into data.color
        data.color = data.tmp
        data.tmp = ""
```

Activity: move circle up/down

You do: take the simulation code from the last activity (color-changing circle) and update it so that the circle moves **up** when the user presses the up key and **down** when the user presses the down key. The circle no longer needs to change colors when other keys are pressed.

Note: you should use `event.keysym`. You'll be able to check it against `"Up"` and `"Down"`.

`mousePressed` Events

In `mousePressed`, the `event` parameter holds the pixel location where the user clicked on the canvas.

- `event.x` is the x location
- `event.y` is the y location

If we want to move a circle around the canvas to be centered wherever you click, we'd need to store the center location and draw the circle based on the model location in `redrawAll`:

```
def mousePressed(event, data):
    data.cx = event.x
    data.cy = event.y
```

Example Mouse Event

```
def init(data):
    data.color = "red"

def redrawAll(canvas, data):
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data.color)

def mousePressed(event, data):
    import random
    newColor = random.choice(["red", "orange", "yellow",
                              "green", "blue", "purple"])
    # Check if the user clicked inside the circle
    # Is the distance between the center and the click less than the radius?
    if ((event.x - 200)**2 + (event.y - 200)**2)**0.5 <= 50:
        data.color = newColor
```

Activity: make circle shrink

You do: take your code from the previous activity and modify it so that the circle **shrinks** whenever the user clicks inside it. (If the user clicks outside, it can change colors instead).

You can start with the bounds check from the previous slide, but you'll need to change what happens in the conditional body!

Example: Circle Application

Now we can use all of this together to build an interactive program that does something interesting!

Let's implement a simple application that generates a new circle of random size and color every time the user clicks on the screen. Every time the user clicks Backspace, a random circle is deleted.

```
def init(data):
    data.circles = []

def redrawAll(canvas, data):
    for circle in data.circles:
        [x, y, size, color] = circle
        canvas.create_oval(x - size, y - size,
                           x + size, y + size, fill=color)

def keyPressed(event, data):
    if event.keysym == "BackSpace":
        if len(data.circles) > 0:
            index = random.randint(0, len(data.circles)-1)
            data.circles.remove(data.circles[index])

def mousePressed(event, data):
    x = event.x
    y = event.y
    size = random.randint(5, 50)
    color = random.choice(["red", "orange", "yellow",
                           "green", "blue", "purple"])
    data.circles.append([x, y, size, color])
```

Learning Goals

Create interactive programs using **text-based interaction** to process user input

Create interactive programs using **event-based interaction** to support user interaction

Sidebar: Controller Functions – Event Loop

The event controller runs an **event loop** to capture the signals that the computer sends out. To implement this event loop, we'll have our interaction system constantly **listen** for events.

When an event occurs, the controller will catch it and send the event data on to the correct rule function; then it will tell the view to update. This is done with a special kind of Tkinter function called `bind` and is provided in the starter code.

With Tkinter we can listen for and bind functions to lots of different event types. We'll care about just two: `<Key>`, a key press, and `<Button-1>`, a left mouse click. There are lots of other Tkinter events we can implement if we want them:

https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/event-types.html