Raspberry Pi Activity: Room Adventure...Revolutions

In this activity, you will modify the Room Adventure game that you created and extended in a previous RPi activity. You will need the following items:

- Raspberry Pi B v3 with power adapter;
- LCD touchscreen; and
- Wireless keyboard and mouse with USB dongle.

If you wish, you can simply bring your laptop with the Python interpreter (and also perhaps IDLE) installed since you will not be using the GPIO pins on the RPi.

The game

The basic premise of the game will not change from the previous activity. Again, the setting of the game is a small "mansion" consisting of four rooms. Each room has various exits that lead to other rooms in the mansion. In addition, each room has items, some of which can simply be observed, and others that can be picked up and added to the player's inventory. For this activity, there is no actual *goal* for the player other than to move about the mansion, observe various items throughout the rooms in the mansion, and add various items found in the rooms to inventory. There is an end state that results in death, however! Of course, this doesn't prevent extending the game with a better story and more variety.

The four rooms are laid out in a simple square pattern. Room 1 is at the top-left of the mansion, room 2 is at the top-right, room 3 is at the bottom-left, and room four is at the bottom-right. Each room has items that can be observed:

- Room 1: A chair and a table;
- Room 2: A rug and a fireplace;
- Room 3: Some bookshelves, a statue, and a desk; and
- Room 4: A brew rig (you know, to brew some delicious libations).

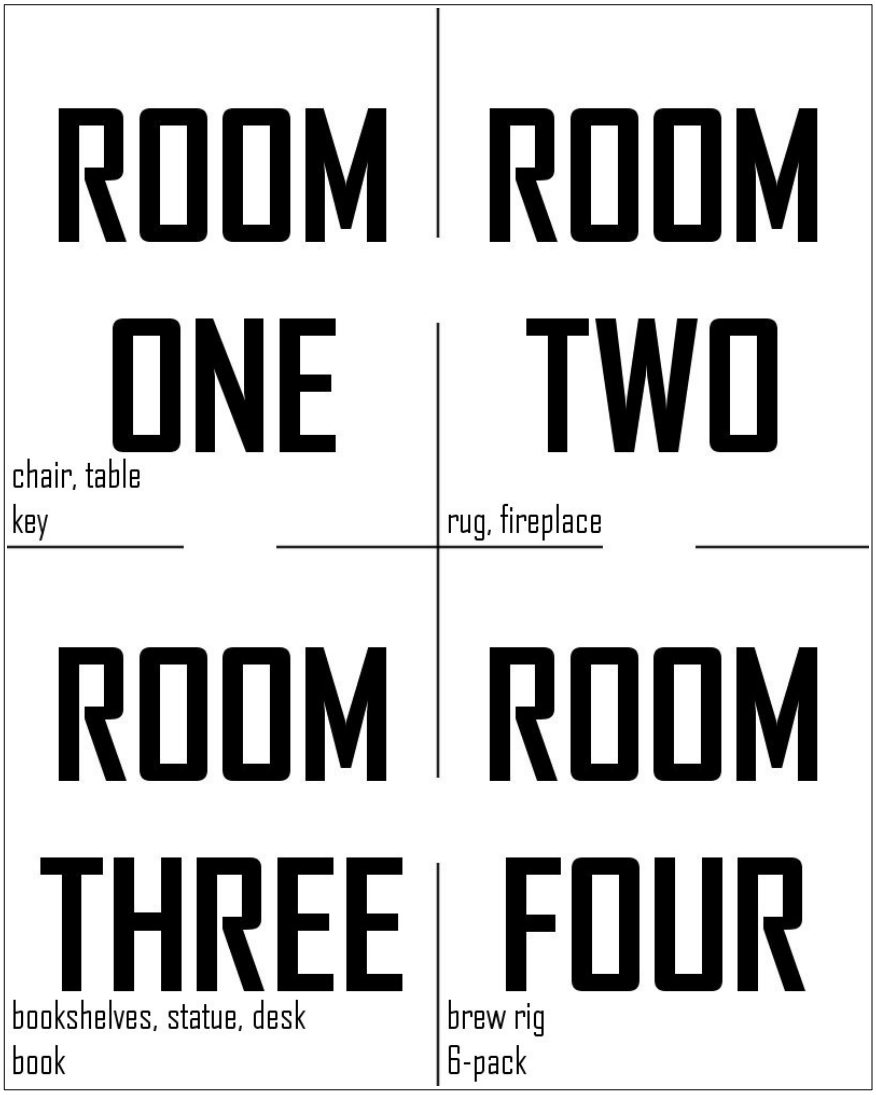
Observable items can provide useful information (once observed) and may reveal new items (some of which can be placed in the player's inventory). In addition, each room may have some items that can be *grabbed* by the player and placed in inventory:

- Room 1: A key;
- Room 3: A book; and
- Room 4: A 6-pack of a recently brewed beverage.

The rooms have various exits that lead to other rooms in the mansion:

- Room 1: An exit to the east that leads to to room 2, and an exit to the south that leads to room 3;
- Room 2: An exit to the south that leads to room 4, and an exit to the west that leads to room 1;
- Room 3: An exit to the north that leads to room 1, and an exit to the east that leads to room 4; and
- Room 4: An exit to the north that leads to room 2, an exit to the west that leads to room 3, and an (unlabeled) exit to the south that leads to...death! Think of it as jumping out of a window.

Here's the layout of the mansion:



The gameplay
The game is text-based, although this activity adds a GUI that integrates the player's input, the status of the room, and an image associated with the room. Information such as which room the player is located in, what objects are in the current room, and so on, is continually provided throughout the game. The player is prompted for an action (i.e., what to do) after which the current situation is updated.

As was the case last time, the game supports a simple vocabulary for the player's actions that is composed of a verb followed by a noun. For example, the action “go south” instructs the player to take the south exit in the current room (if that is a valid exit). If the specified exit is invalid (or, for example, if the player misspells an action), an appropriate response is provided, instructing the player of the accepted vocabulary. Supported verbs are: *go*, *look*, and *take*. Supported nouns depend on the verb; for example, for the verb *go*, the nouns *north*, *east*, *south*, and *west* are supported. This will allow the player to structure the following *go* commands:

- go north

- go east
- go south
- go west

The verbs *look* and *take* support a variety of nouns that depend on the actual items located in the rooms of the mansion. The player cannot, for example, “look table” in a room that doesn't have a table! Some examples of *look* and *take* actions are:

- look table
- take key

The gameplay depends on the user's input. Rooms change based on valid *go* actions, meaningful information is provided based on valid *look* actions, and inventory is accumulated based on valid *take* actions. For this game, gameplay can continue forever or until the player decides to “go south” in room 4 and effectively jump out of the window to his/her death:

At any time, the player may issue the following actions to leave the game:

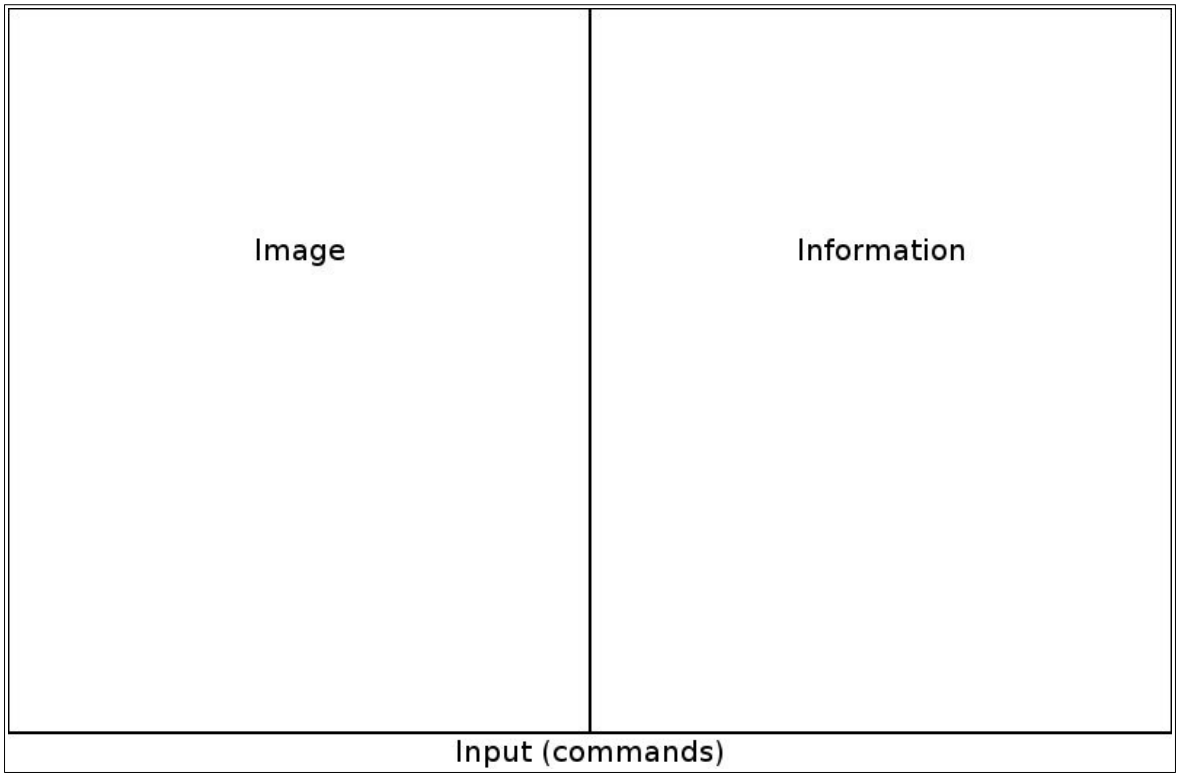
- quit
- exit
- bye
- sionara!

The GUI

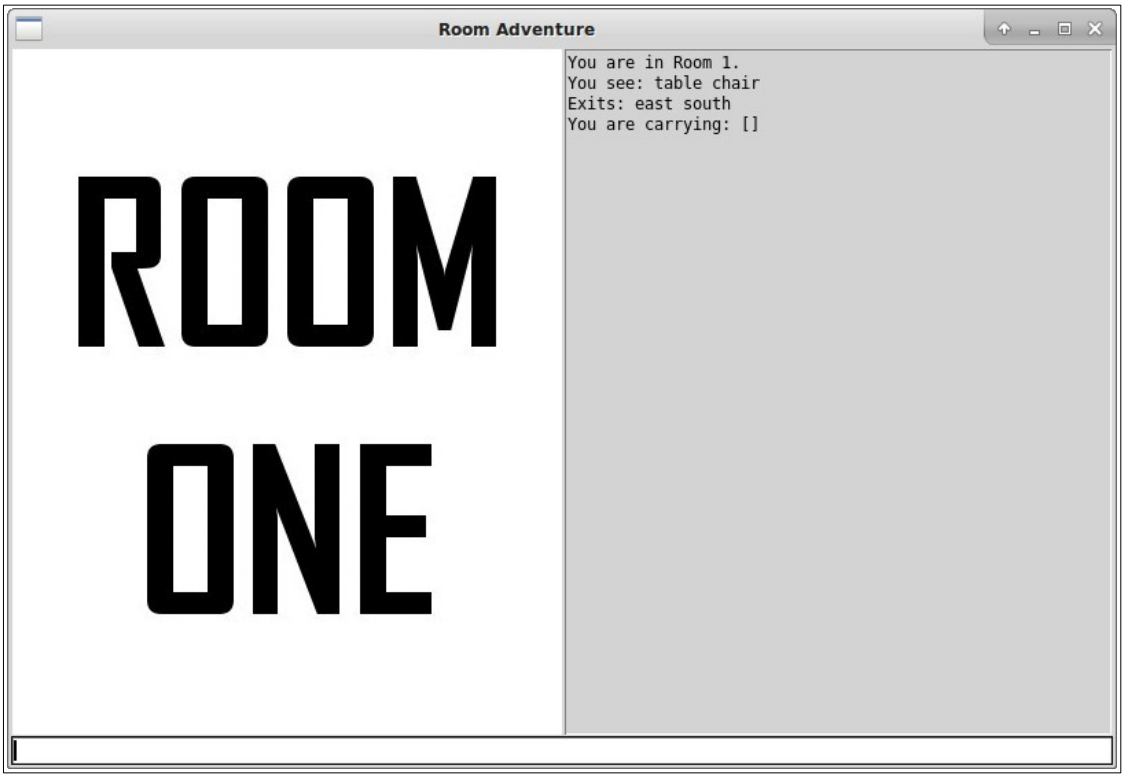
The GUI is split up into three mains parts:

- A text-based representation of the current room that contains its description, items located in the room, and grabbables in the room that can be taken into inventory – along with the player's inventory and some relevant current status;
- An image that provides a static picture of the room; and
- An input action section that takes player input for processing.

Here's a mock-up of the interface:



And here's what the GUI actually looks like once the game is completed and running:



The code
It turns out that a lot of the source code designed in the last Room Adventure activity can be reused in this activity as well. Some of it, however, will need to be moved around and/or redesigned so that the GUI can work seamlessly with the backend (e.g., rooms, items, inventory, player input, etc).

Instead of starting with the source code from the previous activity, we will simply build the new Room Adventure game from scratch by integrating what we can from the old version. So let's begin with the main part of the program (which turns out to be just a few lines of code!):

```
################################################################
# Name:
# Date:
# Description:
################################################################
from tkinter import *

...the rest of the code will go here...

################################################################
# the default size of the GUI is 800x600
WIDTH = 800
HEIGHT = 600

# create the window
window = Tk()
window.title("Room Adventure")

# create the GUI as a Tkinter canvas inside the window
g = Game(window)
# play the game
g.play()

# wait for the window to close
window.mainloop()
```

Note that, in the final program, this source code will be located at the bottom. To start things off, we import Tkinter so that the GUI can be created. In fact, the GUI is created much like the examples in the lesson on GUIs. The width and height of the GUI are defined as constants. A new instance of the class **Tk** is then created. This *window* is given an appropriate title. The rest of the GUI components are defined in a **Game** class, which is instantiated as the variable `g`. The **Game** class will have a `play` method that will kick things off and officially start the game. Of course, the window must remain on the player's desktop until it is closed.

The remainder of the code will be inserted in between the header (and import statements) and the main part of the program. Let's begin with the **Room** class (which happens to be almost exactly the same as the modified version that uses dictionaries instead of parallel lists). Changes are highlighted:

```
# the room class
class Room:
```

```
# the constructor
def __init__(self, name, image):
     # rooms have a name, an image (the name of a file),
     # exits (e.g., south), exit locations
     # (e.g., to the south is room n), items (e.g., table),
     # item descriptions (for each item), and grabbables
     # (things that can be taken into inventory)
     self.name = name
     self.image = image
     self.exits = {}
     self.items = {}
     self.grabbables = []

# getters and setters for the instance variables
@property
def name(self):
     return self._name

@name.setter
def name(self, value):
     self._name = value

@property
def image(self):
     return self._image

@image.setter
def image(self, value):
     self._image = value

@property
def exits(self):
     return self._exits

@exits.setter
def exits(self, value):
     self._exits = value

@property
def items(self):
     return self._items

@items.setter
def items(self, value):
     self._items = value

@property
def grabbables(self):
```

```
        return self._grabbables

    @grabbables.setter
    def grabbables(self, value):
        self._grabbables = value

    # adds an exit to the room
    # the exit is a string (e.g., north)
    # the room is an instance of a room
    def addExit(self, exit, room):
        # append the exit and room to the appropriate
        # dictionary
        self._exits[exit] = room

    # adds an item to the room
    # the item is a string (e.g., table)
    # the desc is a string that describes the item (e.g., it is
    # made of wood)
    def addItem(self, item, desc):
        # append the item and description to the appropriate
        # dictionary
        self._items[item] = desc

    # adds a grabbable item to the room
    # the item is a string (e.g., key)
    def addGrabbable(self, item):
        # append the item to the list
        self._grabbables.append(item)

    # removes a grabbable item from the room
    # the item is a string (e.g., key)
    def delGrabbable(self, item):
        # remove the item from the list
        self._grabbables.remove(item)

    # returns a string description of the room
    def __str__(self):
        # first, the room name
        s = "You are in {}.\n".format(self.name)

        # next, the items in the room
        s += "You see: "
        for item in self.items.keys():
            s += item + " "
        s += "\n"

        # next, the exits from the room
        s += "Exits: "
```

```
            for exit in self.exits.keys():
                 s += exit + " "

            return s
```

The only change to the **Room** class is the addition of a new instance variable that represents a room's image. As such, the constructor has minor changes, and an accessor and mutator is also provided for the new instance variable.

The Game class is quite involved, so we'll build it a little at a time. Let's begin with a shell:

```
# the game class
# inherits from the Frame class of Tkinter
class Game(Frame):
     # the constructor
     def __init__(self, parent):
          # call the constructor in the superclass
          Frame.__init__(self, parent)

     # creates the rooms
     def createRooms(self):
          ...

     # sets up the GUI
     def setupGUI(self):
          ...

     # set the current room image
     def setRoomImage(self):
          ...

     # sets the status displayed on the right of the GUI
     def setStatus(self, status):
          ...

     # play the game
     def play(self):
          # add the rooms to the game
          self.createRooms()
          # configure the GUI
          self.setupGUI()
          # set the current room
          self.setRoomImage()
          # set the current status
          self.setStatus("")

     # processes the player's input
     def process(self, event):
          ...
```

Overall, the **Game** class is almost self-explanatory. It inherits from Tkinter's **Frame** class in order to support GUI components. By default, the class variables `WIDTH` and `HEIGHT` define a GUI that is 800 pixels wide by 600 pixels high. The constructor simply calls the constructor of its superclass (Tkinter's **Frame** class). Several method *stubs* (i.e., temporary templates) are provided to create the rooms (these are the actual room objects that provide backend support for the game), to setup the GUI components, to set the room image on the GUI (as the player moves around the mansion), to set the current status (as the player's input is processed), and to process the player's input. These functions will be covered later.

The play function is brief. It first creates the rooms, sets up the GUI, sets the current room image, and sets the current status. This is all that's required to begin playing the game. Creating the rooms will setup each room's exits, images, items, and grabbables. Setting up the GUI will create the GUI components and display the GUI. Setting the room image will display the appropriate image to the left of the GUI. Setting the current status will display the appropriate status to the right of the GUI.

Let's continue by implementing the `createRooms` function:

```
    # creates the rooms
    def createRooms(self):
        # r1 through r4 are the four rooms in the mansion
        # currentRoom is the room the player is currently in (which
        # can be one of r1 through r4)

        # create the rooms and give them meaningful names and an
        # image in the current directory
        r1 = Room("Room 1", "room1.gif")
        r2 = Room("Room 2", "room2.gif")
        r3 = Room("Room 3", "room3.gif")
        r4 = Room("Room 4", "room4.gif")

        # add exits to room 1
        r1.addExit("east", r2)   # to the east of room 1 is room 2
        r1.addExit("south", r3)
        # add grabbables to room 1
        r1.addGrabbable("key")
        # add items to room 1
        r1.addItem("chair", "It is made of wicker and no one is
sitting on it.")
        r1.addItem("table", "It is made of oak.  A golden key rests
on it.")

        # add exits to room 2
        r2.addExit("west", r1)
        r2.addExit("south", r4)
        # add items to room 2
        r2.addItem("rug", "It is nice and Indian. It also needs to
be vacuumed.")
        r2.addItem("fireplace", "It is full of ashes.")
```

```
        # add exits to room 3
        r3.addExit("north", r1)
        r3.addExit("east", r4)
        # add grabbables to room 3
        r3.addGrabbable("book")
        # add items to room 3
        r3.addItem("bookshelves", "They are empty.  Go figure.")
        r3.addItem("statue", "There is nothing special about it.")
        r3.addItem("desk", "The statue is resting on it.  So is a
book.")

        # add exits to room 4
        r4.addExit("north", r2)
        r4.addExit("west", r3)
        r4.addExit("south", None)     # DEATH!
        # add grabbables to room 4
        r4.addGrabbable("6-pack")
        # add items to room 4
        r4.addItem("brew_rig", "Gourd is brewing some sort of
oatmeal stout on the brew rig.  A 6-pack is resting beside it.")

        # set room 1 as the current room at the beginning of the
        # game
        Game.currentRoom = r1

        # initialize the player's inventory
        Game.inventory = []
```

This updated function is almost exactly the same as in the last Room Adventure program. Differences have been highlighted. Since the function is in the **Game** class, then the variable `self` must be passed in to it. This is standard procedure for most methods in classes.

When the rooms are instantiated, a string representing the file name of an associated image must also be passed in. This is due to the change in the constructor of the **Room** class (which now takes this as a parameter so that an image can be associated with a room). At the end of the function, the current room is set (as `r1`). The `currentRoom` variable is now a class variable (i.e., valid for the entire class). As such, it must be referred to using the class name: `Game.currentRoom`. Finally, the player's inventory is initialized here (also a class variable).

The next function that we'll implement is `setupGUI`:

```
    # sets up the GUI
    def setupGUI(self):
        # organize the GUI
        self.pack(fill=BOTH, expand=1)

        # setup the player input at the bottom of the GUI
```

```
        # the widget is a Tkinter Entry
        # set its background to white and bind the return key to the
        # function process in the class
        # push it to the bottom of the GUI and let it fill
        # horizontally
        # give it focus so the player doesn't have to click on it
        Game.player_input = Entry(self, bg="white")
        Game.player_input.bind("<Return>", self.process)
        Game.player_input.pack(side=BOTTOM, fill=X)
        Game.player_input.focus()

        # setup the image to the left of the GUI
        # the widget is a Tkinter Label
        # don't let the image control the widget's size
        img = None
        Game.image = Label(self, width=WIDTH // 2, image=img)
        Game.image.image = img
        Game.image.pack(side=LEFT, fill=Y)
        Game.image.pack_propagate(False)

        # setup the text to the right of the GUI
        # first, the frame in which the text will be placed
        text_frame = Frame(self, width=WIDTH // 2)
        # the widget is a Tkinter Text
        # disable it by default
        # don't let the widget control the frame's size
        Game.text = Text(text_frame, bg="lightgrey", state=DISABLED)
        Game.text.pack(fill=Y, expand=1)
        text_frame.pack(side=RIGHT, fill=Y)
        text_frame.pack_propagate(False)
```

The comments in the source code explain what's going on. Basically, a Tkinter **Frame** will contain the three main GUI components. We make sure that the frame is the same size as the window. Next, the player input section at the bottom of the GUI is created. The Tkinter widget that is used is the **Entry** widget. It is configured and bound to the `process` function (that we'll cover later). That is, the desired behavior is to invoke the `process` function when the player types an action and presses Enter/Return. In addition, this widget is given focus so that the player doesn't have to keep clicking on it to type an action.

The left of the GUI is then setup. In Tkinter, we can display images in a **Label** widget. We set its width to half of the width of the window and add its image (initially `None`). The statement `Game.image.pack_propagate(False)` ensures that no component or widget inside the label will affect its size. That is, the size of the left side of the GUI will remain fixed.

Finally, the right of the GUI is setup. To properly display the current game status, we place a Tkinter **Text** widget inside of a Tkinter **Frame**. We must also ensure that the size of this widget remains fixed.

Next, let's work on the `setRoomImage` function that sets the appropriate image on the left of the GUI:

```
     # set the current room image
     def setRoomImage(self):
          if (Game.currentRoom == None):
               # if dead, set the skull image
               Game.img = PhotoImage(file="skull.gif")
          else:
               # otherwise grab the image for the current room
               Game.img = PhotoImage(file=Game.currentRoom.image)

          # display the image on the left of the GUI
          Game.image.config(image=Game.img)
          Game.image.image = Game.img
```

This function first checks if the current room is valid. If set to `None`, the implication is that the player has died (i.e., jumped out of the window by going south in room 4). Either way, the appropriate image is displayed on the **Label** widget on the left side of the GUI

Now, let's work on the `setStatus` function that displays the current game status on the right side of the GUI:

```
     # sets the status displayed on the right of the GUI
     def setStatus(self, status):
          # enable the text widget, clear it, set it, and disabled it
          Game.text.config(state=NORMAL)
          Game.text.delete("1.0", END)
          if (Game.currentRoom == None):
               # if dead, let the player know
               Game.text.insert(END, "You are dead.  The only thing
you can do now is quit.\n")
          else:
               # otherwise, display the appropriate status
               Game.text.insert(END, str(Game.currentRoom) +\
                    "\nYou are carrying: " + str(Game.inventory) +\
                    "\n\n" +  status)
          Game.text.config(state=DISABLED)
```

Since the **Text** widget is disabled so that the player cannot edit it, it must first be enabled in so that its contents can be changed. Its current text is then deleted, the new status is added, and the widget is once again disabled. Again, if the player is dead, the current room is set to `None`, and we can easily determine what the current status should be.

The last function that needs to be implemented is the `process` function. It serves as the driver that continuously processes the player's input. In fact, it is invoked each time the player presses Enter/Return in the player action input section at the bottom of the GUI:

```
     # processes the player's input
     def process(self, event):
          # grab the player's input from the input at the bottom of
          # the GUI
          action = Game.player_input.get()
          # set the user's input to lowercase to make it easier to
          # compare the verb and noun to known values
          action = action.lower()
          # set a default response
          response = "I don't understand.  Try verb noun.  Valid verbs
are go, look, and take"

          # exit the game if the player wants to leave (supports quit,
          # exit, and bye)
          if (action == "quit" or action == "exit" or action == "bye"\
               or action == "sionara!"):
               exit(0)

          # if the player is dead if goes/went south from room 4
          if (Game.currentRoom == None):
               # clear the player's input
               Game.player_input.delete(0, END)
               return

          # split the user input into words (words are separated by
          # spaces) and store the words in a list
          words = action.split()

          # the game only understands two word inputs
          if (len(words) == 2):
               # isolate the verb and noun
               verb = words[0]
               noun = words[1]

               # the verb is: go
               if (verb == "go"):
                    # set a default response
                    response = "Invalid exit."

                    # check for valid exits in the current room
                    if (noun in Game.currentRoom.exits):
                         # if one is found, change the current room to
                         # the one that is associated with the
                         # specified exit
                         Game.currentRoom =\
```

```
                    Game.currentRoom.exits[noun]
                # set the response (success)
                response = "Room changed."
        # the verb is: look
        elif (verb == "look"):
            # set a default response
            response = "I don't see that item."

            # check for valid items in the current room
            if (noun in Game.currentRoom.items):
                # if one is found, set the response to the
                # item's description
                response = Game.currentRoom.items[noun]
        # the verb is: take
        elif (verb == "take"):
            # set a default response
            response = "I don't see that item."

            # check for valid grabbable items in the current
            # room
            for grabbable in Game.currentRoom.grabbables:
                # a valid grabbable item is found
                if (noun == grabbable):
                    # add the grabbable item to the player's
                    # inventory
                    Game.inventory.append(grabbable)
                    # remove the grabbable item from the
                    # room
                    Game.currentRoom.delGrabbable(grabbable)
                    # set the response (success)
                    response = "Item grabbed."
                    # no need to check any more grabbable
                    # items
                    break

    # display the response on the right of the GUI
    # display the room's image on the left of the GUI
    # clear the player's input
    self.setStatus(response)
    self.setRoomImage()
    Game.player_input.delete(0, END)
```

You may notice that this function is quite similar to the source code in the last version of the Room Adventure game (that uses dictionaries). Changes are **not** highlighted because there are simply too many of them. In addition, much of the code has been slightly modified and reordered.

The function first grabs the player's input from the bottom of the GUI and converts it to lowercase. A default response is set. If the player wishes to quit, we do so. If the player is dead, we clear the player's

input and do nothing. At this point, the image on the left of the GUI is the skull, and the only thing that the player can do is to quit the game.

Next, the player's input is split into two words: verb noun. The verb is then tested. If the player wishes to go in the direction of another room, it becomes the current room; otherwise, an error response is generated. If the player wishes to look at a valid item in the room, the item's description is set as the response. If the player wishes to take a valid item in the room, the item is added to the player's inventory. All of these things occurred in the previous version of the Room Adventure game.

Finally, the response is set as the current status and is updated in the **Text** widget, the room's image is updated in the **Label** widget, and the player's input is cleared in the **Entry** widget.

Homework: Room Adventure...Revolutions

For the homework portion of this activity, you may have the option to work in **groups** (pending prof approval). It is suggested that groups contain at least one confident Python coder.

Your task is to re-implement the Room Adventure improvements that you made in your text-based version of the game – in this new GUI version of the game. Of course, feel free to add new improvements! Note that this component is worth **one-third of your grade** for this assignment! So be creative and precise. You will definitely want to implement significant improvements, since your grade for this portion will be comparatively assigned (i.e., the best improvements to the game earn the best grade for this portion of the assignment).

Clearly, you will need to add images for each room. Note that images should be 400x550 pixels. The height of 550 pixels is slightly flexible (e.g., 500 or 600 pixels should also work). **Optionally**, you can add sound to the game. An idea would be to have a different sound play in each room! Note that this is a bit more difficult and is entirely optional.

Some **warnings about using sound** in your game, if you decide to do this. For adding the ability to play sound, you are encouraged to use **pygame**. Specifically, you are **not** allowed to use anything specific to Windows. This is a Raspberry Pi activity, so if it can't run on the Pi, you can't use it. You also are **not** allowed to specify an entire file path to your sound file. Think about it like this, is your professor likely to have that exact file path on their own Raspberry Pi when they grade your submission? Is any other person wanting to run your code likely to have that specific file saved in that specific place on their hard drive? Definitely not. So if you want to play a sound file, that file **must** be in the same place as your code. This way you can reference just the file name and not have to include a path. If you want to put all sound files into a folder, make sure that folder is in the same place as your code and use a **forward slash** "/" when referencing the file. For example, in your code use "sounds/sound.wav" to refer to the "sound.wav" file in the "sounds" folder. Do make sure to include all sound files in your submission!

Make sure to put an appropriate header at the top of your program and to appropriately comment your source code as necessary. Since your game will now include images for each room, submit your source code, images, and any other required file(s) (e.g., sound files) **as a single ZIP archive**. If you include sound files, please make sure that they are compressed and/or optimized. **There is a 10MB limit on the size of the ZIP archive. Submissions that are greater than this size will not be accepted!**