Raspberry Pi Activity 3: Room Adventure

In this activity, you will implement a simple text-based game utilizing the object-oriented paradigm. You will need the following items:

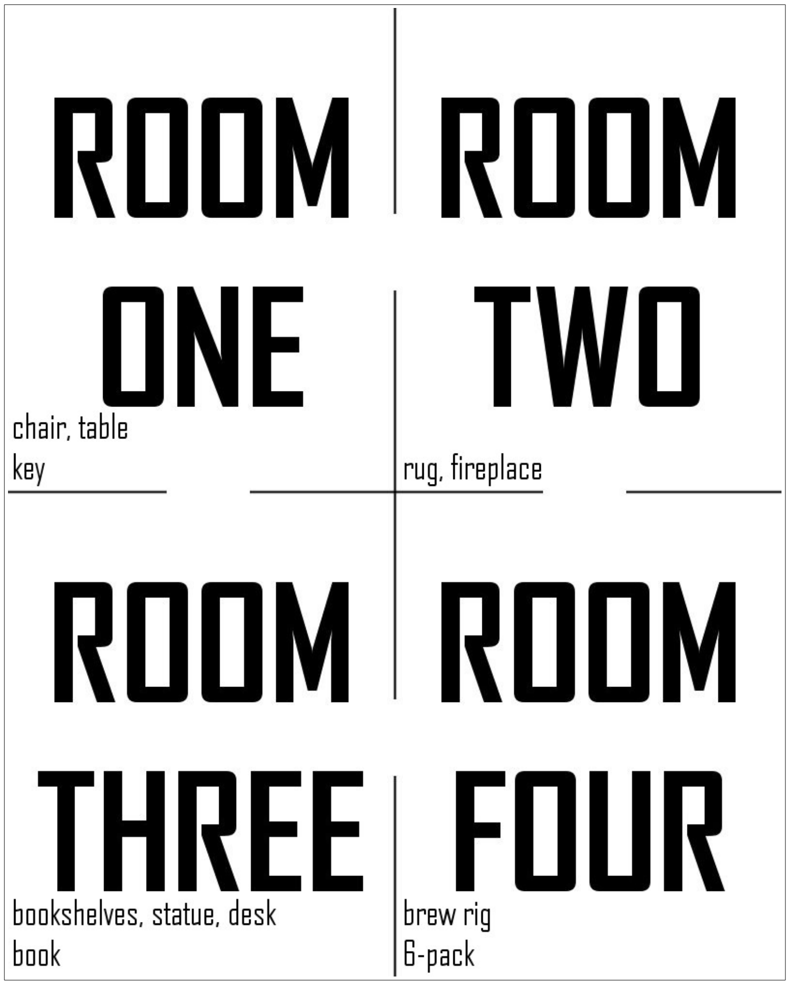
- Raspberry Pi B v2 with power adapter;
- LCD touchscreen with power adapter and HDMI cable; and
- Wireless keyboard and mouse with USB dongle.

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 first step to designing any kind of computer program is to establish the goal or purpose and to identify all of the necessary details. This is very similar to first understanding a problem before setting about to solve it. Although we will build a very simple game, it can still have (very simple) goals.

The setting of the game is a small "mansion" consisting of four rooms. Here's the layout:



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 (some ideas will be discussed later).

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.

The gameplay
The game is be text-based (egads, there are no graphics!). Situational awareness is provided by means of meaningful text that describes the current situation in the mansion. 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.

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 (young man!)

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

The design

By now, you may have already noticed that the four rooms are similar. They each have exits, items that may be observed, and items that may be added to the player's inventory. A good design choice is to utilize the object-oriented paradigm and implement a class that serves as the blueprint for all of the rooms in the game. In fact, let's begin there. Launch IDLE or a text editor and edit a new Python program. Call the document **RoomAdventure.py**. Let's begin with an informative header and the beginning of the room class:

```
######################################################################
# Name:
# Date:
# Description:
######################################################################

######################################################################
# the blueprint for a room
class Room(object):
     # the constructor
     def __init__(self, name):
          # rooms have a name, 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.exits = []
          self.exitLocations = []
          self.items = []
          self.itemDescriptions = []
          self.grabbables = []
```

Note that it is wise to periodically save your program in case something happens. Also, it may be best to simply read the source code in this activity so that you understand what is going on instead of actually writing the code as it appears in the document. The entire code will be shown later so that you can write it in its entirety and see the full context.

So far, we have covered the constructor of the room class. It does what most constructors do: initializes various instance variables. In this case, a room object must be provided a name in order to be successfully instantiated (note the `name` parameter in the constructor). The room's name is assigned, and lists that contain its exits, observable items, and items that can be placed in inventory are initialized (all as empty lists). The variables are defined as follows:

- *name*: contains the room's name;
- *exits*: contains the room's exits (e.g., north, south);
- *exitLocations*: contains the rooms found at each exit (e.g., to the south of this room is room 2);
- *items*: contains the observable items in the room;
- *itemDescriptions*: contains the descriptions of the observable items in the room; and
- *grabbables*: contains the items in the room that can be placed in inventory.

Note that the lists *exits* and *exitLocations* are known as parallel lists. So are the lists *items* and *itemDescriptions*. **Parallel lists** are two or more lists that are associated with each other. That is, they have related data that must be combined to provide meaningful information. The lists are correlated by position (or index). That is, the items at the first index of *exits* and *exitLocations* form a meaningful pair. If this were the object reference for room 1, for example, the first item in the list *exits* could be the string "east". Consequently, the first item in the list *exitLocations* would then be the object reference for room 2, since moving east from room 1 should lead to room 2. Similarly, the first item in the list *items* for room 1 could be "table". Consequently, the first item in the list *itemDescriptions* would then be "It is made of oak. A golden key rests on it." (since this is the description for the table in room 1).

The room class should then provide appropriate getters and setters for each instance variable. Recall that this is accomplished by providing getter and setter methods that manipulate (either access or change) instance variables, each of which typically begins with an underscore. For example, the getter for the list *exits* would be a function named `exits` that would return the instance variable (a list) *_exits*. Here are the getters and setters for each instance variable. These are located in the room class, beneath the constructor (make sure to properly indent!):

```
    # 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 exits(self):
         return self._exits

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

    @property
    def exitLocations(self):
         return self._exitLocations

    @exitLocations.setter
    def exitLocations(self, value):
         self._exitLocations = value

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

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

    @property
    def itemDescriptions(self):
         return self._itemDescriptions

    @itemDescriptions.setter
    def itemDescriptions(self, value):
         self._itemDescriptions = value

    @property
    def grabbables(self):
         return self._grabbables

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

The approach in the game design will be to create an instance of the room class for each room in the mansion. With what has been implemented so far, the room blueprint exists, but there is currently no way to add exits, items, or grabbables. We must therefore provide methods to enable adding each of these things to a room. The first obvious addition is to provide support for adding exits (and their appropriate exit locations). Let's call this method `addExit`, and have it be provided with an exit and associated room as parameters:

```
    # 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 lists
         self._exits.append(exit)
         self._exitLocations.append(room)
```

The next obvious addition is to provide support for adding observable items and their associated descriptions. Let's do this similarly to the previous method (i.e., it will be provided with an item and its description) and call the method `addItem`:

```
    # 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 lists
         self._items.append(item)
         self._itemDescriptions.append(desc)
```

Another obvious addition is to provide support for adding grabbables to the room in a method called `addGrabbable`. Since grabbables have no associated information, a single list is maintained. In addition, the method will be provided the grabbable item's name:

```
    # 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)
```

Since a grabbable item can be *grabbed* and placed in inventory, a method that removes the item from the room once it has been added to the player's inventory must be implemented. We will call this method `delGrabbable` and provided it with the item's name:

```
    # 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)
```

Lastly, it will be quite useful to provide a meaningful description of the room. This will make it simple to display all of the relevant information about the current room (e.g., exits, observable items, grabbables, etc). Recall that classes may specify a `__str__()` function that returns a string representation of a class. Statements that *print* an instance of the class are then directed to this function for the appropriate string to display. Add the following function to the room class:

```
     # 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:
               s += item + " "
          s += "\n"

          # next, the exits from the room
          s += "Exits: "
          for exit in self.exits:
               s += exit + " "

          return s
```

Note that the escaped character, `\n`, adds a linefeed to the string (which means to go to the next line). Strings can span multiple lines! The function *builds* a string (the variable *s* in the function above). If the player is currently in room 1 at the start of the game, for example, the string would be formatted as follows:

```
     You are in Room 1.
     You see: chair table
     Exits: east south
     You are carrying: []
```

At this point, we are finished with the room class. For clarity, here it is in its entirety:

```
######################################################################
# the blueprint for a room
class Room(object):
     # the constructor
     def __init__(self, name):
          # rooms have a name, 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.exits = []
          self.exitLocations = []
          self.items = []
          self.itemDescriptions = []
          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 exits(self):
        return self._exits

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

    @property
    def exitLocations(self):
        return self._exitLocations

    @exitLocations.setter
    def exitLocations(self, value):
        self._exitLocations = value

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

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

    @property
    def itemDescriptions(self):
        return self._itemDescriptions

    @itemDescriptions.setter
    def itemDescriptions(self, value):
        self._itemDescriptions = 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 lists
         self._exits.append(exit)
         self._exitLocations.append(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 exit to the appropriate lists
         self._items.append(item)
         self._itemDescriptions.append(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:
              s += item + " "
         s += "\n"

         # next, the exits from the room
         s += "Exits: "
         for exit in self.exits:
              s += exit + " "

         return s
```

The main part of the game
It is now time to implement the main part of the game. Note that the room class discussed above only specifies what rooms are (i.e., their state and behavior). No rooms have been created thus far, no user input capability has been implemented, no decision-making based on user input has been implemented, and so on. This is our next task.

Let's begin by initializing an empty inventory list and creating the rooms of the mansion. The source code that follows belongs beneath the room class:

```
######################################################################
# START THE GAME!!!
inventory = [] # nothing in inventory...yet
createRooms()  # create the rooms
```

Note the call to the function `createRooms()`. We often implement the main part of a program as a driver, in that it drives actions to be taken. Often, the actions are encapsulated in functions. Therefore, the main part of a program may call many functions. Overall, this helps to improve the readability of our programs and helps to simplify maintaining and updating them as well. The `createRooms()` function creates each instance of the four rooms. It sets their name, exits (and exit locations), observable items (and descriptions), and grabbable items. Lastly, it sets the player's current room at the beginning of the game (room 1). Let's work on the `createRooms()` function:

```
######################################################################
# creates the rooms
def createRooms():
     # 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)
     # since it needs to be changed in the main part of the program,
     # it must be global
     global currentRoom

     # create the rooms and give them meaningful names
     r1 = Room("Room 1")
     r2 = Room("Room 2")
     r3 = Room("Room 3")
     r4 = Room("Room 4")

     # 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
     currentRoom = r1
```

Note the exit to the south of room 4: None. This is the "window" referred to earlier. If the player exits south in room 4, the game will be over. This will be easy to check since the current room will be None (and not some actual instance of the class room). Recall that the reserved word **None** in Python refers to nothing or the absence of a value.

When the program is first run, the player's inventory list is initially empty. Subsequently, the call to the `createRooms()` function creates the rooms and stores them in memory. In addition, the global variable *currentRoom* is set to the local variable *r1* (room 1). Even though *r1* is local to the function `createRooms()`, it will still be in memory for the duration of the game.

At this point, all that's left to do is to display the information associated with the current room, to prompt the player for an input action, and to act based on the player's input action. This will be done beneath the call to the function `createRooms()`. First, let's provide situational awareness and also deal with the possibility that the player has *jumped out the window* (in which case the game should end). These tasks will occur until either the player dies or asks to leave the game. Therefore, the remainder of the program will be contained with a repetition construct (we'll use a while loop). Exiting the while loop will be possible by use of the `break` instruction:

```
# play forever (well, at least until the player dies or asks to quit)
while (True):
     # set the status so the player has situational awareness
     # the status has room and inventory information
     status = "{}\nYou are carrying: {}\n".format(currentRoom,
inventory)

     # if the current room is None, then the player is dead
     # this only happens if the player goes south when in room 4
     # exit the game
     if (currentRoom == None):
          death()
          break

     # display the status
     print "========================================================="
     print status
```

Initially, a default status is set that provides the information associated with the current room and the player's inventory. This is then displayed. However, if the current room is None (i.e., the player exited south in room 4), then the player is dead and the game is ended (via the break statement that will exit the while loop). Note the call to the function `death()`. For now, you can comment this one out. It simply displays an appropriate “message” when the player dies. It will be provided later in the full code listing for the main part of the game.

The next task is to add support for user input:

```
     # prompt for player input
     # the game supports a simple language of <verb> <noun>
     # valid verbs are go, look, and take
     # valid nouns depend on the verb
     # we use raw_input here to treat the input as a string instead of
     # an expression
     action = raw_input("What to do? ")

     # set the user's input to lowercase to make it easier to compare
     # the verb and noun to known values
     action = action.lower()

     # exit the game if the player wants to leave (supports quit,
     # exit, and bye)
     if (action == "quit" or action == "exit" or action == "bye"):
          break
```

The `raw_input` function is used instead of the familiar `input` function since it interprets the user's input as a string by default (unlike the `input` function). The strategy is to prompt the player for input and convert it to lowercase to make comparison to the supported vocabulary easier. We also handle exiting the game. The next task is to parse the player's action; that is, to determine the input and try to apply it to the supported vocabulary of a verb followed by a noun:

```
     # set a default response
     response = "I don't understand.  Try verb noun.  Valid verbs are
go, look, and take"
     # 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]
```

At this point, the user's input is parsed into a verb and a noun. The next step is to check the verb to see if it matches one that is supported (i.e., *go*, *look*, *take*):

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

               # check for valid exits in the current room
               for i in range(len(currentRoom.exits)):
                    # a valid exit is found
                    if (noun == currentRoom.exits[i]):
                         # change the current room to the one that is
                         # associated with the specified exit
                         currentRoom = currentRoom.exitLocations[i]

                         # set the response (success)
                         response = "Room changed."

                         # no need to check any more exits
                         break
```

If the verb is *go*, we then check the noun for a valid exit in the current room. Recall that the exits have an associated exit location (via the parallel lists declared in the room class). The strategy is to identify the specified exit in the list of exits in the current room, and then identify the matching exit location (i.e., the room that the exit leads to). Subsequently, we change the current room to this adjacent room. We then break out of the for loop which will display the response (shown later) and cycle back to the beginning of the while loop.

Support for the verb *look* is similar:

```
          # 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
               for i in range(len(currentRoom.items)):
                    # a valid item is found
                    if (noun == currentRoom.items[i]):
                         # set the response to the item's description
                         response = currentRoom.itemDescriptions[i]

                         # no need to check any more items
                         break
```

Support for the verb *take* is only slightly different:

```
          # 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 currentRoom.grabbables:
                    # a valid grabbable item is found
                    if (noun == grabbable):
                         # add the grabbable item to the player's
                         # inventory
                         inventory.append(grabbable)

                         # remove the grabbable item from the room
                         currentRoom.delGrabbable(grabbable)

                         # set the response (success)
                         response = "Item grabbed."

                         # no need to check any more grabbable items
                         break
```

In this case, if the specified grabbable item is found in the room, it is added to the player's inventory. Subsequently, it is removed from the current room's list of grabbables (after all, we don't want to grab it again!). An appropriate response is set, and the for loop is exited.

The last thing to do is to display the response and cycle back to the beginning of the while loop so that the description of the current room and the player's inventory can be displayed, and input can be solicited from the player once again:

```
     # display the response
     print "\n{}".format(response)
```

Yes, this is a lot of code split up into many parts across this document. This is why it was suggested that you not actually write any segmented code as you read this document and process through the activity. And now, here is the entire source code for the main part of the program:

```
######################################################################
# START THE GAME!!!
inventory = [] # nothing in inventory...yet
createRooms()  # add the rooms to the game

# play forever (well, at least until the player dies or asks to quit)
while (True):
     # set the status so the player has situational awareness
     # the status has room and inventory information
     status = "{}\nYou are carrying: {}\n".format(currentRoom, 
inventory)

     # if the current room is None, then the player is dead
     # this only happens if the player goes south when in room 4
     if (currentRoom == None):
          status = "You are dead."

     # display the status
     print "========================================================="
     print status

     # if the current room is None (and the player is dead), exit the
     # game
     if (currentRoom == None):
          death()
          break

     # prompt for player input
     # the game supports a simple language of <verb> <noun>
     # valid verbs are go, look, and take
     # valid nouns depend on the verb
     # we use raw_input here to treat the input as a string instead of
     # a numeric value
     action = raw_input("What to do? ")

     # set the user's input to lowercase to make it easier to compare
     # the verb and noun to known values
     action = action.lower()

     # exit the game if the player wants to leave (supports quit,
     # exit, and bye)
     if (action == "quit" or action == "exit" or action == "bye"):
          break
     # set a default response
     response = "I don't understand.  Try verb noun.  Valid verbs are
go, look, and take"
     # split the user input into words (words are separated by spaces)
     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
               for i in range(len(currentRoom.exits)):
                    # a valid exit is found
                    if (noun == currentRoom.exits[i]):
                         # change the current room to the one that is
                         # associated with the specified exit
                         currentRoom = currentRoom.exitLocations[i]

                         # set the response (success)
                         response = "Room changed."

                         # no need to check any more exits
                         break
          # 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
               for i in range(len(currentRoom.items)):
                    # a valid item is found
                    if (noun == currentRoom.items[i]):
                         # set the response to the item's description
                         response = currentRoom.itemDescriptions[i]

                         # no need to check any more items
                         break
          # 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 currentRoom.grabbables:
                    # a valid grabbable item is found
                    if (noun == grabbable):
                         # add the grabbable item to the player's
                         # inventory
                         inventory.append(grabbable)

                         # remove the grabbable item from the room
                         currentRoom.delGrabbable(grabbable)

                         # set the response (success)
                         response = "Item grabbed."

                         # no need to check any more grabbable items
                         break

     # display the response
     print "\n{}".format(response)
```

Source code template?
You may be asking yourself if there is source code in the form of a template for this activity. No, there isn't. Although it will be tedious and a bit time-consuming, typing the code by hand will help you to understand the code, to learn how to properly format Python code, and perhaps result in better retention of Python syntax and good program style and structure. Please refrain from simply copy-and-pasting the source code from the PDF version of this document into a Python source code file. It is to your advantage to take time now to learn this so that it becomes easier in the future. After all, aren't you in this curriculum because you are interested in learning this stuff and because you chose to be here? In fact, the prof should code the game with you in real time during this activity, typing each statement, one at a time!

The flow of the source code is as follows:

- The room class;
- The `createRooms()` function;
- The (optional) `death()` function; and
- The main part of the program.

The optional `death()` function

Earlier, you were asked to comment out the call to the function `death()` in the main part of the program. In case you wish to implement it, here it is (yes, it is intentionally obfuscated). It must be defined above the main part of the program. Since it is obfuscated, you may copy-and-paste this function into your source code (pay attention to the end of the lines and indentation!):

```
# displays an appropriate "message" when the player dies
# yes, this is intentionally obfuscated!
def death():
     print " " * 17 + "u" * 7
     print " " * 13 + "u" * 2 + "$" * 11 + "u" * 2
     print " " * 10 + "u" * 2 + "$" * 17 + "u" * 2
     print " " * 9 + "u" + "$" * 21 + "u"
     print " " * 8 + "u" + "$" * 23 + "u"
     print " " * 7 + "u" + "$" * 25 + "u"
     print " " * 7 + "u" + "$" * 25 + "u"
     print " " * 7 + "u" + "$" * 6 + "\"" + " " * 3 + "\"" + "$" * 3 +
"\"" + " " * 3 + "\"" + "$" * 6 + "u"
     print " " * 7 + "\"" + "$" * 4 + "\"" + " " * 6 + "u$u" + " " * 7
+ "$" * 4 + "\""
     print " " * 8 + "$" * 3 + "u" + " " * 7 + "u$u" + " " * 7 + "u" +
"$" * 3
     print " " * 8 + "$" * 3 + "u" + " " * 6 + "u" + "$" * 3 + "u" + "
" * 6 + "u" + "$" * 3
     print " " * 9 + "\"" + "$" * 4 + "u" * 2 + "$" * 3 + " " * 3 +
"$" * 3 + "u" * 2 + "$" * 4 + "\""
     print " " * 10 + "\"" + "$" * 7 + "\"" + " " * 3 + "\"" + "$" * 7
+ "\""
     print " " * 12 + "u" + "$" * 7 + "u" + "$" * 7 + "u"
     print " " * 13 + "u$\"$\"$\"$\"$\"$\"$u"
     print " " * 2 + "u" * 3 + " " * 8 + "$" * 2 + "u$ $ $ $u" + "$"
* 2 + " " * 7 + "u" * 3
     print " u" + "$" * 4 + " " * 8 + "$" * 5 + "u$u$u" + "$" * 3 + "
" * 7 + "u" + "$" * 4
     print " " * 2 + "$" * 5 + "u" * 2 + " " * 6 + "\"" + "$" * 9 +
"\"" + " " * 5 + "u" * 2 + "$" * 6
     print "u" + "$" * 11 + "u" * 2 + " " * 4 + "\"" * 5 + " " * 4 +
"u" * 4 + "$" * 10
     print "$" * 4 + "\"" * 3 + "$" * 10 + "u" * 3 + " " * 3 + "u" * 2
+ "$" * 9 + "\"" * 3 + "$" * 3 + "\""
     print " " + "\"" * 3 + " " * 6 + "\"" * 2 + "$" * 11 + "u" * 2 +
" " + "\"" * 2 + "$" + "\"" * 3
     print " " * 11 + "u" * 4 + " \"\"" + "$" * 10 + "u" * 3
     print " " * 2 + "u" + "$" * 3 + "u" * 3 + "$" * 9 + "u" * 2 +
" \"\"" + "$" * 11 + "u" * 3 + "$" * 3
     print " " * 2 + "$" * 10 + "\"" * 4 + " " * 11 + "\"\"" + "$" *
11 + "\""
```

```
     print " " * 3 + "\"" + "$" * 5 + "\"" + " " * 22 + "\"\"" + "$" *
4 + "\"\""
     print " " * 5 + "$" * 3 + "\"" + " " * 25 + "$" * 4 + "\""
```

Suggested improvements

Although this is a simple game in terms of gameplay, it can quickly become quite complicated to design and implement. It can also become more involved and, dare I say it, even more fun than it already is by making just a few fairly minor improvements:

- Some items that can be placed in the player's inventory are identified in the description of items (e.g., the key on the table in room 1). When placed in inventory, the item's description should be appropriately modified (e.g., the key is no longer on the table).
- Adding more observable items and items that can be added to inventory in each room.
- Adding more rooms (e.g., making some sort of maze).
- Making the mansion three-dimensional (i.e., adding exits above and below to new rooms).
- Adding a goal that requires solving a puzzle. For example, using a key in inventory that was taken from a room to unlock a box that is located in another room. Unlocking the box may reveal new items. This would require adding to the vocabulary (e.g., a new verb *use*). If the player is in the correct room and has the correct item in inventory, issuing the proper action (e.g., "use key") would solve a puzzle and/or reveal new items that can be observed or taken into inventory.
- Adding the ability to look at individual inventory items (i.e., inventory items can have descriptions as well).
- Adding points that the player can accumulate by adding items to inventory or by solving puzzles.
- ...there are more...

Homework: Room Adventure

For the homework portion of this activity, you will work in **groups of three**. I suggest that groups contain at least one confident Python coder.

Your task is to implement **at least one** of the suggested improvements. **Clearly comment your changes and additions to the source code!** If you wish to implement an improvement that is not listed above, please get with the prof first for approval.

You are to submit your Python source code only (as a `.py` file) through the upload facility on the web site.