4
Internet Radio Projects 2b, 2c and 3

Project 2b – parsing the playlist file for the Internet radio

Now that we have the playlist file defined and have added some entries, we need to be able to read the file and build the internal data structures that will be used to hold station data in the Internet radio.

Running parse.py

A fundamental component of the Internet radio is its ability to read the playlist file. Let's look at the code involved in that function:

1. Download the `parse.py` program and place it in the `/home/pi/radio/bin` folder.
2. Start IDLE3 to run the program:
    1. Open **IDLE3**. If only the shell window opens, select **File | New Window** to open an Editor window.
    2. In the Editor window, open the `parse.py` program.

The code in the `parse.py` file should help you to understand how the rest of the application is defined. Since you have built the playlist file already, let's start the application using **Run | Run Module** in the Editor window. The application will read the playlist file, print the results and build the internal storage array for the stations. The following is the code to accomplish this:

```
#!/usr/bin/python3
#global variables
strPlaylist="/home/pi/radio/playlist"
stationarray=[]
stationcount=0 2

def ParsePlaylist(filename):
     stationlist=[]
     printlist=[]
     global stationcount
     try:
         with open(filename, "r") as input:
               for line in input:
                    if ("," in line ):
                        line=line.replace(" ","")
                        line= line.split(",")
                        stationlist=stationlist + line
                        printlist=line
                        stationcount += 1
                        print(stationcount, printlist[0])
                    input.close()
     except IOError:
           print ("Unable to open ", filename)
           return # exit()
     return (stationlist)
stationarray=ParsePlaylist(strPlaylist)
print("\n", stationarray)
```

It is important to note that this code has four separate definition areas:

1. The global variables are defined.
2. A function is defined that holds all the code to open, read, and process the playlist file.
3. There is a standalone Python statement that calls the function and uses the return data to build the station data in a Python list called `stationarray`.
4. Lastly, there is a print statement that shows the Python list for the stations.

While we are not trying to teach Python programming, it's worth pointing out the following explanations, which may help you to better understand the blocks of code:

- The `def` block of code for the `ParsePlaylist` function has local variables visible only within the function. If you define a variable outside any function code, it can be made globally available within a function using the global keyword. If you define a local variable within a function and it has the same name as a variable outside that definition, you can't get to see the global one. It's good practice to always think of the variables outside of functions or classes as global and use the global keyword to highlight the scope needed within the functions.
- There is a `try: / except:` block in the code that is used to capture errors that might occur. In this case, we will want to capture an error if the playlist file does not exist, so it uses the `except IOError:` statement to capture such exceptions.

Project 2c – designing a Python text interface for VLC

This project is a simple text interface with the following features:

- Reads the Internet station streams we want from a file called `/home/radio/playlist`
- Shows the first item in the playlist as the default, and it will play as soon as the application loads
- Prints out a list of the stations in the playlist file
- Allows you to enter a station number to start playing any station
- Implements a simple retry mechanism if stations don't start to play within 2 seconds
- Allows you to enter `0` to exit the program

Running radio.py

Before we can begin this next exercise, you will need to download the `radio.py` program and place it in the `/home/pi/radio/bin` folder.

This download is the complete implementation of the radio using Python 3 with a text-mode interface. The VLC application should already be running if you followed the earlier tasks. You can check with **Task Manager** and also log in to it via Telnet to check whether VLC is functioning.

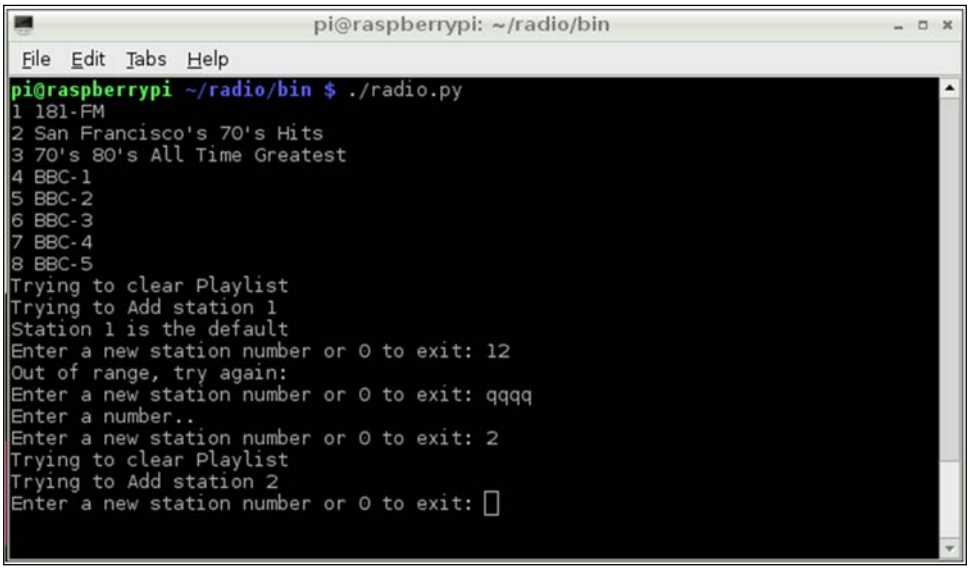
When you download the `radio.py` code, you will notice that it is laid out as a group of defined functions that implement the radio logic as described here:

1. `def ParsePlaylist (filename)`: This opens the playlist file, prints a count number and station description, and finally returns the information to create the `stationarray` list.
2. `def CreateSession( host, port, timeout)`: This connects to the VLC Telnet server and uses a `try:/except:` block to capture timeouts or connection failures. One thing to note here is that the Telnet protocol uses a strict ASCII character stream, so the `.encode` function converts the internal default UTF-8 format to ASCII.

Read about ASCII codes at:
`http://en.wikipedia.org/wiki/ASCII`

3. `def AddStationtoPlaylist( stationnum )`: This uses the `clear` command to empty the playlist and then an `add` command with the station URL to write the station URL to the VLC playlist.
4. `def ClearPlaylist()`: The reason for this function may not be immediately apparent, but it prevents the number of entries in the VLC playlist from accumulating. It uses the `clear` command to reset the VLC playlist. You might notice, however, that the numerical identifiers for the VLC playlist just keep incrementing as you clear, and add to, the playlist.
5. `def PlayRadio()`: This provides the initialization logic and a `while` loop to interface with the user.

The final line of Python code, `PlayRadio()`, simply calls the function to start the operation of the radio. The text-based user interface looks like this:



Notice in the preceding image that the output is rather verbose and you only get the listing of stations once. This interface would be awkward to use, but we'll fix it shortly by adding a nice GUI. Also notice that if you exit the Python program, the radio station last selected continues to play asVLC is still running.

The logic for automating VLC operation

The logic for implementing the Internet radio via a Telnet connection is very simple at a high level. The following is the programmatic logic for implementing the radio:

```
def PlayRadio():
      global stationarray
      global stationcount
      global SessionFlag
      global strPlaylist
      global userloop
      #Parse the playlist file
      tmp = ParsePlaylist(strPlaylist)
      if tmp == False: #return is either the playlist or fail flag
          print("Playlist error .....exiting")
          exit()
      else:
```

```
              stationarray=tmp
        if (stationcount != 0):
            SessionFlag = CreateSession(HOST, PORT, 2)
            if not SessionFlag:
                 print("Unable to connect to VLC...exiting")
                 exit()
            #If we get here we have a playlist and can talk to VLC
            AddStationtoPlaylist( 1, False )
            print("Station 1 is the default")
        else:
            print("No stations to play ....exiting")
            exit()
        #Once the radio is running we just sit in a tight loop
        while userloop: #you can set userloop=False to debug in IDLE
            try:
                 x=int(input("Enter a new station number or 0 to exit:
"))
            except ValueError:
                      print("Enter a number.. ")
             else:
                      if x == 0:
                          break
                      if 1 <= x <= (stationcount):
                          AddStationtoPlaylist(x, False)
                      else:
                          print("Out of range, try again: ")
             exit()
```

The logic is as follows:

1. Parse the playlist if it exists and exit on errors.
2. Add the first station in the playlist on startup.
3. Sit in a loop waiting for user input.
4. Add the new user station selection or exit on zero.

There are, however, some error conditions that can occur annoyingly often. Even in the desktop VLC UI, it's possible to click on a URL, and the station simply will not play. There are no errors shown in the UI, and you have to click again to start the channel playing.

With an automated VLC over Telnet, the same problem may occur, where a station URL is either not resolved or a connection to the remote server stalls for some reason. To try to provide a robust retry mechanism, we looked in depth at the VLC command set via Telnet. Here's what we found:

- Sending commands to VLC is done half duplex. You build a command string programmatically in a Python string (or in `netcat`), and it is sent by a single function from the Telnet library. VLC can take from several milliseconds up to several hundred milliseconds to respond to a command request.
- The response to a command from VLC in the most primitive case is simply a cursor string "`>`". For those commands that provide return information, the information is sent followed by a cursor string.
- In most cases, VLC provides correct information, but if there is some form of error, VLC can provide the wrong response or even change its mind after it has sent a response. For example, our logic uses the `is_playing` flag. The operation of this flag would seem to be quite clear, if set to one, the station is playing audio; if set to zero, it's not playing.

In the logic used in `radio.py`, we depended on the `is_playing` flag to know whether the station we had added to the playlist was resolved, whether a connection was made to the source, and indeed whether audio was being played. However, it did not work in exactly the way we expected.

- When you send a `clear` or `add` command to VLC if it is not currently playing a station, the flag is zero. Life is good.
- When you send a clear or add command to VLC and it is currently playing a station, it can take several tens of milliseconds for the `is_playing` flag to drop to zero.
- If you send an add command to VLC and the `is_playing` flag is zero, the flag sets within a few milliseconds. (Is life good? Well, maybe not.) However, if the station connection has an error and you get no audio stream, VLC will then drop the flag to zero. This can take 500 – 1000 ms to happen. If you read the flag state earlier, however, you would not know that it has since been set to zero.

The code for `AddStationtoPlaylist` is shown as follows:

```
def AddStationtoPlaylist( stationnum, retryflag ):
    """Clear playlist and add new URL entry, if retryflag is True then
it's a retry"""
    loopcount=15        #loop for maximum 15 * 0.15 seconds = 2.25
seconds max
    global outbuf
    global inbuf
```

```
        global SessionFlag
        global tn
        if (SessionFlag == True):
            if retryflag: print("Retrying....")
            ClearPlaylist()
            if debug : print("Trying to Add station", stationnum)
            TelnetTx("add " + stationarray[ ( (stationnum * 2) - 1) ] +
"\n")
            TelnetRx("> ")
            TelnetTx("is_playing\n")
            TelnetRx("> ")
            #we need to check that the vlc is_playing flag drops to zero
            while inbuf[0] == 48 and loopcount:
                loopcount -= 1
                if debug : print("Loopcount = ", loopcount)
                if debug : print("The is_playing flag is still 0")
                if debug : print("Waiting for is_playing to set")
                TelnetTx("is_playing\n")
                if debug : print("Test is_playing flag")
                TelnetRx("> ")          #loop to test flag again without
any delay
                #when is_playing is set to one we know that vlc is
responding
                #to the new station URL
            if debug : print("The is_playing flag is now 1")
            TelnetTx("is_playing\n")
            TelnetRx("> ")
            while inbuf[0] == 49 and loopcount: #49 = ascii 1
                loopcount -= 1

                if debug : print("Wait to see if is_playing is stable,
Loopcount = ", loopcount)
                sleep (0.15)
                TelnetTx("is_playing\n")
                TelnetRx("> ")
                if inbuf[0] == 48: AddStationtoPlaylist( stationnum, True
)
                if inbuf[0] == 49: continue
            return(True)
        else:
            print("Session is closed")
            return(False)
```

In the preceding code, there are two loops with a loop count of 15 that check that the `is_playing` flag drops to zero after a clear command, and that the `is_playing` flag remains set for at least a count of 15 when the URL is added.

For the `clear` command loop, there is no delay in the loop, but program execution does not proceed until the flag is zero. If you turn on the `debug` flag in the code, you may notice that it typically takes from 2 – 7 loops for the `is_playing` flag to drop to zero if it was set.

For the `add URL` command loop, there is a delay in the loop of 0.15 seconds and the `is_playing` flag must be set for the complete loop count. This means the flag must remain set for at least 2.25 seconds after adding the URL. If the flag does not remain set for this time period after the add (`retry`) command is sent, we request a retry. For the retry, we do not clear the playlist first since we know there is only one entry in it, and this saves a little time.

Much more could be done in the logic to ensure that all potential errors are caught, but the code, as implemented, is fairly robust. Improvements such as verifying that the title is set correctly (which proves the resolution of the station URL) and verifying that the statistics show that the audio is being decoded would be great additions.

Project 3 – implementing a TKinter GUI for the Internet radio

With the text-mode interface (`radio.py`) complete and a basic GUI (`tktest.py`) tested, we can move to the final integration tasks of the project. There is very little new code in `tkradio.py`; it's simply a merging of previous work.

Window-based interfaces are typically user-event driven and this significantly reduces the amount of programming required to support a GUI. In the case of a TKinter/Python application, the task is split into roughly the following components:

- Initialization code
- Telnet communications functions
- Radio logic functions
- A base window to hold all the GUI components
- Labels to provide UI indicator functions
- Buttons to capture user action directives
- A Spinbox that holds the station list

The code for the final implementation is about 200 lines long, but it is split into relatively easy-to-consume, small, functional blocks. We won't discuss all the code, but it is worth covering some of the major elements.

The TKinter window reference and properties are set using the following code:

```
#Define the TKinter window
root=Tk() #Creates the window reference
root.wm_title("Internet Radio") # Window title
root.config(background = "#FFFFF0")

#TKinter variables, these can be bound to a control
vtime=StringVar(value="Time")
vstatus=StringVar(value=".....")
vspinnum=StringVar(value="")
```

Immediately following the window definition, we declare the special TKinter variables. Each of these variables is bound to a particular GUI control or label element, and if you alter the value, the control is updated to reflect the change in the background. To ensure that the changes made to these variables are immediately reflected in the controls, you can perform an update (`root.update()`) function if required. If you don't force an immediate update, TKinter will render the update in a lazy fashion based on internal timing.

The following is the `AddStationtoPlaylist()` code, which is responsible for clearing the playlist, adding a new station to play, and monitoring the VLC state using the `is_playing` flag:

```
def AddStationtoPlaylist( stationnum, retryflag ):
    """Clear playist and add new URL entry, if retryflag is True then
it's a retry"""
    loopcount=15
    global outbuf
    global inbuf
    global SessionFlag
    global tn
    global vstatus
    if (SessionFlag == True):

        if retryflag:
            print("Retrying....")
            #set the UI status field and force an update
            vstatus.set("Retrying....")
            root.update()
        ClearPlaylist()
        if debug :
                 print("Trying to Add station", stationnum)
        TelnetTx("add " + stationarray[ ( (stationnum * 2) - 1) ] +
"\n")
```

```
        TelnetRx("> ")
        TelnetTx("is_playing\n")
        TelnetRx("> ")
        #we need to check that the vlc is_playing flag drops to zero
        while inbuf[0] == 48 and loopcount:
            loopcount -= 1
            if debug : print("Loopcount = ", loopcount)
            if debug : print("The is_playing flag is still 0")
            if debug : print("Waiting for is_playing to set")
            TelnetTx("is_playing\n")
            if debug : print("Test is_playing flag")
            TelnetRx("> ")
            #loop to test flag again without any delay
           #when is_playing is set to one we know
          #that vlc is responding to the new station URL
        if debug : print("The is_playing flag is now 1")
        TelnetTx("is_playing\n")
        TelnetRx("> ")
        while inbuf[0] == 49 and loopcount: #49 = ascii 1
            loopcount -= 1
            if debug :
                 print("Wait to see if is_playing is stable,
                           Loopcount = ", loopcount)
            time.sleep (0.15)
            TelnetTx("is_playing\n")
            TelnetRx("> ")
            #If is_playing=48=zero then we will do a retry
            if inbuf[0] == 48:
                 AddStationtoPlaylist( stationnum, True )
            #if is_playing=49=one we assume all is ok
            if inbuf[0] == 49:
                 vstatus.set("..........")
                 continue
        return(True)
    else:
        print("Session is closed")
        return(False)
```

The preceding code handles both the initial `clear/add` function and the `retry` function. When performing a retry, the TKinter variable, `vstatus`, is changed to `Retrying….` to provide a user interface indicator. When the station is finally playing (as shown by the state of `is_playing`), the indicator is rewritten back to "……….". Note here that `root.update()` is called to ensure that the interface is immediately updated to provide a user indicator.

In the following code `forPlayradio()`, the code is much shorter than that of the `radio.py` program. The reason is that there is no user input loop to be maintained. Once the radio is playing a station, there is no need to loop while waiting for user input. All user input is captured in the TKinter window message system. The following code encapsulates the discussion in this paragraph:

```
def PlayRadio():
     global stationarray
     global stationcount
     global SessionFlag
     global strPlaylist
     global userloop
     if (stationcount != 0):
         SessionFlag = CreateSession(HOST, PORT, 2)
         if not SessionFlag:
             print("Unable to connect to VLC...exiting")
             exit()
         #If we get here we have a playlist and can talk to VLC
         AddStationtoPlaylist( 1, False )
         print("Station 1 is the default")
         return(True)
     else:
         print("No stations to play ....exiting")
         exit()
```

The code to start the Internet radio is quite straightforward:

```
stationarray = ParsePlaylist(strPlaylist)

PlayRadio()

#Now start the GUI elements

buildwindow()

update_clock()          #Kick off the clock

root.mainloop()         #start window event loop
```

We start by parsing the playlist, and if that is okay, start the Telnet session and set the default, station one, to play. The code to build the entire GUI is contained in the `buildwindow()` function. Once the window is rendered, the clock is started, and finally, we enter the windows' message loop with `root.mainloop()`. From this point on, the only code called is due to the actions in the user interface and is called from the TKinter message process.

One thing you might notice in the code for the Internet radio is the liberal use of debug print statements. These are a valuable way to check your functionality during development, but be aware that printing to the console within the IDLE environment is very slow. This impacts on any timing you may have implemented within your program, so make sure you do final checks, either redirecting the output to a file or by starting the program from a terminal session.

While we have not covered the use of the IDLE debug environment, you can set breakpoints and step through the Python code in IDLE. This feature is really helpful since it will open and step through libraries, and there is typically extensive documentation that can help your understanding.