Criterion C: Development

Web Form `HTML`

```
<!DOCTYPE html>
<head>
    <title>Form</title> <!—Form title; this shows up in the tab.—>
    <link rel="stylesheet" href="style.css"> <!—Link stylesheet—>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
</script> <!—Import Ajax—>
    <script src="form.js"></script> <!—Link JS File—>
</head>
<html>
<div id="container"> <!—a container div to center the form in the middle of
the page—>
<form id="form" target="_self" onsubmit="postToGoogle();" autocomplete="on">
<!—Create a form that, when the submit button is clicked, executes a JS
function to send the data to a Google Sheet

    <!—Labels for entries and their input boxes—>

    <label>Team Name</label> <br>
    <input name="entry.1084862479" id="name" /> <br>
<br>
    <label for="email">Email</label> <br>
    <input type="email" name="entry.2141364104" id="email" /> <br>
<br>
    <label for="summary">A summary of what you're performing.</label> <br>
    <textarea name="entry.973299103" id="summary" ></textarea> <br>
<br>
    <label for="phone">Phone #</label> <br>
    <input type="tel" name="entry.1493511292" id="phone" /> <br>

    <label for="category">Type</label> <br> <!—select 1 out of 4 of these
categories for performance—>
    <select id="category" name="entry.1296106793">
      <option value="Dance - Classical Dance">Dance-Classical Dance</option>
      <option value="Dance - Folk/Non-Film/Film Dance">Dance - Folk/Non-
Film/Film Dance</option>
      <option value="Vocal - Classical">Vocal - Classical</option>
      <option value="Vocal - Folk/Film/Non-Film Vocal">Vocal - Folk/Film/Non-
Film Vocal</option>
```

```
    </select>
    <label for="age">Age Group</label> <br> <!—select 1 out of 4 of these
categories for age—>
    <select id="age" name="entry.1807843353">
      <option value="0-7">Sub-junior - up to 7 years</option>
      <option value="8-12">Junior - Age 8-12 years</option>
      <option value="13-18">Teen - Age 13-18 years</option>
      <option value="19+">Adult - 19+ years</option>
    </select>
<br> <br>
<button type="submit" value="Submit" class="common_btn">Submit</button> <!—
the submit button that, when clicked, causes postToGoogle() to execute—>
</form>
</div>
</html>
```

I chose to simplify my web form to be on one page so that the users would have the easiest possible way of registering. I mainly used the framework from W3School's articles on how to make a form for my form, only adding a few font fixes and a shadow around the entire form. For more information on W3School's form, please take a look at the appendix. At first I decided that the 4-choice questions would be multiple choice, but I realized that the best way to save space was to have a selector rather than a multiple choice widget. In fact, I wonder why people use a multiple choice widget for more than two choices -- a selector widget just seems so much easier and takes much less space overall...

For my form, I had a little bit of trouble deciding which part of the form would trigger the function that would send the results to a Google Form. Obviously, it would be the submit button, but I spent quite a time trying to make the click event register on the form.

Web Form `CSS`

```
input, select, textarea {
  width: 100%; /* make the width of the inputs 100% of the container */
  padding: 10px; /* add a little padding */
  border: 1px solid #ccc; /* add a border to emphasize the buttons */
  border-radius: 0px; /* make it square */
  box-sizing: border-box; /* make padding + border included in dimensions */
  margin-top: 6px; /* add margins at the top and bottom */
  margin-bottom: 6px;
  resize: vertical; /* You can resize the boxes to your liking if you don't
like the size */
}
.common_btn { /* Make the submit button green, in the center, big, and
slightly rounded. */
```

```
  background-color: #4cbf99;
  border: 1px solid #4cbf99;
  border-radius: 4px;
  padding: 10px;
  color: white;
  display: flex;
  justify-content: center;
  margin-left: 45%;
}
#container { /* Make the entire shape of the form approximately square and
approximately in the middle of the page. Additionally, make sure fonts can be
displayed correctly. */
  width: 700px;
  margin: 0 auto;
  position: relative;
  top: 50%;
  -ms-transform: translateY(20%);
  transform: translateY(20%);
  box-shadow: 0px 0px 30px 2px rgba(0,0,0,0.8);
  padding: 3%;
  font-family: -apple-system, BlinkMacSystemFont, sans-serif;
}
```

There's not really much need to explain this -- just a few commands to keep the form centered, easily readable, and have a shadow border.

Web Form `JS`

```
function postToGoogle() {
  var field1 = $("#name").val();
  var field2 = $("#email").val();
  var field3 = $("#summary").val();
  var field4 = $("#phone").val();
  var field5 = $("#category option:selected").text();
  var field6 = $("#age option:selected").text();

  if(field1 == ""){
    alert('Please provide your name.');
    document.getElementById("name").focus();
    return false;
  }
  if(field2 == ""){
    alert('Please provide your email.');
    document.getElementById("email").focus();
      return false;
  }
```

```
  if(field3 == "" || field3.length < 2 ){
    alert('Please provide a more detailed summary.');
    document.getElementById("email").focus();
      return false;
  }
  if(field4 == "" || field4.length > 10 || field4.length < 10){
    alert('Please provide your phone number.');
    document.getElementById("phone").focus();
    return false;
  }
  if(field5 == ""){
    alert('Please provide your category.');
    document.getElementById("category").focus();
      return false;
  }
  if(field6 == ""){
    alert('Please provide your age.');
    document.getElementById("age").focus();
      return false;
  }

  $.ajax({
    url:
"https://docs.google.com/forms/d/e/1FAIpQLScGBFSBWTIKNoDzHfoJApQzCPbwHJcSLxBP
gvkAXbHugeqBBQ/formResponse?",
    data: {"entry.1084862479": field1, "entry.2141364104": field2,
"entry.1493511292": field3, "entry.973299103": field4, "entry.1296106793":
field5, "entry.1807843353": field6},
    type: "POST",
    dataType: "xml",
});
return false;
}
```

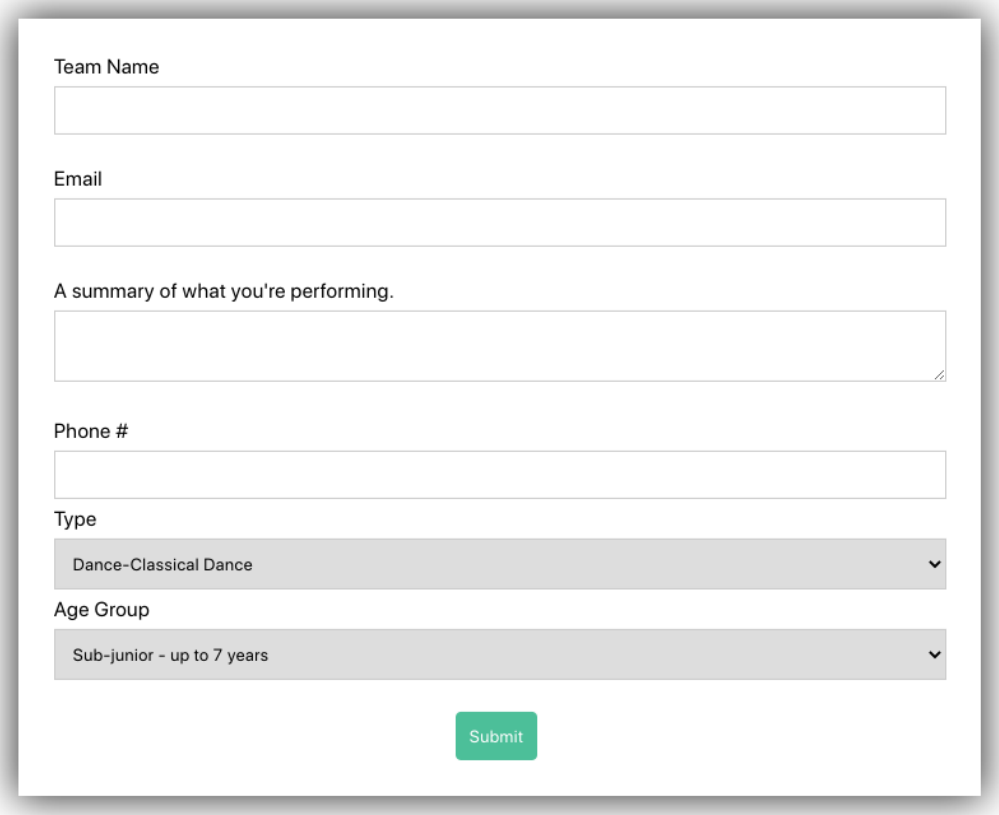


Figure 1: A screenshot of the web form

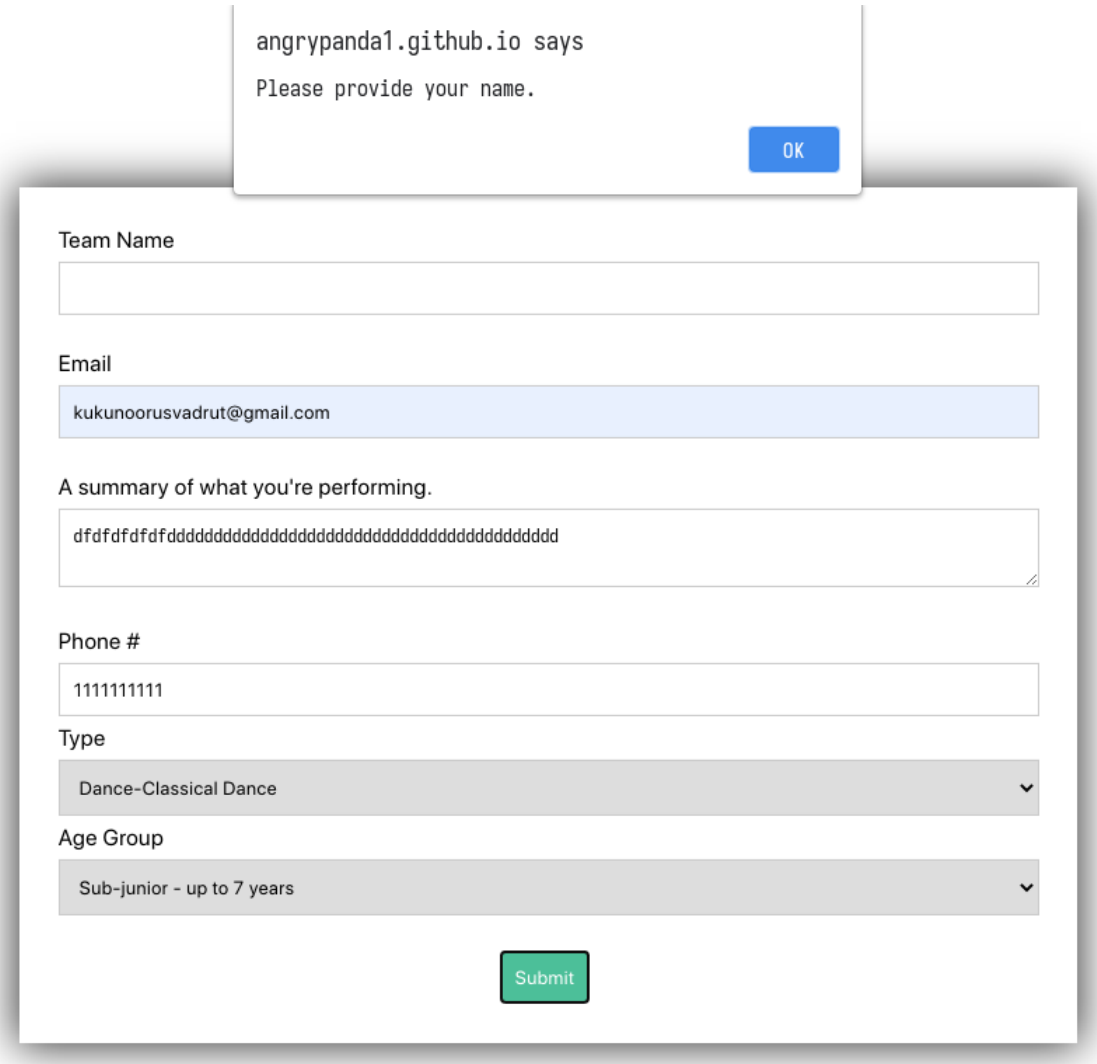
I chose to put these three pieces of code in separate documents because of my thoughts on procedural documents. If I had put all the three pieces of code into one document, it would have worked, but it would not have been easy to maintain should any bugs pop up. Additionally, it would have been hard for other people to understand and use. Having the program split into several distinct parts and several different files also echoes the UNIX philosophy that I've been trying to have in my programs:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out

after you've finished using them.

(From the Bell System Technical Journal)

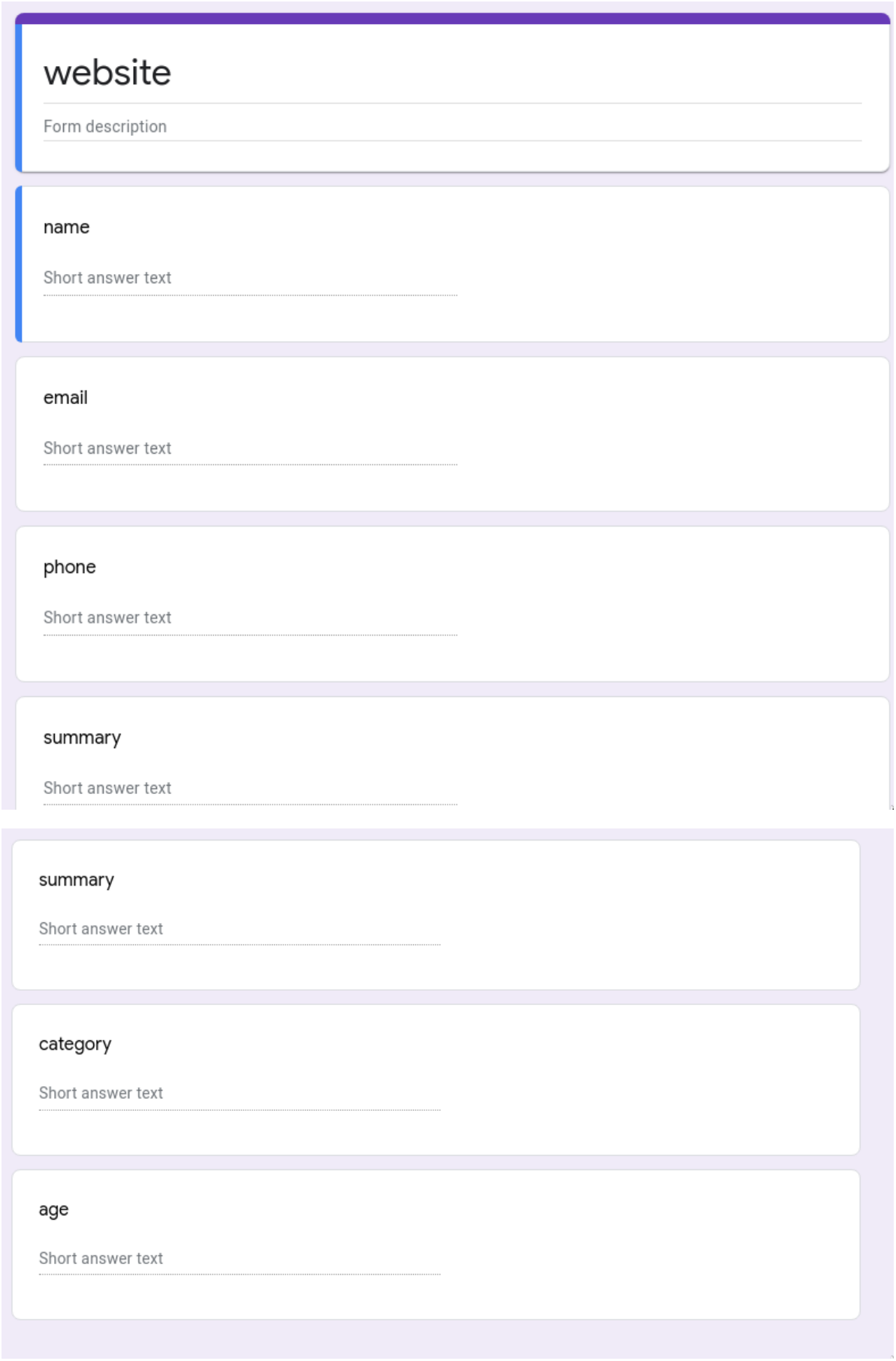
For this form, I chose to have as many precautions as possible for making sure the form was filled out correctly to give as little trouble as possible to the WATG organizers. Therefore, for each entry, I had a conditional function that checked if each box was filled in correctly based on the number and type of characters in each box. If a user got it wrong, he or she would get a notification that they had filled out a certain part of the form correctly.



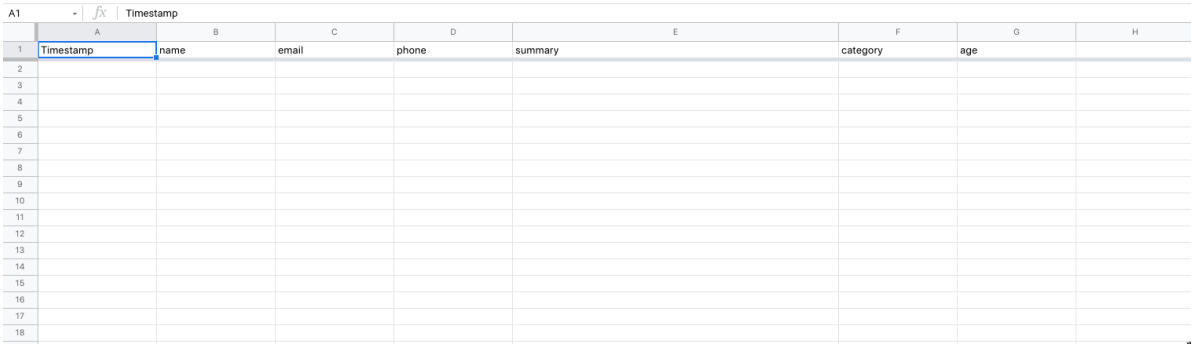
A notification that the user has not filled out his name before submitting the form

Originally, I tried to use the Google Sheets API to send the form data, but after watching countless YouTube videos, I took the easy route out that I found in one. I made each text field a variable and created a Google Form with those exact same values. Finally, I used Ajax to automatically copy and paste the values from the form into the text fields in the Google Sheet, and artificially press submit. That way, I could have all the benefits of the Google Form without any of the

disadvantages like customizing it. For more information on how I used Ajax to copy-paste form data, please refer to the appendix.



The Google Form



The Google Sheet

Twilio Notifier: Spreadsheet PYTHON

```
from __future__ import print_function
import os.path
from googleapiclient.discovery import build
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials

# If modifying these scopes, delete the file token.json.
SCOPES = ['https://www.googleapis.com/auth/spreadsheets.readonly']

# The ID and range of a sample spreadsheet.
SAMPLE_SPREADSHEET_ID = '1HMkMUoh9tdiDYsHzcO8WWRsw6oZAzUeh5aws8OipPGU'
SAMPLE_RANGE_NAME = 'Sheet1!A2:D'

class my_dictionary(dict):

    # __init__ function
    def __init__(self):
        self = dict()

    # Function to add key:value
    def add(self, key, value):
        self[key] = value

numbers = my_dictionary()



creds = None
# The file token.json stores the user's access and refresh tokens, and is
# created automatically when the authorization flow completes for the first
# time.
if os.path.exists('token.json'):
    creds = Credentials.from_authorized_user_file('token.json', SCOPES)
# If there are no (valid) credentials available, let the user log in.
```

```
if not creds or not creds.valid:
    if creds and creds.expired and creds.refresh_token:
        creds.refresh(Request())
    else:
        flow = InstalledAppFlow.from_client_secrets_file(
            'credentials.json', SCOPES)
        creds = flow.run_local_server(port=0)
    # Save the credentials for the next run
    with open('token.json', 'w') as token:
        token.write(creds.to_json())

service = build('sheets', 'v4', credentials=creds)

# Call the Sheets API
sheet = service.spreadsheets()
result = sheet.values().get(spreadsheetId=SAMPLE_SPREADSHEET_ID,
                            range=SAMPLE_RANGE_NAME).execute()
values = result.get('values', [])


def makeDictionary():
    counter = 1
    counter1 = 1
****while counter < 5:**********************
        numbers.add(counter, values[counter1])
        counter += 1
        counter1 += 1
    return numbers
```

Originally, I had decided on using Java for this part of the program, but I chose Python for 2 reasons:

- I am much more experienced with Python; I've been coding for 5 years in Python and 2 in Java
- There is much more support on-line (At least, from what I've seen -- I know Java is still the most used language in the world today, but I found the support and documentation to be a bit dated and lacking)

For the first part of my program, I simply utilized the `quickstart.py` that the Google Sheets API provided me. In short, this verifies my credentials with Google, connects to a spreadsheet that I have created, and reads the data. (For more information on this, please refer to the appendix.). What interested me the most, however, was how I was going to format this data. According to my client there were going to be at least 30-40 participants, so I needed to find a reliable data structure to use. I finally decided on using a dictionary with a counter as the key to determine which participants was going to be notified and a list as the value

which had both the time of the participant and the phone number of the participant, respectively.

Obviously, I was going to have to make a function that created this dictionary first -- just a simple declaration wouldn't do. For extra complexity points, I decided to make a short class that made this dictionary. I made an init function that initialized a dictionary with null values. After figuring out how to add a value to the dictionary, I added a helper function to the class that would allow me to more easily and quickly add entries to the dictionary and the list(s) inside it. Currently, I simply input the number of participants directly into the code (the place where I input is highlighted with asterisks), but later I would like to have another function that counts the number of rows in the Google Sheet and input this number automatically. But either way, the dictionary with the counter, name of participant, and phone number has been created.

Now, for the actual texts.

Twilio Notifier: Actual Notifier `PYTHON`

```python
# Download the helper library from https://www.twilio.com/docs/python/install
import os
import time
from twilio.rest import Client
from datetime import datetime as datetime_
from datetime import timedelta
from sheets import makeDictionary

account_sid ='ACcd5262f0c01336d3f22e5a5b54f72e2a'
auth_token = 'cde623bc05f7a79dc522a9a203de3528'
client = Client(account_sid, auth_token)
phone = '+17692082613'


def currentTime():
    n = datetime_.now()
    p = n + timedelta(minutes=10)
    now = p.strftime("%H:%M")
    return now


# Your Account Sid and Auth Token from twilio.com/console
# and set the environment variables. See http://twil.io/secure

client = Client(account_sid, auth_token)
numbers = makeDictionary()
```

```
texts = ["You have 10 minutes until your performance. Please proceed to the
stage.", "You have five minutes left until your performance."]

counter = 0

while True:
    performanceTime = list(numbers.values())[counter][0]
    if currentTime() != performanceTime:
        print("Waiting for 60 seconds to check again.")
        time.sleep(60)
        continue
    else:
        message = client.messages \
                        .create(
                            body = texts[0],
                            from_ = phone,
                            to = list(numbers.values())[counter][-1]
                        )
        print("Reminder 1 sent!")
        time.sleep(300)
        message2 = client.messages \
                        .create(
                            body = texts[1],
                            from_ = phone,
                            to = list(numbers.values())[counter][-1]
                        )
        print("Reminder 2 sent!")
        counter += 1
        if counter == len(numbers) - 1:
            print("All notifications sent. Did you enjoy the show? ")
            break
        else:
            continue
```

I copied the first few lines from the Twilio quickstart guide. (For more information, please refer to the appendix.) These lines verify my credentials -- quite similar to the Google Sheets API -- and define the phone number I am using to send (and possibly receive, if I wanted to) SMS. These lines also import a function from the previous file -- specifically, the `makeDictionary()` function which uses the `my_dictionary` class in the previous file.

Next, I had to find out a way to verify that it was the correct time to send a text -- therefore, I had to make a function that calculated the correct time so it could be used to compare against the performance times present in the dictionary. Using the Python `datetime` module, I figured out a way to create a function that could calculate the current time in `hour : minute` format. Of course, since the notifications were starting ten minutes before the actual performance time, I used

another function of `datetime`, `timedelta`, to add on ten minutes to the current time to account for the notification time.

Now it was time for the function that actually sent the texts. I chose a while loop for safe measures, since I didn't know how many performers there would be and I was worried that putting slightly incorrect parameters for the `for` loop might make the whole program stop if there were more participants than usual. First, I added a short if-statement to compare the current time (+ 10 minutes) with the performance time in the first entry of the dictionary. If it wasn't the correct time, I made the program sleep for 60 seconds to check again to prevent memory overload. I could have made the interval longer, but chose not to since I knew this program would be started quite close to the first performance, and I didn't want my texts for the rest of the performers to be a few minutes behind -- that would be disastrous. Now that I think about it, I should have made the interval shorter...

If the current time + 10 matches the performance time, the program immediately sends the first text, reminding the performer that they have 10 minutes until their performance. The program then simply sleeps for five minutes, and then sends the user the final text, saying they have five minutes until their performance. It then checks if there are any more performers left in the queue -- if there are, it stops the program after a printed goodbye, if not, it goes back to the beginning of the loop. And so, the program continues until the show is over.