202102675 이문영

이전 주차의 receive.py의 코드를 거의 그대로 가져다 사용했으며 몇가지 코드만 이번 실습 과제를 수행할 수 있게 변경하였습니다. 실습 과제는, wav파일을 읽고 해석하는 것이 아닌, 마이크로 수신하여 해석하는 방식으로 진행했습니다.

다음은 필요한 패키지를 import하고 상수와 rules를 정의한 것입니다.

```
import pyaudio, struct, numpy as np
import reedsolo


INTMAX = 2**15 - 1
UNIT = 0.1
FS = 48000
CHUNK = int(FS * UNIT)
PADDING = 20    #PADDING은 mfsk로 주파수에 대응하는 HEX를 찾을때, 얼만큼의 주파수 오차를 허용할지를 정해주는 상수입니다. 현재 20만큼의 주파수 오차를 허용하여 16진수 값을 맵핑합니다.
DATA_THRESHOLD = 300
DATA_LEN = 12
RSC_LEN = 4
SYMBOL_LEN = DATA_LEN + RSC_LEN
rules = {'START': 512,
'0': 768,
'1': 896,
'2': 1024,
'3': 1152,
'4': 1280,
'5': 1408,
```

```
  '6': 1536,
  '7': 1664,
  '8': 1792,
  '9': 1920,
  'A': 2048,
  'B': 2176,
  'C': 2304,
  'D': 2432,
  'E': 2560,
  'F': 2688,
  'END': 2944}

```

다음은 pyaudio를 활용해 마이크 입력장치로부터 들어오는 음성 데이터 stream을 읽고 이를 mfsk방식으로 해석하는 과정을 구현한 코드입니다. 이전주차와 동일합니다.

```
def receive_data():
    p = pyaudio.PyAudio()
    stream = p.open(format=pyaudio.paInt16, channels=1, rate=FS, input=True)
    print("Listening...")

    hex_data = ''
    started = False
    end_count = 0


    while True:
        raw = stream.read(CHUNK)
        chunk = struct.unpack('<' + 'h'*CHUNK, raw)

        fft = np.fft.fft(chunk)
```

```
        freqs = np.fft.fftfreq(len(chunk), d=1/FS)
        freq = abs(freqs[np.argmax(abs(fft))])
        volume = np.std(chunk)

        if volume < DATA_THRESHOLD:
            print(f"[NOISE] Volume too low ({volume:.2f}) → Ignored")
            continue


        matched = None
        for k, v in rules.items():
            if v - PADDING < freq < v + PADDING:
                matched = k
                break

        if matched == 'START':
            print(f"[START] {matched} with {freq:.1f} Hz")
            started = True
            hex_data = ''
        elif matched == 'END' and started:
            print(f"[END] END with {freq:.1f} Hz")
            end_count += 1
            if end_count >= 2:
                break
        elif matched and matched not in ['START', 'END'] and started:
            hex_data += matched
            print(f"[DATA] {matched} with {freq:.1f} Hz \n Current data: {hex_data}")
        else:
            print(f"[FREQ={freq:.1f}] Volume: {volume:.2f}")


```

```
    stream.stop_stream(); stream.close(); p.terminate()
    print("raw data(hex string) :", hex_data)
    print("length:", len(hex_data))


    try:
        byte_data = bytes.fromhex(hex_data)
    except:
        byte_data = '[DECODE ERROR]'
    print("byte text:", byte_data)

```

다음은 위 코드의 연장선상으로, 이번주차 과제의 핵심인 reedsolo패키지를 활용하여, 오류가 포함된 데이터로부터 원본 데이터를 복원하는 코드입니다.

```
    rsc = reedsolo.RSCodec(RSC_LEN) #RSC_LEN = 4
    decoded = ''
    for i in range(0, len(byte_data), SYMBOL_LEN):
    # SYMBOLEN = RSC_LEN : 4 + DATA_LEN : 12 = 16
        block = byte_data[i:i+SYMBOL_LEN]
        try:
            text_bytes = rsc.decode(block)[0]
            decoded += text_bytes.decode('utf-8')
        except Exception as e:
            print(f"[ERROR] {e} | block: {block}")

    print("복원된 텍스트:", decoded)

```

`reedsolo` 모듈의 `RSCodec` 클래스를 사용하여, 4바이트의 오류 정정 코드를 포함하는 리드-솔로몬 디코더 객체를 생성하고 `rsc` 변수에 저장합니다.

이 객체는 총 16바이트(데이터 12바이트 + 오류 정정 4바이트) 블록을 복호화할 수 있습니다.

```
rsc = reedsolo.RSCodec(RSC_LEN) #RSC_LEN = 4
```

반복문을 통해, SYMBOL_LEN 만큼의 인덱스 증가로 byte_date를 순회합니다.

```
for i in range(0, len(byte_data), SYMBOL_LEN):
```

`SYMBOL_LEN = 16` 만큼 `byte_data` 를 슬라이스하여 각 블록을 `block` 변수에 저장합니다.
이후, `rsc.decode(block)[0]` 을 사용하여 사전에 생성한 Reed-Solomon 디코더 객체로 오류를 정정하고 원래의 바이트 데이터를 복원합니다.
복원된 바이트는 `UTF-8` 로 디코딩하여 사람이 읽을 수 있는 문자열로 해석합니다.

```
    block = byte_data[i:i+SYMBOL_LEN]
    try:
        text_bytes = rsc.decode(block)[0]
        decoded += text_bytes.decode('utf-8')
    except Exception as e:
        print(f"[ERROR] {e} | block: {block}")
```

실제 실행 결과

```
[DATA] 0 with 770.0 Hz
 Current data: 707574207361416C20636173A4A4AF59696E6F20696E6A7572796D880
[DATA] 8 with 1790.0 Hz
 Current data: 707574207361416C20636173A4A4AF59696E6F20696E6A7572796D8808
[DATA] A with 2050.0 Hz
 Current data: 707574207361416C20636173A4A4AF59696E6F20696E6A7572796D8808A
[DATA] 8 with 1790.0 Hz
 Current data: 707574207361416C20636173A4A4AF59696E6F20696E6A7572796D8808A8
[END] END with 2940.0 Hz
[END] END with 2940.0 Hz
raw data(hex string) : 707574207361416C20636173A4A4AF59696E6F20696E6A7572796D8808A8
length: 60
byte text: b'put saAl cas\xa4\xa4\xafYino injurym\x88\x08\xa8'
복원된 텍스트: put sail casino injury
```