PCで学ぶ！Python音声処理入門

本日の内容

- **音声データの基本理論**（アナログ→デジタル変換）
- **主要ライブラリの役割と特徴**
- **実践：録音・再生・可視化**
- **音声エフェクトの実装**
- **WAVファイルの扱い方**
- **高度な音声処理（合成・認識・加工）**

1. 音声データの基本（理論編）

音とコンピュータの根本的な違い

音: 空気中を伝わる**連続的な圧力変化**
（アナログ信号）

コンピュータ: **0と1のデジタル信号**
のみ理解可能

課題: 連続的な波をどうやって離散的な数値に変換するか？

解決策：アナログ-デジタル変換（A/D変換）

サンプリング
（標本化）
時間を細かく刻む

→

量子化
各時点での音の
強さを数値化

サンプリング周波数の深い理解

基本概念

音の波を**一定間隔で測定**して数値化

1秒間の測定回数 = サンプリング周波数（Hz）

具体例で理解

```
# 44100Hzの場合
1秒 ÷ 44100 = 約0.0000227秒 = 約0.023ミリ秒間隔で測定
```

周波数別の特徴

周波数	用途・特徴
8000 Hz	電話品質（人の声が聞き取れる最低限）
16000 Hz	音声認識でよく使用（処理効率重視）
44100 Hz	CD音質（音楽再生の標準）
48000 Hz	プロ音響（放送・映像業界標準）
96000 Hz	ハイレゾ音源（超高音質）

ナイキスト定理と実用的意味

ナイキスト定理

「サンプリング周波数の半分までの周波数しか正確に記録できない」

実例

- 44100Hz → 22050Hzまで記録可能
- 人間の可聴域：20Hz～20000Hz
- なぜ44100Hz？ → 20000Hz × 2 + 余裕 = 約44000Hz

実用的な選択指針

```
# 用途別推奨サンプリング周波数
音声認識・合成: 16000Hz  # 効率重視
音楽再生: 44100Hz        # 品質と容量のバランス
音楽制作: 48000Hz～96000Hz  # 最高品質
```

量子化ビット深度の実践的理解

量子化とは

各サンプル点での**音の強弱を何段階で表現するか**

ビット数と表現可能範囲

```
8 bit:  2^8  = 256段階          # 古いゲーム音源レベル
16 bit: 2^16 = 65,536段階       # CD音質
24 bit: 2^24 = 16,777,216段階   # プロ用途
32 bit: 2^32 = 約42億段階       # 計算処理用
```

実際の音質への影響

16bit: 一般リスナーには充分な品質

24bit: 録音・編集時のノイズマージンが大きい

32bit: 数値計算処理で精度劣化を防ぐ

チャンネル数とステレオの仕組み

チャンネルの概念

モノラル（1ch）: [左右同じ音]
→ データ量少、音源方向不明

ステレオ（2ch）: [左の音, 右の音]
→ 立体感、楽器の定位表現可能

マルチチャンネル: **5.1**ch, **7.1**ch など
→ 映画館のような包囲音響

データ構造の違い

```
# モノラル: 1次元配列
mono_data = [0.1, -0.2, 0.3, -0.1, ...]

# ステレオ: 2次元配列（各行が1サンプル）
stereo_data = [[0.1, 0.0],   # 左チャンネル, 右チャンネル
               [-0.2, 0.1],
               [0.3, -0.1], ...]
```

PCM（パルス符号変調）とは

PCMの定義

「サンプリング・量子化された生のデジタル音声データ」

圧縮との違い

```
PCM(WAV):   [生データ] → ファイルサイズ大、音質劣化なし
MP3:        [圧縮データ] → ファイルサイズ小、音質多少劣化
FLAC:       [可逆圧縮] → サイズ中、完全復元可能
```

WAVファイルの構造

```
WAVファイル = [ヘッダ情報] + [PCMデータ]
          ↑              ↑
    (44byte程度)    (実際の音声波形)

ヘッダ情報の内容:
- サンプリング周波数、ビット深度、チャンネル数、データサイズ等
```

なぜPythonでNumpy配列なのか？

音声データの本質

「音声 = 時系列に並んだ膨大な数値の集合」

```
# 5秒間の44100Hz音声の場合
total_samples = 5 * 44100 = 220,500個の数値
```

Numpy配列の強力さ

```
import numpy as np
# 全サンプルを一括で2倍（音量UP）
louder_audio = audio_data * 2
# 全サンプルに0.5を加算（DCオフセット調整）
offset_audio = audio_data + 0.5
# 配列同士の演算（エコー効果）
echo_audio = audio_data + audio_data * 0.5
# このような操作がC言語並みの速度で実行される！
```

Numpyが音声処理に最適な理由

1. ベクトル化演算

```
# 遅い方法（Pythonループ）
result = []
for sample in audio_data:
    result.append(sample * 2)

# 高速方法（Numpy）
result = audio_data * 2  # 数百倍〜数千倍高速！
```

2. メモリ効率 & 3. ライブラリ親和性

連続メモリ領域

Pythonリストと違いメモリを効率的に使用

標準データ形式

ほぼ全ての音声ライブラリがNumpy配列でやり取り

データ型の使い分け

よく使われるデータ型

```
np.int16:    [-32768 ~ 32767]
             → WAVファイル保存、整数計算

np.float32:  [-1.0 ~ 1.0 (通常)]
             → リアルタイム処理、メモリ節約

np.float64:  [-1.0 ~ 1.0 (より高精度)]
             → 高品質音声分析、PyWorld等
```

型変換の実例

```
# 録音データ（sounddevice） → float32
recording = sd.rec(...)  # shape: (samples,), dtype: float32

# WAV保存用に変換
wav_data = (recording * 32767).astype(np.int16)

# 分析用に変換
analysis_data = recording.astype(np.float64)
```

主要ライブラリの役割分担-1

音声I/O（入出力）レイヤー

sounddevice
リアルタイム録音・再生（低遅延）

soundfile
多様な音声ファイル読み書き

scipy.io.wavfile
WAVファイル専用、軽量

数値計算・処理レイヤー

numpy
基本配列演算、エフェクト処理

scipy
高度な信号処理、フィルタ処理

matplotlib
波形・スペクトログラム可視化

主要ライブラリの役割分担-2

AI・高度処理レイヤー

pyopenjtalk
オフライン日本語音声合成

vosk
オフライン音声認識

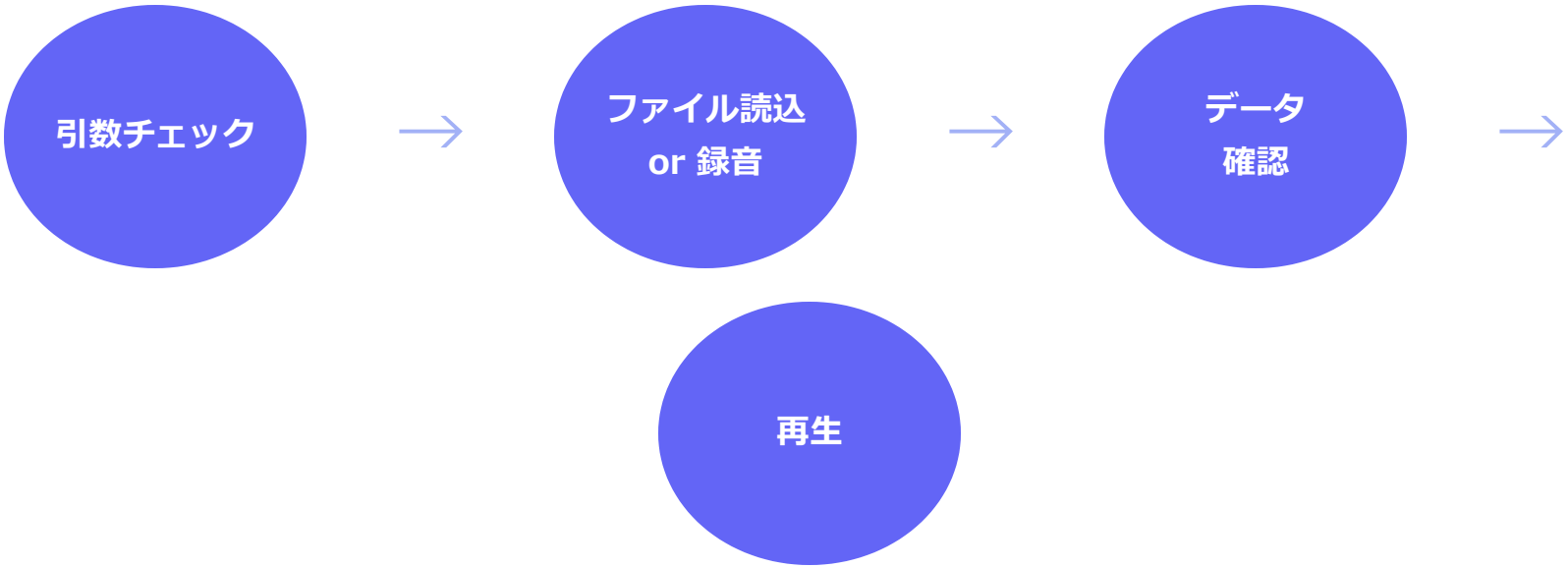
pyworld
音声分析・変調（ボコーダー）

2. 最初の実践：録音と再生

practice_1_record_playback.py の設計思想

- **シンプルさ**: 最小限のコードで音声処理を体験
- **柔軟性**: コマンドライン引数でファイル指定可能
- **拡張性**: 他のスクリプトのベースとして機能
- **教育効果**: sounddeviceとNumpy配列の関係を理解

処理の流れ



sounddeviceの核心機能

sd.rec()の詳細パラメータ

```
recording = sd.rec(
    frames=int(DURATION * FS),  # 録音サンプル数
    samplerate=FS,              # サンプリング周波数
    channels=1,                 # チャンネル数
    dtype='float32'             # データ型（省略時の推奨）
)
```

戻り値の特徴

```
print(f"データ型: {recording.dtype}")      # float32
print(f"形状: {recording.shape}")          # (220500,) または (220500, 1)
print(f"値の範囲: [{recording.min():.3f}, {recording.max():.3f}]")
```

sd.wait()の重要性

非同期処理の制御

```
# 録音開始（非ブロッキング）
recording = sd.rec(int(DURATION * FS), samplerate=FS, channels=1)
print("録音中...")  # すぐに実行される

# 録音完了まで待機（ブロッキング）
sd.wait()
print("録音完了")  # 録音終了後に実行される
```

wait()を忘れた場合の問題

```
recording = sd.rec(...)
# sd.wait() なし
sd.play(recording, FS)  # まだ録音中のデータを再生 → ノイズ！
```

soundfileの詳細機能

対応フォーマット

```
# 読み込み可能な主要フォーマット
WAV, FLAC, OGG, AIFF, AU, RAW, HTK, SDS, WAVEX, SD2, CAF, WVE

# 使用例
data, sr = sf.read('input.flac')      # FLACファイル読み込み
sf.write('output.wav', data, sr)      # WAVファイル書き込み
```

dtype指定の重要性

```
# sounddeviceと合わせるため、float32を明示
myrecording, FS = sf.read(args.file, dtype='float32')

# 指定しない場合、ファイルによってはfloat64になることがある
# → sounddeviceとのデータ型不整合でエラーの可能性
```

実行パターンと活用法

基本実行

```
# 5秒録音して即座に再生
python practice_1_record_playback.py
```

ファイル再生

```
# 指定ファイルを再生
python practice_1_record_playback.py -f my_voice.wav
python practice_1_record_playback.py --file background_music.flac
```

デバッグ・検証での活用

```
# 他のスクリプトで生成したファイルの確認
python practice_4_wav.py -o test.wav          # ファイル生成
python practice_1_record_playback.py -f test.wav  # 生成結果確認
```

活用のポイント: このスクリプトは他の処理スクリプトの動作確認にも使える万能ツール

データ形状の理解

モノラルの場合

```
# 形状: (samples,)
myrecording.shape  # (220500,)
myrecording.ndim   # 1

# アクセス方法
first_sample = myrecording[0]     # スカラー値
```

ステレオの場合

```
# 形状: (samples, channels)
stereo_data.shape  # (220500, 2)
stereo_data.ndim   # 2

# アクセス方法
left_channel = stereo_data[:, 0]   # 左チャンネル全体
right_channel = stereo_data[:, 1]  # 右チャンネル全体
first_frame = stereo_data[0, :]    # 最初のステレオフレーム [L, R]
```

第1部まとめ

習得した基礎知識

音声デジタル化
サンプリング・量子化の仕組み

ライブラリ構成
sounddevice, soundfile, numpyの役割

データ構造
モノラル・ステレオの配列形状

実践スキル
録音・再生・ファイル操作

重要なポイント

- **44100Hz = CD音質**: ナイキスト定理による理論的根拠
- **Numpy配列**: 音声処理における標準データ形式
- **float32 vs int16**: 用途に応じたデータ型選択
- **非同期処理**: sd.wait()の重要性