Для работы с файлами разметки Praat (формат .TextGrid) существует библиотека TextGridTools. Её можно установить через командную строку с помощью pip:

```
!pip install tgt
```

По ссылке расположена документация, с которой стоит ознакомиться:

https://textgridtools.readthedocs.io/en/stable/api.html

```
import tgt
```

Прочитаем TextGrid:

```
!wget https://pkholyavin.github.io/mastersprogramming/cta0001.TextGrid
```

```
grid = tgt.io.read_textgrid("cta0001.TextGrid")
```

Посмотрим, что внутри у полученного объекта, с помощью функции `dir()` (не считая служебных методов, которые начинаются с нижнего подчёркивания):

```
[i for i in dir(grid) if not i.startswith("_")]
```

В атрибуте tiers хранятся все уровни:

```
grid.tiers
```

Получим названия всех уровней:

```
grid.get_tier_names()
```

Получим уровень по названию:

```
grid.get_tier_by_name("words")
```

Объект класса TextGrid - итерируемый объект, мы можем получить все уровни простым циклом:

```
for tier in grid:
    print(tier)
```

Каждый уровень (объект класса IntervalTier или PointTier) - это тоже итерируемый объект:

```
word_tier = grid.get_tier_by_name("words")
for interval in word_tier:
    print(interval)
```

Посмотрим, что внутри у объекта IntervalTier:

```
[i for i in dir(word_tier) if not i.startswith("_")]
```

Некоторые полезные атрибуты:

```
print(word_tier.name)
print(word_tier.start_time)
print(word_tier.end_time)
# не забудем, что, в отличие от Wave Assistant, Praat хранит время в секундах
```

Посмотрим, что внутри у элементов аннотации:

```
one_word = word_tier[0]
[i for i in dir(one_word) if not i.startswith("_")]
```

Получим эти атрибуты:

```
print(one_word.start_time)
print(one_word.end_time)
print(one_word.text)
```

А что внутри у класса Point?

```
point = grid.get_tier_by_name("word boundaries")[0]
[i for i in dir(point) if not i.startswith("_")]
```

Задание для выполнения в классе: напишите цикл, который перебирает все интервалы из уровня `"phonetic real"` и выводит на экран название каждого интервала и его серединную точку.

Создадим пустой TextGrid:

```
grid = tgt.core.TextGrid()
```

Добавим новый IntervalTier:

```
new_tier = tgt.core.IntervalTier(name="new tier")
grid.add_tier(new_tier)
```

Добавим в него новый интервал, который начинается в 0 с, заканчивается в 1 с и называется "some text"

```
new_tier.add_interval(tgt.core.Interval(0, 1.0, "some text"))
new_tier
```

Добавим новый PointTier:

```
new_point_tier = tgt.core.PointTier(name="new point tier")
grid.add_tier(new_point_tier)
new_point_tier.add_point(tgt.core.Point(0.5, "some text"))
new_point_tier
```

Запишем в разных форматах:

```
tgt.io.write_to_file(grid, "new_grid_short.TextGrid", format="short")
tgt.io.write_to_file(grid, "new_grid_long.TextGrid", format="long")
```

У файлов .TextGrid есть "длинный" и "короткий" варианты. Они содержат одну и ту же информацию, но "длинный" больше подходит для того, чтобы читать его глазами.

```
!wget https://pkholyavin.github.io/mastersprogramming/cta0001.seg_B2
```

Вспомним, как обрабатывать метки парами:

```
from itertools import product
letters = "GBRY"
nums = "1234"
levels = [ch + num for num, ch in product(nums, letters)]
```

```
level_codes = [2 ** i for i in range(len(levels))]
code_to_level = {i: j for i, j in zip(level_codes, levels)}
level_to_code = {j: i for i, j in zip(level_codes, levels)}
def read_seg(filename: str, encoding: str = "utf-8-sig") -> tuple[dict, list[dict]]:
    with open(filename, encoding=encoding) as f:
        lines = [line.strip() for line in f.readlines()]

    # найдём границы секций в списке строк:
    header_start = lines.index("[PARAMETERS]") + 1
    data_start = lines.index("[LABELS]") + 1

    # прочитаем параметры
    params = {}
    for line in lines[header_start:data_start - 1]:
        key, value = line.split("=")
        params[key] = int(value)

    # прочитаем метки
    labels = []
    for line in lines[data_start:]:
        # если в строке нет запятых, значит, это не метка и метки закончились
        if line.count(",") < 2:
            break
        pos, level, name = line.split(",", maxsplit=2)
        label = {
            "position": int(pos) // params["BYTE_PER_SAMPLE"] // params["N_CHANNEL"],
            "level": code_to_level[int(level)],
            "name": name
        }
        labels.append(label)
    return params, labels


def print_label_pairs(filename):
    params, labels = read_seg(filename)
    for start, end in zip(labels, labels[1:]):
        print(start, end)
```

Задание для выполнения в классе: напишите функцию, которая принимает на вход имя файла .seg и делает следующее:

1. Читает из файла метки и параметры (вызывая готовую функцию `read_seg()`)
2. создаёт новый TextGrid и уровень IntervalTier
3. Добавляет новый уровень в новый TextGrid
4. Перебирает циклом все пары соседних меток
5. Добавляет в уровень все интервалы, полученные таким образом (соответственно, время начала каждого интервала - позиция левой метки в паре, время конца - позиция правой, текст - имя левой метки)

6. Записывает получившийся объект TextGrid в файл .TextGrid

Не забудем, что в файлах .TextGrid время хранится **в секундах**! Чтобы перевести время из отсчётов в секунды, нужно разделить его на частоту дискретизации.

Откроем полученный файл в Praat и посмотрим на него.

Домашнее задание: напишите без использования сторонних библиотек функцию, которая получает на вход имя файла .TextGrid (на ваш выбор – короткого или длинного), а возвращает список словарей вида:

```
[
    {
        "name": "название уровня",
        "type": "interval" или "point",
        "data":
        [
            #для interval tier
            (метка начала в миллисекундах, метка конца в миллисекундах, текст метки),
            #для point tier
            (метка в миллисекундах, текст метки),
        ]
    }
]
```

Помните, что спецификация текстгридов позволяет иметь много уровней с одинаковым названием.

Альтернативное домашнее задание: написать программу, которая:

1. Обрабатывает все файлы .seg в архиве cta_seg
2. Для каждого аллофона вычисляет его среднюю длительность (в секундах) и стандартное отклонение
3. Для файла cta0001 генерирует файл .TextGrid с двумя уровнями. Первый должен содержать информацию из .seg_B1 (границы звуков и их названия), а второй должен совпадать с первым, но имя каждого интервала должно содержать не название звука, а его длительность, нормализованную путём z-нормализации (https://en.wikipedia.org/wiki/Standard_score) и округлённую до 3 знаков после запятой.

Чтобы вычислить нормализованную длительность звука, нужно из его физической длительности (в секундах) вычесть среднее значение длительности этого аллофона **по всему корпусу** и разделить на стандартное отклонение.

Т.е. чтобы сделать это для, например, звука [u0] из слова "юрий", нужно определить среднее и ст. отклонение по всем звукам [u0] из всего корпуса и использовать эти значения. Для звука [r'] эти значения уже будут другими.

В качестве иллюстрации: сгенерируем массив из 100 случайных чисел и вычислим его среднее значение и стандартное отклонение.

```
import numpy as np
```

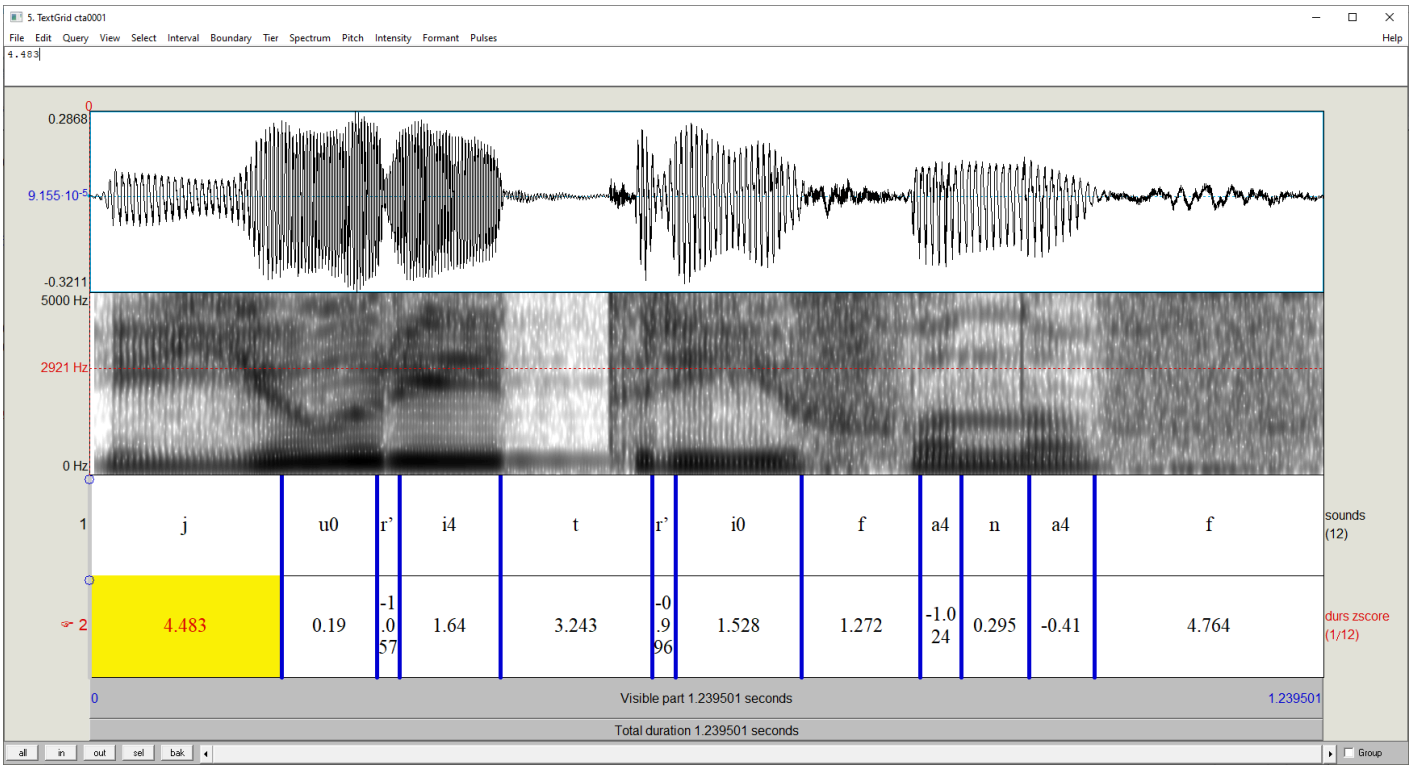
```
rng = np.random.default_rng()
nums = rng.normal(loc=3, scale=1.5, size=100)  # нормальное распределение с МО=3 и СКО=1.5
mean_value = np.mean(nums)
st_dev = np.std(nums)
print(mean_value, st_dev)
```

Сгенерируем ещё одно случайное число из того же распределения и нормализуем его:

```
new_num = rng.normal(loc=3, scale=1.5)
norm_num = (new_num - mean_value) / st_dev
print(new_num, norm_num)
```

Не забудьте открыть полученный файл в Praat, чтобы убедиться в том, что он:

1. Открывается
2. Содержит нужные данные
3. Полученные значения адекватны - получиться должно примерно следующее (точные значения могут отличаться):



Дополнительный материал: конвертация из триграфов Praat в символы Unicode

Для хранения символов, не входящих в таблицу ASCII (символов МФА, кириллицы и других алфавитов), Praat пользуется своей собственной системой: каждому символу, не входящему в ASCII, сопоставляется т.н. триграф. Под триграфом понимается последовательность из обратного слеша \ и двух символов ASCII. Например, символу ɨ соответствует триграф \i-.

https://www.fon.hum.uva.nl/praat/manual/TextGrid_file_formats.html
https://www.fon.hum.uva.nl/praat/manual/Special_symbols.html

Всего в Praat определено несколько сотен таких триграфов, набор периодически расширяется (всего их возможно несколько тысяч, что, конечно, гораздо меньше, чем количество символов, определённых в Unicode).

Файлы .TextGrid могут содержать как один вариант записи, так и другой. Для конвертации рекомендуется использовать саму программу Praat (вручную, написав скрипт или через библиотеку `parselmouth`). Однако можно заняться конвертацией самостоятельно, если очень хочется. Для этого посмотрим на ту часть исходного кода Praat, которая отвечает за конвертацию.

```
# скачаем файлы исходного кода
!wget https://raw.githubusercontent.com/praat/praat/master/kar/longchar.cpp
!wget https://raw.githubusercontent.com/praat/praat/master/kar/UnicodeData.h
```

```
with open("UnicodeData.h") as f:
    lines = f.readlines()

# здесь хранятся коды символов Unicode в шестнадцатеричном представлении
unicode_vals = {}
for line in lines:
    if not line.startswith("#define"):
        continue
    _, name, val = line.strip().split()
    if not name.startswith("UNICODE"):
        continue
    unicode_vals[name] = chr(int(val, 16))


import re

with open("longchar.cpp") as f:
    lines = f.readlines()

trigraph2unicode = {}

# здесь хранится таблица соответствий
for line in lines:
    line = line.replace("\\'", "'")
    line = line.replace('\\"', '"')
    # напишем регулярное выражение, которое ищет в строке таблицы символы, входящие в тригра
    m = re.search("('[^,]+'), ?('[^,]+'),.+(UNICODE_\w+)", line)
    if m is None:
        continue
    ch1, ch2, name = m[1][1:-1], m[2][1:-1], m[3]
    if ch2 == " ":
        continue
    trigraph2unicode["\\" + ch1 + ch2] = unicode_vals[name]


# сделаем словарь для обратной конвертации
unicode2trigraph = {j: i for i, j in trigraph2unicode.items()}


# проверим:
trigraph2unicode["\\O\""]
```

```
'Ö'
```

```
unicode_string = "bɨl tʲixʲij sʲerɨj vʲetʃʲir"
trigraph_string = unicode_string.translate(str.maketrans(unicode2trigraph))
print(trigraph_string)
```