Poio API Documentation

Release 0.3.4

Peter Bouda

May 18, 2018

Contents

1 Contents **3**
- 1.1 Introduction to Poio API . . . 3
- 1.2 Conversion of file formats and annotation mapping . . . 9
- 1.3 Parser and Writer classes to map from and to file formats . . . 14
- 1.4 Linguistic analysis and pipelines based on GrAF graphs . . . 22

2 API documentation **25**
- 2.1 PoioAPI Package . . . 25
- 2.2 PoioAPI IO Package . . . 28

3 Indices and tables **29**

Python Module Index **31**

Poio API provides access to language documentation data and a wide range of annotations schemes stored in different file formats.

The project's homepage is: http://media.cidles.eu/poio/poio-api/

CHAPTER 1

Contents

1.1 Introduction to Poio API

Poio API is a free and open source Python library to access and search data from language documentation in your linguistic analysis workflow. It converts file formats like Elan's EAF, Toolbox files, Typecraft XML and others into annotation graphs as defined in ISO 24612. Those graphs, for which we use an implementation called "Graph Annotation Framework" (GrAF), allow unified access to linguistic data from a wide range sources.

Think of GrAF as an assembly language for linguistic annotation, then Poio API is a libray to map from and to higher-level languages.

Poio API is developed as a part of the curation project of the F-AG 3 within CLARIN-D.

References:

- ISO 24612: http://www.iso.org/iso/catalogue_detail.htm?csnumber=37326
- Graph Annotation Framework (GrAF): http://www.xces.org/ns/GrAF/1.0/

1.1.1 Quick Example

This block of code loads a Elan EAF file as annotation graph and writes the data as html table into a file:

```
# imports
import poioapi.annotationgraph

# Load the data from EAF file
ag = poioapi.annotationgraph.AnnotationGraph.from_elan("elan-example3.eaf")

# Output as html
import codecs
f = codecs.open("example.html", "w", "utf-8")
f.write(ag.as_html_table(False, True))
f.close()
```

To try it out you may download the example file from the Elan homepage.

1.1.2 Data Structure Types

We use a data type called *DataStructureType* to represent annotation schemes in a tree. A simple data structure type describing that the researcher wants to tokenize a text into words before adding a word-for-word translation and a translation for the whole utterance looks like this:

```
[ 'utterance', [ 'word', 'wfw' ], 'translation' ]
```

A slightly more complex annotation schema is GRAID (Grammatical Relations and Animacy in Discourse), developed by Geoffrey Haig and Stefan Schnell. GRAID adds the notion of clause units as an intermediate layer between utterance and word and three more annotation tiers on different levels:

```
[ 'utterance',
        [ 'clause unit',
                [ 'word', 'wfw', 'graid1' ],
        'graid2' ],
  'translation', 'comment' ]
```

One advantage in representing annotation schemes through those simple trees, is that the linguists instantly understand how such a tree works and can give a representation of "their" annotation schema. In language documentation and general linguistics researchers tend to create ad-hoc annotation schemes fitting their background and then normally start to create only those annotations related to their current research project. This is for example reflected in an annotation software like ELAN, where the user can freely create tiers with any names and arrange them in custom hierarchies. As we need to map those data into our internal representation, we try to ease the creation of custom annotation schemes that are easy to understand for users. For this we will allow users to create their own data structure types and derive the annotation schemes for GrAF files from those structures.

In Poio API there are several data structure types pre-defined as classes in the module *poioapi.data*, for example:

- *poioapi.data.DataStructureTypeGraid*
- *poioapi.data.DataStructureTypeMorphsynt*

The user of the API can of course create her own fixed data structure type, by deriving a custom class from the base class *poioapi.data.DataStructureType*. In you workflow you might also create an object with your own tier hierarchy by passing a list of lists (as in the examples above) when creating an object from *DataStructureType*:

```
import poioapi.data

my_data_structure = poioapi.data.DataStructureType(
    [ 'utterance', [ 'word', 'wfw' ], 'translation' ])
```

If you create an annotation graph from one of the supported file formats, the hierarchies that are present in file are accesible via the *tier_hierarchies* property of the annotation graph object. As an example, we use the example file from the Elan homepage:

```
import poioapi.annotationgraph

ag = poioapi.annotationgraph.AnnotationGraph.from_elan("elan-example3.eaf")
print(ag.tier_hierarchies)
```

Which will output:

```
[

  ['utterance..K-Spch'],

  ['utterance..W-Spch',
    ['words..W-Words',
      ['part_of_speech..W-POS']
    ],
    ['phonetic_transcription..W-IPA']
  ],

  ['gestures..W-RGU',
    ['gesture_phases..W-RGph',
      ['gesture_meaning..W-RGMe']
    ]
  ],

  ['gestures..K-RGU',
    ['gesture_phases..K-RGph',
      ['gesture_meaning..K-RGMe']
    ]
  ]

]
```

This is a list of tier hierarchies. In this case, there are four hierarchies in the .eaf file: two for each speaker, where one has the root tier with utterances (*utterance..K-Spch* and *utterance..K-Spch*), the other one with the root tier for gestures (*gestures..W-RGU* and *gestures..K-RGU*)

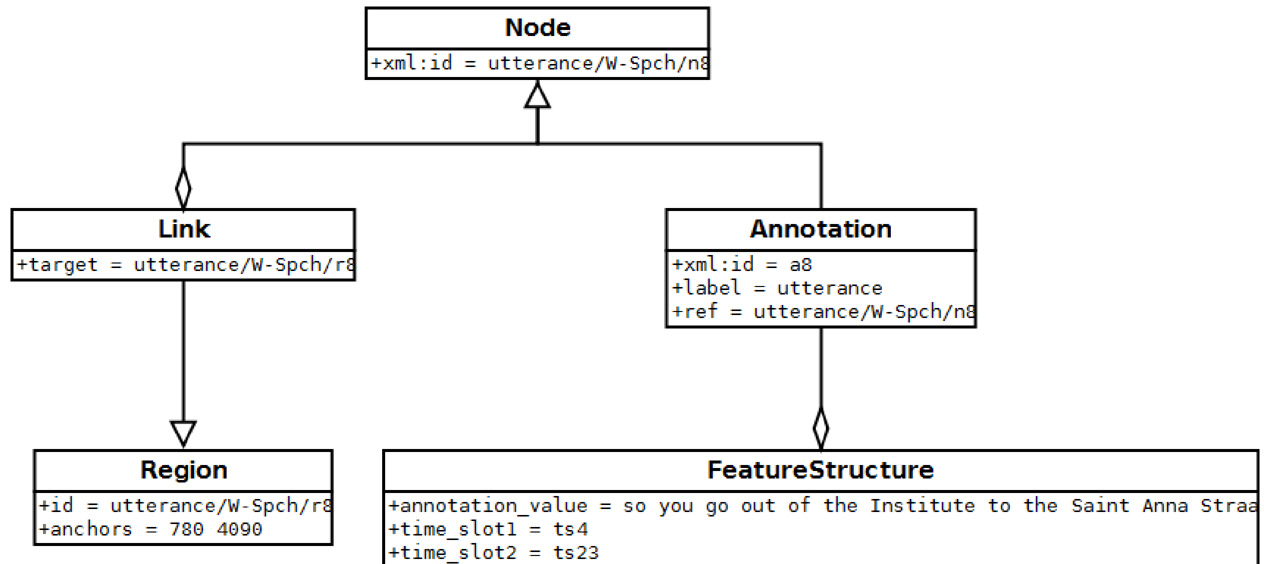
The user can now easily create an instance of the class *DataStructureType* with one of the hierarchies.

Per default, the first tier hierarchy from the file is that as the current active hierarchy (for example for queries or HTML output). To set another tier hierarchy as the default hierarchy you can set the attribute *structure_type_handler* to one of the other hierarchies in the data:

```
ag.structure_type_handler = poioapi.data.DataStructureType(
    ag.tier_hierarchies[1])
```

1.1.3 Structure of GrAF graphs in Poio API

To represent data from tier-based annotations, Poio API internally uses the library graf-python to store data and annotations. Those data structures conform to the so-called GrAF standard and consist of **nodes** and **edges* enriched by **feature structures** that contain the linguistic annotations. The nodes itself are **linked** to the primary data (text, audio, video, . . .) via **regions**. The following schema pictures the content of one node:



Edges are then simple connections between individual nodes that can also have an *Annotation* with the same feature structures as the nodes.

Poio API only uses a subset of all possible GrAF graphs to represent tier-based annotations. That means that Poio API will automatically only create certain edges between nodes and their annotations, to represent a parent-children relationship between annotations that are on different tiers in the original annotation file. Poio API will not create any additional edges between annotations on one single tier and between annotations of tiers that are not parent or child of each other. In addition to this, Poio API will also create some fixed feature structures from the content of annotations when you load a file. A standard string annotation (i.e. the part-of-speech tag in a Typecraft XML file) is stored as feature *annotation_value* in a node. See section *Example: GrAF from an Elan EAF file* for an in-depth description of such a GrAF structure when you load an Elan EAF file.

You, as a user, are of course free to create any edges or add any feature structures and features when you process the graphs in your worklow. You have access to the GrAF object in Poio API after you loaded the content of a file into an object of the class `AnnotationGraph`. The GrAF object is stored in the property *graf*:

> # imports import poioapi.annotationgraph
>
> # Load the data from EAF file ag = poioapi.annotationgraph.AnnotationGraph.from_elan("elan-example3.eaf")
>
> my_graf_object = ag.graf
>
> # … then do something with the GrAF object…

Keep in mind that probably none of your custom changes might be saved to some of the supoprted output file formats like Elan EAF or Typecraft XML. If you want to make sure that all your data persists when reading and writing files you should store the graphs as GrAF-XML, which will contain all information in the GrAF object:

```
# ... you did something with ag.graf ...

# save it
ag.to_graf("my_graf_object.hdr")

# load again
ag = poioapi.annotationgraph.AnnotationGraph.from_graf("my_graf_object.hdr")
```

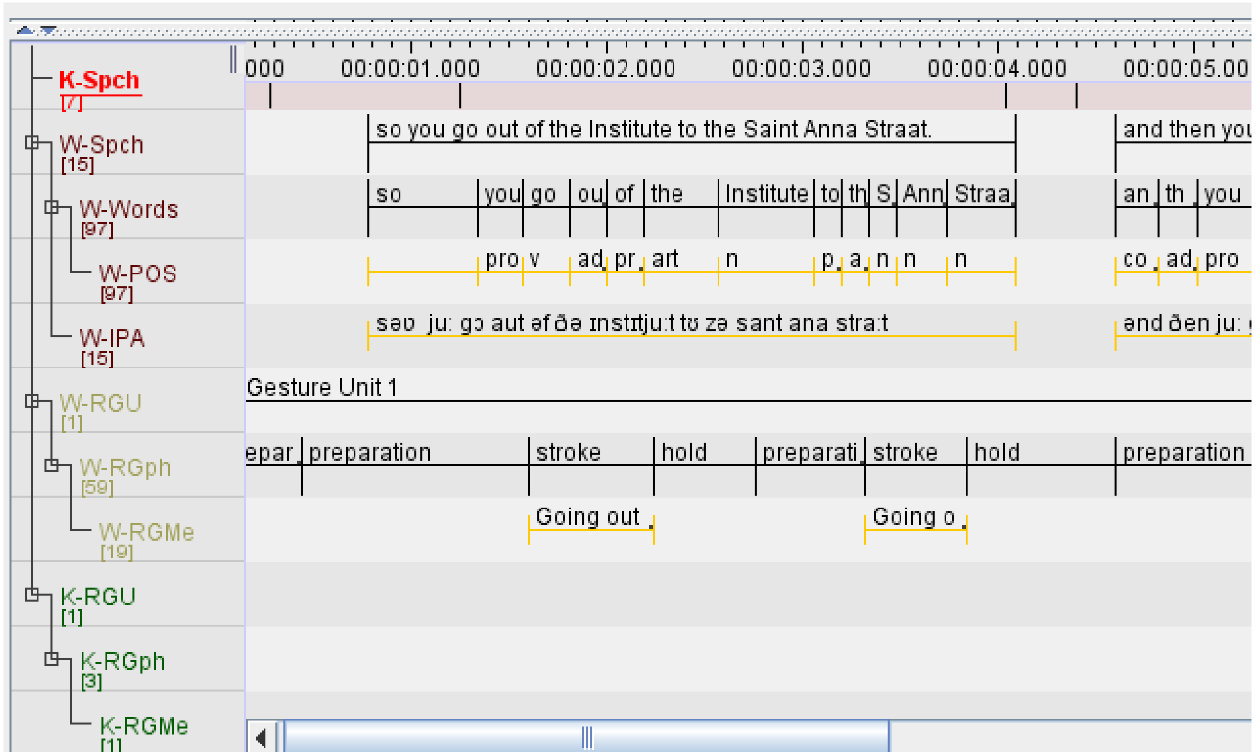
Other file formats might only store a subset of the content of *ag.graf*.

1.1.4 Example: GrAF from an Elan EAF file

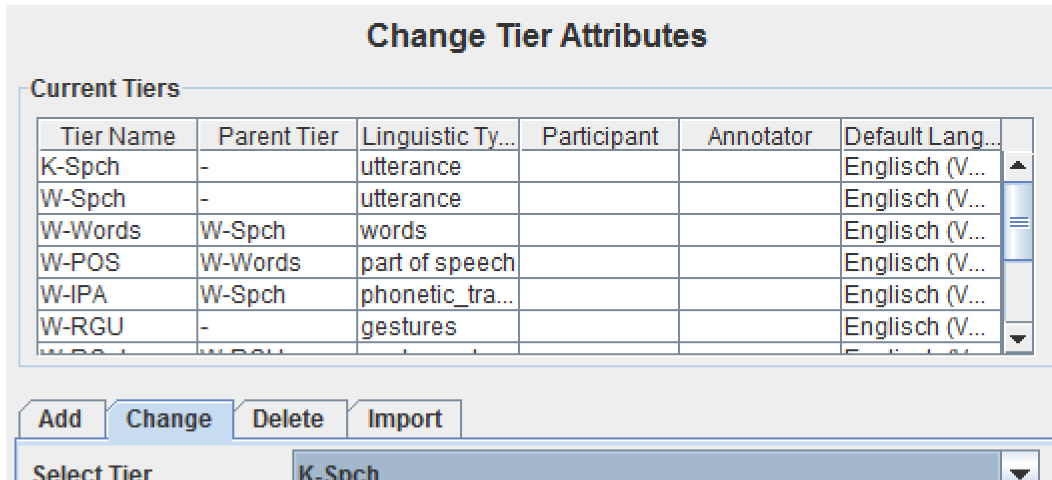
Elan is a widely used transcription and annotation software developed at the Max-Planck-Institute in Nijmegen. Due to its popularity the file format used by Elan, an XML format called "EAF" ("Elan Annotation Standard"), has become the de facto standard in language documentation and is used by several project in qualitative and quantitative language typology. Poio API fully supports to convert EAF files to GrAF annotation graphs and back again without any loss of information.

Basically, Poio API extracts all <*annotation*> tags from the EAF file and converts them to GrAF nodes and annotations. The <*time_slot*> tags in the EAF file are used to create the regions for the nodes in GrAF. The rest of the EAF file is left intact and stored as a separate file *prefix-extinfo.xml* in parallel to the other GrAF files as described in section *Structure of GrAF graphs in Poio API* (where *prefix* is again the base name of the header file of GrAF).

The structure of the GrAF files is defined by the tier hierarchy in the Elan file. As an example we will use the example data file that you may download from the the Elan website (next to "Example Set"). If you open those files in Elan and sort the tiers by hierarchy you will have the following tier hierarchy:



In this case, there are four *root tiers* with annotations: *K-Spch*, *W-Spch*, *W-RGU* and *K-RGU*. The latter three each has several child tiers. Each tier has a *linguistic type*, which you can see if you click on *Tier -> Change Tier Attributes...* :



In this case the tier *K-Spch* has the linguistic type *utterance*, and so on. These linguistic types correspond to the names in the data structure types of Poio API (see section *Data Structure Types*). Which means that if you transform an EAF file into GrAF files with Poio API it will create one file for each of the linguistic types. Each of those files file will contain all the annotations of all the tiers that have the corresponding linguistic type. In our example, Poio API will create one file *prefix-utterance.xml* that contain the annotations from the tiers *K-Spch* and *W-Spch*. The file *prefix-words.xml* will then contain all annotations from tier *W-Words* with links to the parent annotations in *prefix-utterance.xml*. You can find an example of the GrAF structure for the sample EAF file on Github.

The first annotation of the tier *W-Spch* with the annotation value "so you go out of the Institute to the Saint Anna Straat." looks like this in GrAF:

```
<node xml:id="utterance..W-Spch..na8">
  <link targets="utterance..W-Spch..ra8"/>
</node>
<region anchors="780 4090" xml:id="utterance..W-Spch..ra8"/>
<a as="utterance" label="utterance" ref="utterance..W-Spch..na8" xml:id="a8">
  <fs>
    <f name="annotation_value">so you go out of the Institute to the Saint Anna␣
↪Straat.</f>
  </fs>
</a>
```

The *<node>* is linked to a *<region>* that contains the values of the time slots of the original EAF file. The annotation *<a>* for the node has a feature structure *<fs>* with one features *<f>* for the annotation value.

The first annotation of *W-Spch* in *prefix-words.xml* looks like this:

```
<node xml:id="words..W-Words..na23">
  <link targets="words..W-Words..ra23"/>
</node>
<region anchors="780 1340" xml:id="words..W-Words..ra23"/>
<edge from="utterance..W-Spch..na8" to="words..W-Words..na23" xml:id="ea23"/>
<a as="words" label="words" ref="words..W-Words..na23" xml:id="a23">
  <fs>
    <f name="annotation_value">so</f>
  </fs>
</a>
```

The node for the word annotation is similar to the utterance node, except for an additional *<edge>* tag that links the node to the corresponding utterance node. Nodes like this are created for alle the annotations in the EAF file. When the original annotation does not link to the video or audio file via a timeslot, for example because it is on a tier with

a linguistic type that has the stereotype *Time Subdivision*, then no region and no link will be created for the node in GrAF. As an example, here is the POS annotation that is linked to a word node via an edge:

```
<node xml:id="part_of_speech..W-POS..na121"/>
<edge from="words..W-Words..na24" to="part_of_speech..W-POS..na121" xml:id="ea121"/>
<a as="part_of_speech" label="part_of_speech" ref="part_of_speech..W-POS..na121"␣
↪xml:id="a121">
  <fs>
    <f name="annotation_value">pro</f>
  </fs>
</a>
```

References:

- EAF Format: http://www.mpi.nl/tools/elan/EAF_Annotation_Format.pdf
- Information about Elan: http://tla.mpi.nl/tools/tla-tools/elan/elan-description/
- Elan Tools and Documentation: http://tla.mpi.nl/tools/tla-tools/elan/download/

1.2 Conversion of file formats and annotation mapping

From the user's perspective the conversion of file formats with mapping of annotations (within or across tiers) consists of three steps:

- Generate an empty mapping file. This file will contain all annotation labels that could not be mapped to the output file format. The mapping file is a JSON file.
- Edit the JSON file and add missing output annotation labels.
- Run the conversion with the mapping file as additional input.

We will demonstrate the three steps with an example conversion from Toolbox to Typecraft. Typecraft has a fixed set of annotations that we can use, while most Toolbox files use a variety of tag sets. This makes Toolbox to Typecraft conversion examplary use case, because in most cases the user has to define a mapping so that she can import her files into the Typecraft web application. The general mechanism is of course applicable to other conversion workflows as well.

1.2.1 Conversion on the command line

To convert a file on the command the Poio API source contains a script *poio_converter.py* in the *examples* folder. To convert a file, you have to specify the input file and the output file and the type of both files (for example *toolbox* and *typecraft*). To convert a file from the Toolbox format to the Typecraft XML format you call the script like this:

```
$ python poio_converter.py -i toolbox -o typecraft toolboxfile.txt typecraftfile.xml
```

Any annotations that are not part of the Typecraft tagset will be left empty in the output file, as we cannot import the XML into Typecraft if there is any annotation that does not belong to the tagset.

1.2.2 The JSON mapping file

To map annotation from your Toolbox tagset to the Typecraft tagset you have to create a special mapping JSON file first. This is done with the *-m* option of the converter script:

$ python poio_converter.py -m -i toolbox -o typecraft toolboxfile.txt typecraft_mapping.json

This will generate a file *typecraft_mapping.json* in the current folder. The content of the file might look like this:

```
{
        "gloss": {
        "1PAST": "",
            "FV7": "",
        "PRN": ""
    },
        "part of speech": {
        "int": ""
    }
}
```

The JSON structure consists of two block for *gloss* and *part of speech* tags in this case. The right hand sind of each tag is an empty string. You have to fill in the Typecraft tags that you want to map to for each of your Toolbox tags. For example:

```
{
    "gloss": {
            "1PAST": "PAST",
        "FV7": "FV",
            "PRN": ""
    },
    "part of speech": {
            "int": "PROint"
    }
}
```

If you don't want to map one of the tags then you can just leave the right hand side empty, as in the example above for the Toolbox tag "PRN". You can now pass the JSON mapping file to the converter via the *-t* option when you run the full conversion:

```
$ python poio_converter.py -m -i toolbox -o typecraft -t typecraft_mapping.json␣
↪toolboxfile.txt typecraftfile.xml
```

In some cases you might want to map a tag to another tier, for example from the "gloss" tier to the "part of speech" tier. To map between tiers you have to specify the tier name on the right hand side of the mapping. Just use a list and put the output tier name as the first element in the list. For example, to map the "PRN" tag from Toolbox to the "part of speech" tier of Typecraft:

```
"PRN": [ "part of speech", "PN"]
```

1.2.3 Tier names of the input file

Poio API specifies default tier names for each input file format. For example, in the case of toolbox the tier names of the "gloss" tier might be "ge" or "g". Those names are defined in the Toolbox software and might be changed by the user. If your gloss names are different from the default names in Poio API you can define new tier names by adding a *tier_names* map to the JSON mapping file. To use an additional gloss tier name "gloss" you can start the JSON file with:

```
{
    "tier_mapping": {
        "gloss": [
                "ge",
                "g",
```

(continues on next page)

(continued from previous page)

```
        "gloss"
            ],
    },
        "gloss": {
    [ ... rest of the file as above ... ]
```

1.2.4 More examples of JSON mapping files

More examples of JSON mapping files can be found in the Poio API repository. We already defined two default mappings for corpora from different sources. One is the default mapping for Toolbox files:

https://github.com/cidles/poio-api/blob/master/src/poioapi/mappings/TOOLBOX_TYPECRAFT.json

The other mapping is used for data from certain Word files that contain interlinear glossed text and is just referenced here as an example:

https://github.com/cidles/poio-api/blob/master/src/poioapi/mappings/MANDINKA_TYPECRAFT.json

1.2.5 Map programmatically in Python

You can also use Poio API directly from Python to generate and apply a JSON mapping file when you convert from one file format to another. Please also check the *poio_converter.py* script for example code.

The basic idea is that each Writer class in Poio API is responsible to check for and convert from tags that are part of the file format of that Writer class. For this, each writer can provide a method missing_tags() that will write a JSON mapping file. The following code parses a Toolbox file into a GrAF annotation graph, creates a Typecraft writer and calls missing_tags() with the output file name and the annotation graph as arguments:

```
from poioapi.annotationgraph import AnnotationGraph
import poioapi.io.typecraft

ag = AnnotationGraph.from_toolbox("toolboxfile.txt")
    typecraft = poioapi.io.typecraft.Writer()
     typecraft.missing_tags("maping.json", ag)
```

To apply the mapping file when writing the file you pass an additional argument extra_tag_map to the write() method of the writer:

```
typecraft.write("typecraftfile.xml", ag,
    extra_tag_map="mapping.json", language="your_iso_code")
```

1.2.6 Internals: The mapping classes

The TierMapper class

To generalize conversion and annotation mapping in Poio API we define a fixed set of tier types. Each of the tier types has then one or more names in each specific file format. This allows the conversion to work with the fixed set of tier types, the converter does not have to handle all the different tier names that might be used in the different file formats. The class poioapi.mapper.TierMapper is responsible for the mapping between tier types and tier names. A tier type might be linked to an ISOcat category, as soon as there is an agreement about tier types within the linguistic community.

To support the files formats and corpora that we encountered so far we defined the following tier types in Poio API:

- utterance
- word
- morpheme
- part of speech
- gloss
- graid1
- graid2
- translation
- comment

This list might look kind of ad hoc, and in fact it is the result of the use cases we had so far. If there is any requirement for new tier types we can easily add new types to this list. Compare the tier types that we listed here to the names in the JSON mapping files: the *tier_names* dictionary uses exactly these tier types as keys. In fact, we add any user defined tier names from the JSON files to the TierMapper object that we use during conversion.

Beside the user defined tier names there is a set of default tier names for each file format. In the case of Toolbox we pre-defined the following tier names in the module poioapi.io.toolbox:

- utterance: utterance_gen
- word: tx, t
- morpheme: mb, m
- part of speech: ps, p
- gloss: ge, g
- translation: ft, f
- comment: nt

The user can easily add and modify this list via JSON mapping files, as described above.

The AnnotationMapper class

The AnnotationMapper is used by the Writer classes to map the annotation labels. It is also responsible to validate tags, i.e. to check if the annotation label is part of the tagset of the given file format. Internally, the class uses a dictionary to map the annotations. This dictionary is created from a default JSON file for each combination of input file format and output file format. For example, there is a default JSON file for the mapping of annotations from Toolbox files to Typecraft files:

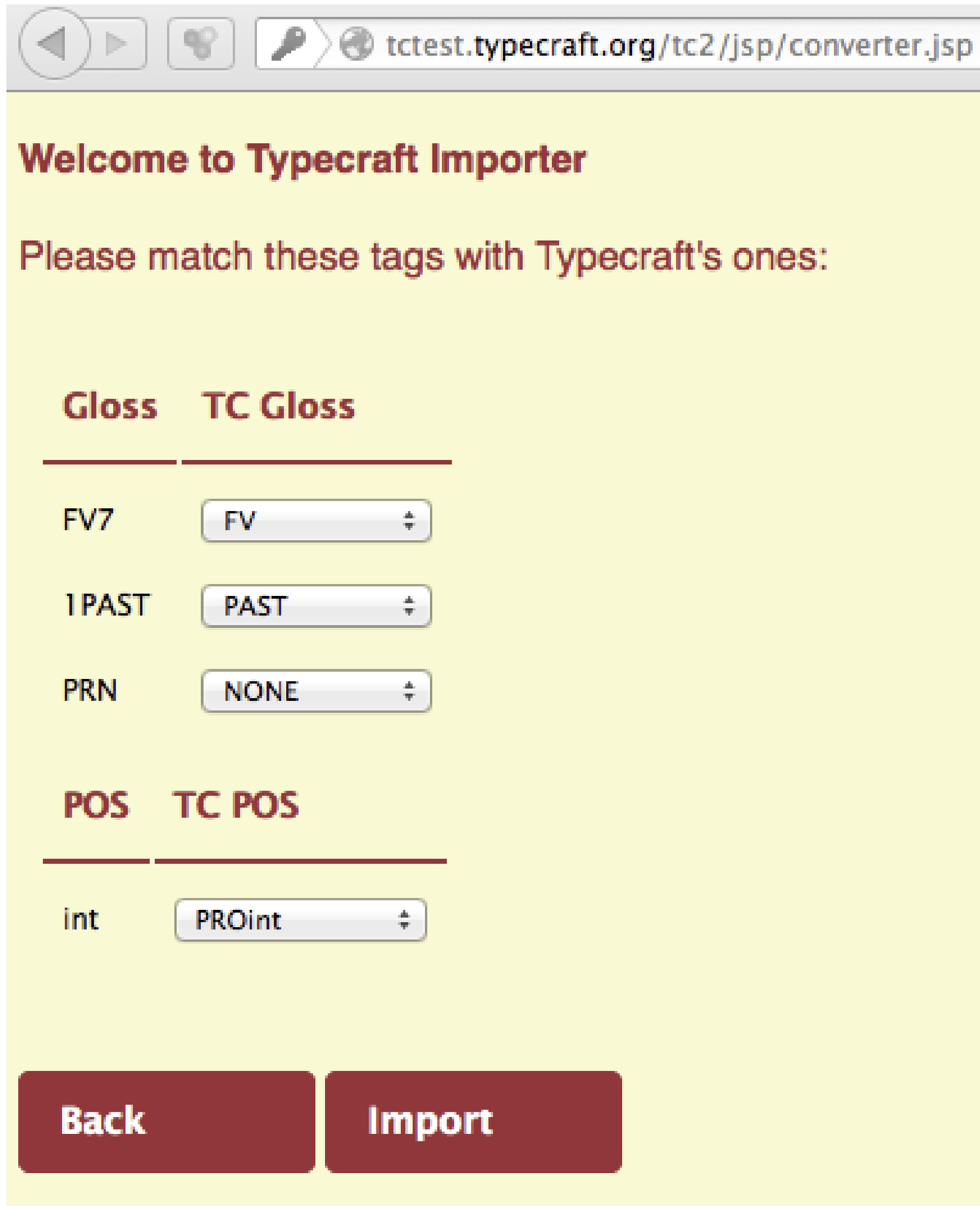
https://github.com/cidles/poio-api/blob/master/src/poioapi/mappings/TOOLBOX_TYPECRAFT.json

Additionally, the programmer can add more JSON files to an AnnotationMapper object to update the mapping. This is done by the script poio_converter.py when the user specified a mapping file on the command line, as described above.

1.2.7 Application: Toolbox import in the Typecraft web application

An example application of the full Poio API conversion functionality is the import of Toolbox files in the Tpyecraft web application. Internally, Typecraft uses Poio API to convert a Toolbox file into the Typecraft XML format and then imports this XML. This modularization of the import allows us to support other file formats in the future, for example the import of pure text-based IGT from Word files. All we need to modify is the conversion workflow in Poio API, the web application practically only needs a new entry in a dropdown so that the user can specify the input file format.

As the workflow in Poio API consists of three steps we can also allow the user to edit the annotation mapping in between. In the first step, we generate an JSON mapping file for any missing tags in the default mapping. Based on this JSON file we generate a user interface that allows the user to specify additional mappings:



Based the user input we generate a new JSON mapping file and add that file to the conversion when we execute the final conversion step to generate the Typecraft XML. The abstraction in Poio API allows us to use the same workflow for all file formats that are supported by Poio API.

1.3 Parser and Writer classes to map from and to file formats

This chapter explains how the Parser and Writer classes in Poio API work. You will learn how to write your own parsers and writers to support a custom file format. Poio API already support a lot of file formats out of the box, which are explained in the following sections. In any case the parser class is used by a general *Converter* class to map the file format onto a GrAF object. The user may then modify the GrAF object and write back the changes to any of the supported file format (or a custom format, if you implemented a writer). The following Python code demonstrates how one file format can be convert to another one with support of an existing parser and writer class:

```
parser = poioapi.io.wikipedia_extractor.Parser("Wikipedia.xml")
writer = poioapi.io.graf.Writer()

converter = poioapi.io.graf.GrAFConverter(parser, writer)
converter.parse()
converter.write("Wikipedia.hdr")
```

This code parses from the XML output of the Wikipedia Extractor and writes the content as GrAF files.

Contents

1.3.1 How to write a Parser/Writer for a new file format

In order to support your own file format in Poio API, you would need to implement your own parser as a sub-class of the base class `poioapi.io.graf.BaseParser`. The base class contains six abstract methods that will allow the GrAF converter to build a GrAF object from the content of your files. The six methods are:

- get_root_tiers() - Get the root tiers.
- get_child_tiers_for_tier(tier) - Get the child tiers of a given tier.
- get_annotations_for_tier(tier, annotation_parent) - Get the annotations on a given tier.
- tier_has_regions(tier) - Check if the annotations on a given tier specify regions.
- region_for_annotation(annotation) - Get the region for a given annotation.
- get_primary_data() - Get the primary data that the annotations refer to.

Note: All the methods must be implemented, otherwise an exception will be raised.

The tiers and annotations that are passed to the methods are normally objects from the classes `poioapi.io.graf.Tier` and `poioapi.io.graf.Annotation`. If you need to pass additional information between the methods, that are not present in our implementation of the classes, you might also sub-class `Tier` and/or `Annotation` and add your own properties. **By sub-classing, you make sure that the properties from our implementation are still there. The converter needs them to build the GrAF object.**

Each `Tier` contains a *name* and an *annotation_space* property (the latter is *None* by default). The class `ElanTier` exemplifies the sub-classing of *Tier*. In the case of Elan, we need to store an additional property *linguistic_type* to be able to implement the complete parser:

```
class ElanTier(poioapi.io.graf.Tier):
    __slots__ = ["linguistic_type"]

    def __init__(self, name, linguistic_type):
        self.name = name
        self.linguistic_type = linguistic_type
        self.annotation_space = linguistic_type
```

`Tier` s use the *annotation_space* to describe that they share certain annotation types. If the *annotation_space* is *None* the GrAF converter will use the *name* as the label for the annotation space.

Each `Annotation` is defined with a unique *id* property and can contain a *value* and a ‘ features‘ property. Features are stored in a dictionary in the *feature_structure* of the annotation in the GrAF representation.

References:

- `poioapi.io.graf.BaseParser`
- `poioapi.io.graf.Tier`
- `poioapi.io.graf.Annotation`

Example: A simple parser based on static data

The transformation of annotation data to GrAF is done by the class `poioapi.io.graf.GrAFConverter`. This class will use the parser’s methods to retrieve the information from the file.

Sub-classing from BaseParser

First, we will sub-class our own parser `SimpleParser` from the class `poioapi.io.graf.BaseParser` with empty methods. We will set some static data within the class that represent our tier names and the annotations for each tier:

```
class SimpleParser(poioapi.io.graf.BaseParser):

    tiers = ["utterance", "word", "wfw", "graid"]
    utterance_tier = ["This is a utterance", "that is another utterance"]
    word_tier = [['This', 'is', 'a', 'utterance'], ['that', 'is', 'another',
                 'utterance']]
    wfw_tier = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
    graid_tier = ['i', 'j', 'k', 'l', 'm', 'n', 'o', 'p']

    def __init__(self):
        pass

    def get_root_tiers(self):
        pass

    def get_child_tiers_for_tier(self, tier):
        pass

    def get_annotations_for_tier(self, tier, annotation_parent=None):
        pass

    def tier_has_regions(self, tier):
        pass

    def region_for_annotation(self, annotation):
        pass

    def get_primary_data(self):
        pass
```

If your annotations are stored in a file, then you need to implement your own strategy how to load the file’s content into your parser class. The `__init__()` of your parser class might be a good place to load your file.

References:

- `poioapi.io.graf.GrAFConverter`

Implementation of the parser methods

We will start with the `get_root_tiers()` method. This method will return all the root tiers as objects of the class `Tier` (or a sub-class of it). In our case, this is only the utterance tier:

```
def get_root_tiers(self):
    return [poioapi.io.graf.Tier("utterance")]
```

The method `get_child_tiers_for_tier()` returns all child tiers of a given tier, again as `Tier` objects. In our simple example, we assume that the child of the *utterance* tier is the *word* tier, which has the children *graid* and *wfw*:

```
def get_child_tiers_for_tier(self, tier):
    if tier.name == "utterance":
        return [poioapi.io.graf.Tier("word")]
    if tier.name == "word":
        return [poioapi.io.graf.Tier("graid"), poioapi.io.graf.Tier("wfw")]

    return None
```

Note: This two methods must always return a list of `Tier` objects or *None*.

The method `get_annotations_for_tier()` is used to collect the annotations for a given tier. Each annotation must at least cotain a unique *id* and an annotation *value*. Both properties are already present in the class `Annotation` that we use here to return the annotations. For the utterance tier we can simply convert the list of strings in our *self.utterance_tier* data store:

```
def get_annotations_for_tier(self, tier, annotation_parent=None):
    if tier.name == "utterance":
        return [poioapi.io.graf.Annotation(i, v)
                    for i, v in enumerate(self.utterance_tier)]

        [...]
```

For all tiers that are children of another tier, the annotations within the tiers are normally also children of another annotation on the parent tier. In this case the `Converter` will pass a value in the parameter *annotation_parent*. In our case, the *id* of the parent annotation points to the location of the child annotations in the lists *self.word_tier*, *self.graid_tier* and *self.wfw_tier*:

```
[...]

    if tier.name == "word":
        return [poioapi.io.graf.Annotation(2 + 4 * annotation_parent.id + i, v) for i,
↪ v
                    in enumerate(self.word_tier[annotation_parent.id])]

    if tier.name == "graid":
        return [poioapi.io.graf.Annotation(
            annotation_parent.id + 10, self.graid_tier[annotation_parent.id - 2])]

    if tier.name == "wfw":
        return [poioapi.io.graf.Annotation(
            annotation_parent.id + 12, self.wfw_tier[annotation_parent.id - 2])]
```

(continues on next page)

(continued from previous page)

```
    return []
```

Note: This method must always return a list with `Annotation` elements or an empty list.

The method `tier_has_regions()` describes which tiers contain regions. These regions are intervals that refer to the primary data. Depending on the type of the primary data the regions can encode intervals of time (encoded as milliseconds, in most cases) or a range in a string (from start to end position). In our case we assume that only the root tier *utterance* is connected to the primary data via regions:

```
def tier_has_regions(self, tier):

    if tier.name == "utterance":
        return True

    return False
```

To get the regions of a specific annotation the `Converter` will call the method `region_for_annotation()`. This method must return a tuple with start and end of the regions. In our example the tier with regions is the utterance tier. So the region for the first utterance is `(0, 19)`, if we assume that we want to return the content of the two utterances connected with a blank " " as the primary data. We can simply calculate the regions from the length of the strings in `self.utterance_tier`:

```
def region_for_annotation(self, annotation):

    if annotation.id == 0:
        return (0, len(self.utterance_tier[0]))
    elif annotation.id == 1:
        return (len(self.utterance_tier[0]) + 1,
                len(self.utterance_tier[0]) + 1 + len(self.utterance_tier[1]))
```

Last but not least, we also have to return the primary data. As the utterance tier was the root tier and we already defined the regions for the utterance annotations based on the strings in `self.utterance_tier` we can simply join the two strings and return the result as the primary data:

```
def get_primary_data(self):
    return ' '.join(self.utterance_tier)
```

Using the parser to convert to GrAF

You can now use the `SimpleParser` class to convert the static data into a GrAF object:

```
parser = SimpleParser()

converter = poioapi.io.graf.GrAFConverter(parser)
converter.parse()

graf = converter.graf
```

The *converter* object contains two more objects that contain information from the parsed data:

- The tier hierarchies is stored in *converter.tier_hierarchies*.
- The primary data for the annotations is stored in *converter.primary_data*.

If you want to write the data to GrAF files, you have to create a GrAF writer object and pass it to the *Converter*'s constructor:

```
parser = SimpleParser()
writer = poioapi.io.graf.Writer()

converter = poioapi.io.graf.GrAFConverter(parser, writer)
converter.parse()
converter.write("simple.hdr")
```

The section *Spreadsheet to GrAF conversion* discusses a slightly more complex use case: how to write a parser for custom annotations stored in a Microsoft Excel file.

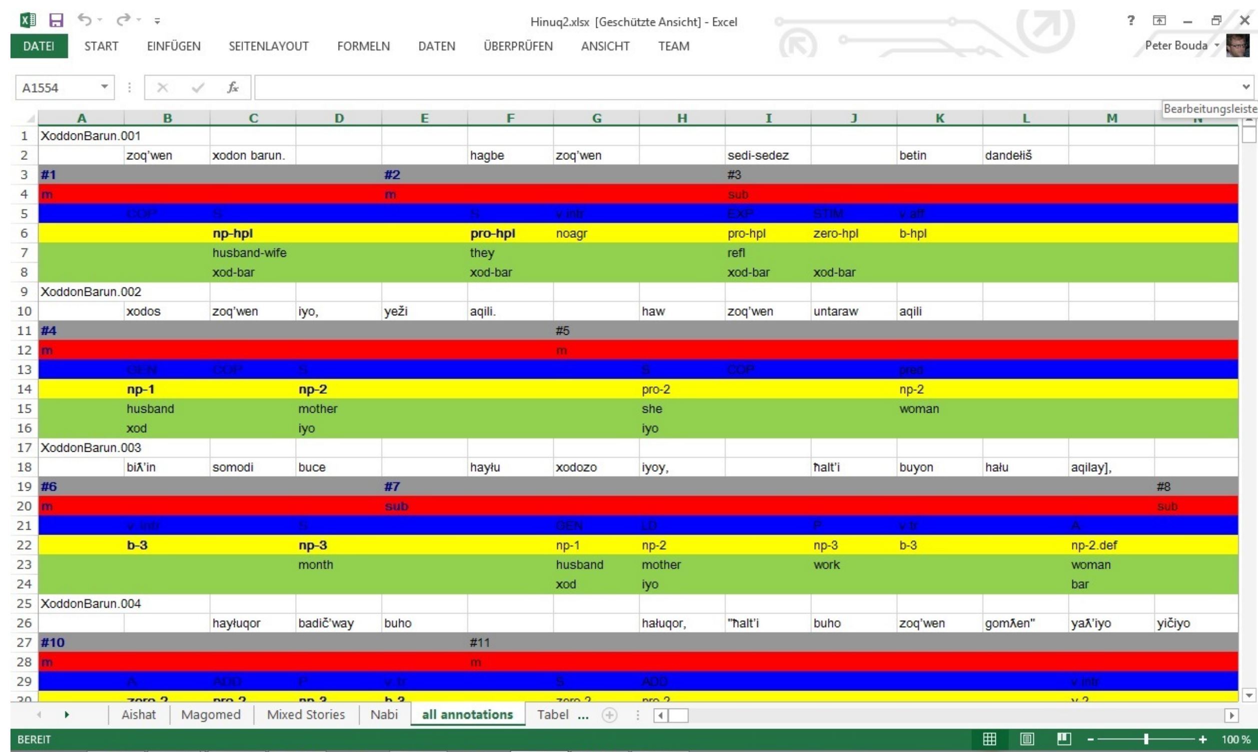
1.3.2 Spreadsheet to GrAF conversion

The section *How to write a Parser/Writer for a new file format* described how the general conversion mechanism works in Poio API, and how you can implement your own parser to convert a custom file format to a GrAF object. In this section we will continue with a more complex example based on annotations in Microsoft Excel or LibreOffice/OpenOffice Calc. We will show how can export Excel data into a CSV file and discuss a CSV parser that we will finally use to convert the Excel annotations into GrAF-XML files.

The data in this section comes from real-world language documentation project about north-east caucasian languages. We will use several annotated texts in the language Hinuq. The Excel file was kindly provided by Diana Forker.

The data in Excel

The data in Excel consists of several "tiers" that were encoded as rows in an Excel worksheet. For each utterance of the original text there are eight rows in the Excel sheet. Here is a screenshot of Excel with that shows the first three utterances:

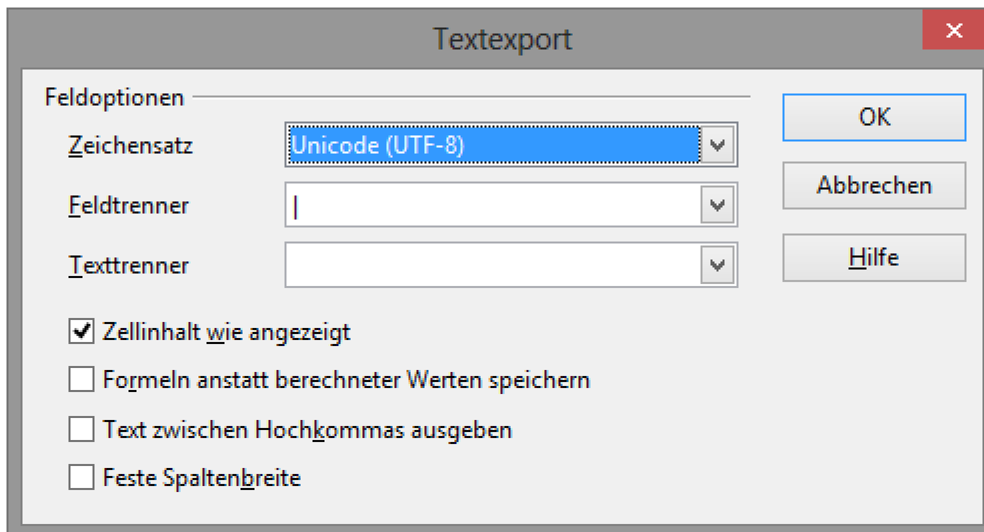


The first row, for example, contains a unique ID, while the second row consists of the tokenized utterance ("word" tier), with optional spaces between them. Row three and four contain an ID and an annotation for the so-called "clause unit", a term stemming from the annotation framework GRAID (Grammatical Relations and Animacy in Discourse), developed by Geoffrey Haig and Stefan Schnell. The subsequent four rows contain GRAID annotations, custom annotations and translations based on the word tier.

In this case Diana was interested in the different word orders that were used in the Hinuq texts. Word order appear within the "clause units", as those represent parts of utterances like main clauses and sub clauses. To analyze the word order now, the interesting units within the clause units are the participants and the verb. Participants of clauses are normally encoded regarding the syntactic/semantic role within the clause, Diana used the labels "S", "A" and "P" that are widely used within general linguistics and language typology. The verbs have different tags like "v.tr", "v.intr" or "v.aff". All the interesting tags appear in row five in Excel. Because we are only interested in clause units and the tags that represent participant and verbs within each clause unit, our parser will only return informations from row three, four and five of each utterance. It should be easy enough to extend the parser to more rows later. The names that those three rows will be *clause_id*, *clause_type* and *grammatical_relation*, containing the IDs of the clause units, the type of the clauses (main or sub class) and the grammatical relations as discussed, respectively. These are the *tiers* that our parser will process and for which it will return the annotations from the Excel CSV file.

Export the data

The first step is to export the data from Excel to a CSV file. CSV files are much easier to read in with Python. Unfortunately, Microsoft Excel still has a big problem when it comes to export Unicode CSV files. In the case of the Hinuq data, Diana used a lot of different Unicode characters that we need to preserve when exporting. We thus used Open Office to export the data to a CSV file that uses a UTF-8 encoding. You can open your Excel file in Open Office, click on *File* → *Save As* and then choose *Text CSV* as file format. In the following dialog choose *UTF-8* as encoding and the pipe symbol "|" as field seperator. We also chose an empty string as text seperator:



In the following steps we assume that the filename is *Hinuq.csv* and that the file was saved with those settings.

The implementation of the parser

The easiest way to implement a parser for Poio API is to sub-class from `poioapi.io.graf.BaseParser` as described in section *How to write a Parser/Writer for a new file format*. We have to implement six abstract methods so that the `poioapi.io.graf.GrAFConverter` class can then build a GrAF from the CSV data. The six methods are:

- get_root_tiers() - Get the root tiers.

- get_child_tiers_for_tier(tier) - Get the child tiers of a give tier.
- get_annotations_for_tier(tier, annotation_parent) - Get the annotations on a given tier.
- tier_has_regions(tier) - Check if the annotations on a given tier specify regions.
- region_for_annotation(annotation) - Get the region for a given annotation.
- get_primary_data() - Get the primary data that the annotations refer to.

First, we will implement the constructor of our new parser class `ExcelParser`. The constructor does most of the work in our class, as it is responsible to parse the CSV file and put all the interesting information in Python data structures. This is possible here, because the CSV file does not contain so much data and we can still store everything in memory. If your data is too big you may implement a more sophisticated method to stream the data while the converter is calling the methods. Our full constructor looks like this:

```
import csv
import codecs

import poioapi.io.graf
import poioapi.annotationgraph
import poioapi.data

class ExcelParser(poioapi.io.graf.BaseParser):

    def __init__(self, filepath):
        self.word_orders = dict()
        self.clauses = list()
        self.clause_types = dict()
        self.last_id = -1
        with codecs.open(filepath, "r", "utf-8") as csvfile:
            hinuq2 = csv.reader(csvfile, delimiter='|')
            i = 0
            for row in hinuq2:
                if i == 2:
                    clause_ids = row
                elif i == 3:
                    clause_types = row
                elif i == 4:
                    grammatical_relations = row
                i += 1
                if i > 7:
                    # now parse
                    word_order = []
                    c_id = None
                    prev_c_id = None
                    for j, clause_id in enumerate(clause_ids):

                        # new clause
                        if clause_id != "":
                            # add word order to previous clause
                            if len(word_order) > 0:
                                self.word_orders[c_id] = word_order
                            word_order = []

                            # add new clause
                            c_id = self._next_id()
                            self.clauses.append(c_id)
                            self.clause_types[c_id] = clause_types[j].strip()
```

(continues on next page)

(continued from previous page)

```
                    grammatical_relation = grammatical_relations[j].strip()
                    word_order.append(grammatical_relation)

                if len(word_order) > 0:
                    self.word_orders[c_id] = word_order
                i = 0
```

The important data structures here are the three properties *self.clauses*, *self.clause_types* and *self.word_orders*. The first is a list of IDs, while the latter two a dictionaries with the clause IDs as keys. The store the annotations (clause type and grammatical relations from row four and five of the Excel file) for each clause unit as values.

The six abstract methods of the base class are then easy to implement, we will just list them as a big block of code here:

```
def _next_id(self):
    self.last_id += 1
    return self.last_id

def get_root_tiers(self):
    return [poioapi.io.graf.Tier("clause_id")]

def get_child_tiers_for_tier(self, tier):
    if tier.name == "clause_id":
        return [poioapi.io.graf.Tier("grammatical_relation"),
                poioapi.io.graf.Tier("clause_type")]

    return None

def get_annotations_for_tier(self, tier, annotation_parent=None):
    if tier.name == "clause_id":
        return [poioapi.io.graf.Annotation(i, v)
            for i, v in enumerate(self.clauses)]

    elif tier.name == "clause_type":
        return [poioapi.io.graf.Annotation(
            self._next_id(), self.clause_types[annotation_parent.id])]

    elif tier.name == "grammatical_relation":
        return [poioapi.io.graf.Annotation(self._next_id(), v)
            for v in self.word_orders[annotation_parent.id]]

    return []

def tier_has_regions(self, tier):
    return False

def region_for_annotation(self, annotation):
    pass

def get_primary_data(self):
    pass
```

The tier hierarchy is simple, we have the root tier *clause_id* and two child tiers *grammatical_relation* and *clause_type*. The two methods `get_root_tiers()` and `get_child_tiers_of_tier()` implement this hierarchy. The next method `get_annotations_for_tier()` returns the contents of the different tiers as `Annotation` objects. We just have to make sure that all the IDs are unique, which is the responsibilty of the method `_next_id()`.

The last three methods can stay empty, as there are no regions on any tier and did not access the primary data in the Excel file. Based on this parser class we can now write a simple converter for our type of CSV files, as demonstrated in the next section.

How to use the parser to convert to GrAF-XML

Next we want to implement a helper function that creates an `AnnotationGraph` object from an Excel file, which we will then use to analyze the word orders in the Hinuq texts. We can simply create a parser object from our new class `ExcelParser` and pass it to the `poioapi.io.graf.GrAFConverter` class. After parsing, we have access to the GrAF object and the tier hierarchy through the converter object. We need to copy these objects into the `AnnotationGraph` object to be ableto use some of the methods of the `AnnotationGraph` later when we analyze the word order. The full code of our helper method is:

```
def from_excel(filepath):
    ag = poioapi.annotationgraph.AnnotationGraph()
    parser = ExcelParser(filepath)
    converter = poioapi.io.graf.GrAFConverter(parser)
    converter.parse()
    ag.tier_hierarchies = converter.tier_hierarchies
    ag.structure_type_handler = poioapi.data.DataStructureType(
        ag.tier_hierarchies[0])
    ag.graf = converter.graf
    return ag
```

With this preparation we can now follow up with the analysis of word order in the Excel file. This analysis is part of a separate IPython notebook that you can view and download here:

http://nbviewer.ipython.org/urls/raw.github.com/pbouda/notebooks/master/Diana%20Hinuq%20Word%20Order.ipynb

The first block of code in the notebook loads a file *helper/diana.py*, which contains exactly the class `ExcelParser` and the helper function `from_excel()` from above. You can download the helper file here:

https://raw.github.com/pbouda/notebooks/master/helpers/diana.py

1.4 Linguistic analysis and pipelines based on GrAF graphs

We think that GrAF graphs can play an important role in the implementation of scientific workflows in linguistics. Based on the GrAF objects that Poio API generates you might pipe the data to scientific Python libraries like networkx, numpy or scipy. The American National Corpus implemented connectors for GrAF and two linguistic frameworks. The conversion of custom file formats to GrAF through Poio API can thus act as an entry point to those pipelines and support to merge data and annotation from a wide range of heteregenous data sources for further analysis.

1.4.1 Search in annotation graphs: filters and filter chains

The **filter** class `poioapi.annotationgraph.AnnotationGraphFilter` can be used to search in annotation graphs in Poio API. The filter class can only be used together with the annotation graph class `poioapi.annotationgraph.AnnotationGraph`. The idea is that each annotation graph can contain a set of filters, that each reduce the full annotation graph to a subset. This list of filters is what we call a **filter chain**. Each filter consists of search terms for each of the tiers that were loaded from an input file, as described in section *Data Structure Types*. The search terms can be simple strings or regular expressions.

To be able to apply a filter to an annotation graph you have to load some data first. In this example we will use the example file from the Elan homepage. First, we create a new annotation graph and load the file:

```
import poioapi.annotationgraph

ag = poioapi.annotationgraph.AnnotationGraph()
ag.from_elan("elan-example3.eaf")
```

In the next step we set the default tier hierarchy for the annotation graph. As the example file contains four root tiers with subtiers we have to choose one of the hierarchies carefully. In our case we choose the hierarchy with the root tier *utterance..W-Spch* that we find at index *1* of the property *ag.tier_hierarchies* after we loaded the file. We choose this tier hierchary to be used for all subsequent filter operations:

```
ag.structure_type_handler = \
    poioapi.data.DataStructureType(ag.tier_hierarchies[1])
```

In our case the hierarchy *ag.tier_hierarchies[1]* contains the following tiers:

```
['utterance..W-Spch',
    ['words..W-Words',
        ['part_of_speech..W-POS']],
    ['phonetic_transcription..W-IPA']]
```

Now we are ready to create a filter for the data. We will filter the data with serch terms on two of the subtiers of our tier hierarchy: we will search for `follow` on the *words* tier and for the regular expression `\bpro\b` on the *POS* tier. We can look up the full names of the tiers in the above tier hierarchy. The following code creates a filter object and adds the two search terms for the two tiers:

```
af = poioapi.annotationgraph.AnnotationGraphFilter(ag)
af.set_filter_for_tier("words..W-Words", "follow")
af.set_filter_for_tier("part_of_speech..W-POS", r"\bpro\b")
```

The final step is to append the filter to the filter chain of the annotation graph:

```
ag.append_filter(af)
```

The append operation will already start the process of graph filtering. The result is stored in the property *filtered_node_ids* of the annotation graph object, which is a list of root nodes where child nodes matched the search term:

```
print(ag.filtered_node_ids)
[['utterance..W-Spch..na10',
  'utterance..W-Spch..na12',
  'utterance..W-Spch..na19']]
```

You can get a visible result set by writing a filtered HTML representation of the annotation graph:

```
import codecs
html = ag.as_html_table(True)
f = codecs.open("filtered.html", "w", "utf-8")
f.write(html)
f.close()
```

You can add more filters to the annotation graph by creating more filter objects and passing them to *append_filter()*. If you want to remove a filter you can call *pop_filter()*, which will remove the filter that was last added to the annotation graph object:

```
ag.pop_filter()
```

A convenient way to create filter objects is by passing a dictionary with tier names and search terms to the method *create_filter_for_dict()* of the annotation graph object. The following code will create the same filter as in the example above:

```
search_terms = {
    "words..W-Words": "follow",
    "part_of_speech..W-POS": r"\bpro\b"
}
af = ag.create_filter_for_dict(search_terms)
```

You can then append the filter to the filter chain. A complete script that demonstrates filters and filter chains is available on Github:

https://github.com/cidles/poio-api/blob/master/examples/filter.py

1.4.2 Real world examples

Counting word orders

The following example is based on the parser explained in section *Spreadsheet to GrAF conversion*. The whole workflow to count word order in GrAF is implemented as IPython notebook, which you can view and download here:

http://nbviewer.ipython.org/urls/raw.github.com/pbouda/notebooks/master/Diana%20Hinuq%20Word%20Order.ipynb

D3.js for visualization

The graf-python documentation contains a nice example how to visualize GrAF data with the help of the networkx library and the Javascript visualization library D3.js:

https://graf-python.readthedocs.org/en/latest/Translation%20Graph%20from%20GrAF.html

To just see the example visualization click here:

http://bl.ocks.org/anonymous/4250342

1.4.3 GrAF connectors

The American National Corpus implemented GrAF connectors for the Unstructured Information Management applications (Apache UIMA) fraemwork and the general architecture for text engineering (GATE) software. You can download the ANC software here:

- http://www.anc.org/software/uimautils/
- http://www.anc.org/software/gate-tools/

CHAPTER 2

API documentation

2.1 PoioAPI Package

2.1.1 poioapi.data

This module contains the classes to access annotated data in various formats.

The parsing is done by Builder classes for each file type, i.e. Elan's .eaf files, Kura's .xml file, Toolbox's .txt files etc.

class poioapi.data.**DataStructureType**(*custom_data_hierarchy=None*)

Data structure type constructor.

Attributes

'name' [str] Name of the structure.

data_hirerarchy [array] Structure of the array.

Methods

empty_element()	Return the appended list of a certain data hierarchy.
get_children_of_type(ann_type)	Returns all the elements that are above a given type in the type hierarchy.
get_parents_of_type(ann_type)	Returns all the elements that are above a given type in the type hierarchy.
type_has_region(ann_type)	Checks whether the given type has regions that connect it to the base data.

__init__(*custom_data_hierarchy=None*)

Class's constructor.....

empty_element()

Return the appended list of a certain data hierarchy.

Returns

_append_list [array_like] The actual list with the appended elements.

get_children_of_type(*ann_type*)

Returns all the elements that are above a given type in the type hierarchy.

Parameters

ann_type [str] Value of the field in the data structure hierarchy.

Returns

_get_parents_of_type_helper [array_like] The return result depends on the return of the called method.

See also:

`_get_parents_of_type_helper`

get_parents_of_type(*ann_type*)

Returns all the elements that are above a given type in the type hierarchy.

Parameters

ann_type [str] Value of the field in the data structure hierarchy.

Returns

_get_parents_of_type_helper [array_like] The return result depends on the return of the called method.

See also:

`_get_parents_of_type_helper`

type_has_region(*ann_type*)

Checks whether the given type has regions that connect it to the base data.

Parameters

ann_type [str] Value of the field in the data structure hierarchy.

Returns

is_region [bool] Whether the annotation type has regions.

class poioapi.data.**DataStructureTypeGraid**(*custom_data_hierarchy=None*)

Data structure type using a GRAID format.

Attributes

'name' [str] Name of the structure.

data_hirerarchy [array_like] Structure of the array.

Methods

`empty_element`()	Return the appended list of a certain data hierarchy.
`get_children_of_type`(ann_type)	Returns all the elements that are above a given type in the type hierarchy.
`get_parents_of_type`(ann_type)	Returns all the elements that are above a given type in the type hierarchy.

Continued on next page

Table 2 – continued from previous page

type_has_region(ann_type)	Checks whether the given type has regions that connect it to the base data.

class poioapi.data.**DataStructureTypeGraidDiana**(*custom_data_hierarchy=None*)

Data structure type using a GRAID format.

Attributes

'name' [str] Name of the structure.

data_hirerarchy [array_like] Structure of the array.

Methods

empty_element()	Return the appended list of a certain data hierarchy.
get_children_of_type(ann_type)	Returns all the elements that are above a given type in the type hierarchy.
get_parents_of_type(ann_type)	Returns all the elements that are above a given type in the type hierarchy.
type_has_region(ann_type)	Checks whether the given type has regions that connect it to the base data.

class poioapi.data.**DataStructureTypeMorphsynt**(*custom_data_hierarchy=None*)

Data structure type using a Morphsyntax format.

Attributes

'name' [str] Name of the structure.

data_hirerarchy [array_like] Structure of the array.

Methods

empty_element()	Return the appended list of a certain data hierarchy.
get_children_of_type(ann_type)	Returns all the elements that are above a given type in the type hierarchy.
get_parents_of_type(ann_type)	Returns all the elements that are above a given type in the type hierarchy.
type_has_region(ann_type)	Checks whether the given type has regions that connect it to the base data.

exception poioapi.data.**DataStructureTypeNotCompatible**

exception poioapi.data.**DataStructureTypeNotSupportedError**

exception poioapi.data.**NoFileSpecifiedError**

exception poioapi.data.**UnknownAnnotationTypeError**

exception poioapi.data.**UnknownDataStructureTypeError**

exception poioapi.data.**UnknownFileFormatError**

2.1.2 poioapi.annotationtree

2.1.3 poioapi.annotationgraph

2.2 PoioAPI IO Package

2.2.1 poioapi.io.elan

2.2.2 poioapi.io.graf

2.2.3 poioapi.io.pickle

Indices and tables

- genindex
- modindex
- search

2.2.4 poioapi.io.typecraft

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

p

poioapi.data, 25

Index

Symbols

__init__() (poioapi.data.DataStructureType method), 25

D

DataStructureType (class in poioapi.data), 25
DataStructureTypeGraid (class in poioapi.data), 26
DataStructureTypeGraidDiana (class in poioapi.data), 27
DataStructureTypeMorphsynt (class in poioapi.data), 27
DataStructureTypeNotCompatible, 27
DataStructureTypeNotSupportedError, 27

E

empty_element() (poioapi.data.DataStructureType method), 25

G

get_children_of_type() (poioapi.data.DataStructureType method), 26
get_parents_of_type() (poioapi.data.DataStructureType method), 26

N

NoFileSpecifiedError, 27

P

poioapi.data (module), 25

T

type_has_region() (poioapi.data.DataStructureType method), 26

U

UnknownAnnotationTypeError, 27
UnknownDataStructureTypeError, 27
UnknownFileFormatError, 27