CSVW Converters Documentation

Release 1.0.0

Rinke Hoekstra

Nov 16, 2020

Contents

This package is a comprehensive tool (CoW[1]) for batch conversion of multiple datasets expressed in CSV. It uses a JSON schema expressed using an extended version of the CSVW standard, to convert CSV files to RDF in scalable fashion.

Instead of using the command line tool there is also the webservice cattle, providing the same functionality that CoW provides without having to install it. CSV files can be uploaded to the service and a JSON schema will be created, using that JSON schema cattle is able to create a RDF structured graph. More information about cattle, including how to use it, can be found at: https://github.com/CLARIAH/cattle.

CSV on the Web (CSVW) is a W3C standard for metadata descriptions for tabular data. Typically, these data reside in CSV files. CSVW metadata is captured in `.csv-metadata.json` files that live alongside the CSV files that they describe. For instance, a CSV file called `data.csv` and its metadata `data.csv-metadata.json` would be hosted at:

```
http://example.com/data.csv
http://example.com/data.csv-metadata.json
```

Another feature of CSVW is that it allows the specification of a mapping (or interpretation) of values in the CSV in terms of RDF. The `tableSchema` element in CSVW files defines per column what its properties should be, but may also define custom mappings to e.g. URIs in RDF.

Interestingly, the JSON format used by CSVW metadata is an extension of the JSON-LD specification, a JSON-based serialization for Linked Data. As a consequence of this, the CSVW metadata can be directly attached (as provenance) to the RDF resulting from a CSVW-based conversion.

This is exactly what the CoW converter does.

The rest of this documentation will be fairly technical, for some hands-on examples you can take a look at the Wiki.

[1] *COW*: **C**SV **O**n the **W**eb.

CHAPTER 1

Features & Limitations

Compared to the CSVW specification, the converter has a number of limitations and extra features. These are:

1. CoW *does not* perform any schema checking, and ignores any and all parts of the CSVW Specification that are not directly needed for the RDF conversion.
2. CoW extends the CSVW specification in several ways:
   - Advanced formatting of URLs and values
   - Dealing with multiple null values and null values for one or more other columns.
   - Simple SKOS support (generating collections and schemes)
   - Optionally skipping/not skipping empty cells
   - A default set of namespace prefixes
3. CoW does some smart guessing:
   - Determining file encoding
   - Determining the delimiter
   - Generating a skeleton schema for any CSV file (see *here*)
4. CoW produces extensive provenance:
   - Converted data is encapsulated in a Nanopublication
   - The original CSVW schema is encapsulated in the *np:hasProvenance* graph associated with the nanopublication.

CHAPTER 2

Installation

2.1 Prerequisites

- Python 3.8 (installed on most systems)
- `pip3`
- `virtualenv` (simply *pip3 install virtualenv*)[2]

2.2 Installing with pip (preferred)

Open up a terminal (or Command Prompt when you are using Windows) and instantiate a virtual Python environment:

```
virtualenv .
```

Activate the virtual environment:

```
source bin/activate
```

Install CoW in the new environment:

```
pip3 install cow_csvw
```

To upgrade a previously installed version of CoW, do:

```
pip3 install --upgrade cow_csvw
```

(you might need permissions if you're installing outside a virtualenv). To check the version currently installed:

```
cow_tool --version
```

To get help:

[2] These instructions use `virtualenv` but you can also install all packages globally, or use an alternative such as `conda`.

```
cow_tool
```

CHAPTER 3

Usage

The primary command line script for CSVW-based conversion is `cow_tool`. It can be used for two tasks:

1. Generating a *skeleton CSVW JSON-Schema* for a specific CSV file.
2. Using such a schema to *convert a CSV file to RDF* (in NQuads format)

General usage instructions can be obtained by running `cow_tool -h`:

```
usage: cow_tool [-h] [--dataset DATASET] [--delimiter DELIMITER]
                [--quotechar QUOTECHAR] [--processes PROCESSES]
                [--chunksize CHUNKSIZE] [--base BASE]
                {convert,build} file [file ...]
```

The table below gives a brief description of each of these options.

Table 1: Commandline options for `cow_tool`

Option	Explanation
`dataset`	Specifies the name of the dataset, if it is different from the filename with the `.csv` extension stripped.
`delimiter`	Forces the use of a specific delimiter when parsing the CSV file (only used with `build` option)
`quotechar`	Forces the use of a specific quote character (default is `"`, only used with `build` option)
`encoding`	Forces the use of a specific file encoding when parsing the CSV file (only used with `build` option)
`processes`	Specifies the number of parallel processes to use when converting a CSV file (default is 4)
`chunksize`	Specifies the number of lines that will be passed to each process (default is 5000)
`base`	The base for URIs generated with the schema (only used with `build` option, the default is `http://data.socialhistory.org`)
`{convert, build}`	The `convert` option triggers a conversion to RDF for the files specified in `file [file ...]`. The `build` option generates a skeleton JSON schema for the files specified.
`file [file ...]`	A list of files to be converted (or "built"); any unix-style wildcards are allowed.

3.1 Generating a Skeleton Schema

Since JSON is a rather verbose language, and we currently do not have a convenient UI for constructing CSVW schema files, CoW allows you to generate a skeleton schema for any CSV file.

Suppose you want to build a skeleton schema for a file `imf_gdppc.csv` (from[4]) that looks like:

```
Rank;Country;GDP_Per_Capita
1;Qatar;131,063
2;Luxembourg;104,906
3;Macau;96,832
4;Singapore;90,249
5;Brunei Darussalam;83,513
6;Kuwait;72,675
7;Ireland;72,524
8;Norway;70,645
```

Make sure you have your virtual environment enabled (if applicable), and run:

```
cow_tool build imf_gdppc.csv --base=http://example.com/resource
```

The `--base` option specifies the base for all URIs generated through the schema. This is `https://iisg.amsterdam/` by default (see http://datalegend.net)

This will generate a file called `imf_gdppc.csv-metadata.json` with the following contents:

```
{
 "dialect": {
  "quoteChar": "\"",
  "delimiter": ";",
  "encoding": "ascii"
 },
 "dcat:keyword": [],
 "dc:license": {
  "@id": "http://opendefinition.org/licenses/cc-by/"
 },
 "dc:publisher": {
  "schema:name": "CLARIAH Structured Data Hub - Datalegend",
  "schema:url": {
   "@id": "http://datalegend.net"
  }
 },
 "url": "imf_gdppc.csv",
 "@context": [
  "http://csvw.clariah-sdh.eculture.labs.vu.nl/csvw.json",
  {
   "@base": "http://example.com/resource/",
   "@language": "en"
  },
  {
   "owl": "http://www.w3.org/2002/07/owl#",
   "napp-eng81": "https://iisg.amsterdam/napp/dataset/englandwales1881/",
   "dbo": "http://dbpedia.org/ontology/",
   "clioctr": "https://iisg.amsterdam/clio/country/",
   "hisclass": "https://iisg.amsterdam/hisclass/",
   "hisco-product": "https://iisg.amsterdam/hisco/product/",
```

(continues on next page)

[4] https://en.wikipedia.org/wiki/List_of_countries_by_GDP_%28PPP%29_per_capita

(continued from previous page)


```
    "ldp": "http://www.w3.org/ns/ldp#",
    "clio": "https://iisg.amsterdam/clio/",
    "occhisco": "https://iisg.amsterdam/napp/OCCHISCO/",
    "dbr": "http://dbpedia.org/resource/",
    "skos": "http://www.w3.org/2004/02/skos/core#",
    "xml": "http://www.w3.org/XML/1998/namespace/",
    "sdmx-concept": "http://purl.org/linked-data/sdmx/2009/concept#",
    "napp": "https://iisg.amsterdam/napp/",
    "prov": "http://www.w3.org/ns/prov#",
    "sdmx-code": "http://purl.org/linked-data/sdmx/2009/code#",
    "napp-can91": "https://iisg.amsterdam/napp/dataset/canada1891/",
    "hiscam": "https://iisg.amsterdam/hiscam/",
    "dbpedia": "http://dbpedia.org/resource/",
    "np": "http://www.nanopub.org/nschema#",
    "hisclass5": "https://iisg.amsterdam/hisclass5/",
    "canfam-auke": "https://iisg.amsterdam/canfam/auke/",
    "dcterms": "http://purl.org/dc/terms/",
    "schema": "http://schema.org/",
    "foaf": "http://xmlns.com/foaf/0.1/",
    "sdv": "http://example.com/resource/vocab/",
    "hisco": "https://iisg.amsterdam/hisco/",
    "bibo": "http://purl.org/ontology/bibo/",
    "sdmx-dimension": "http://purl.org/linked-data/sdmx/2009/dimension#",
    "hsn": "https://iisg.amsterdam/hsn2013a/",
    "dc": "http://purl.org/dc/terms/",
    "hisco-relation": "https://iisg.amsterdam/hisco/relation/",
    "hisco-status": "https://iisg.amsterdam/hisco/status/",
    "dbp": "http://dbpedia.org/property/",
    "clioprop": "https://iisg.amsterdam/clio/property/",
    "csvw": "http://www.w3.org/ns/csvw#",
    "clioind": "https://iisg.amsterdam/clio/indicator/",
    "dc11": "http://purl.org/dc/elements/1.1/",
    "qb": "http://purl.org/linked-data/cube#",
    "canfam-dimension": "http://data.socialhistory.org/vocab/canfam/dimension/",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "canfam": "https://iisg.amsterdam/canfam/dataset/canada1901/",
    "napp-sct81": "https://iisg.amsterdam/napp/dataset/scotland1881/",
    "sdmx-measure": "http://purl.org/linked-data/sdmx/2009/measure#",
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "sdr": "http://example.com/resource/",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "time": "http://www.w3.org/2006/time#",
    "napp-dimension": "http://data.socialhistory.org/vocab/napp/dimension/"
   }
  ],
  "dc:title": "imf_gdppc.csv",
  "@id": "http://example.com/resource/imf_gdppc.csv",
  "dc:modified": {
   "@value": "2018-11-14",
   "@type": "xsd:date"
  },
  "tableSchema": {
   "aboutUrl": "{_row}",
   "primaryKey": "Rank",
   "columns": [
    {
     "datatype": "string",
```


(continues on next page)

(continued from previous page)

```
    "titles": [
     "Rank"
    ],
    "@id": "http://example.com/resource/imf_gdppc.csv/column/Rank",
    "name": "Rank",
    "dc:description": "Rank"
   },
   {
    "datatype": "string",
    "titles": [
     "Country"
    ],
    "@id": "http://example.com/resource/imf_gdppc.csv/column/Country",
    "name": "Country",
    "dc:description": "Country"
   },
   {
    "datatype": "string",
    "titles": [
     "GDP_Per_Capita"
    ],
    "@id": "http://example.com/resource/imf_gdppc.csv/column/GDP_Per_Capita",
    "name": "GDP_Per_Capita",
    "dc:description": "GDP_Per_Capita"
   }
  ]
 }
}
```

The exact meaning of this structure is explained in *the section below*.

3.2 Converting a CSV file

If we now want to convert our example file `imf_gdppc.csv`, you first make sure you have your virtual environment enabled (if applicable), and run:

```
cow_tool convert imf_gdppc.csv
```

This will produce a file *imf_gdppc.csv.nq* that holds an NQuads serialization of the RDF.

This is also the preferred method for converting multiple files at the same time. For instance, if you want to convert *all* CSV files in a specific directory, simply use unix-style wildcards:

```
cow_tool convert /path/to/some/directory/*.csv
```

Going back to our running example, the resulting RDF will be serialized as N-Quads. This is a computer friendly but not so much human friendly serialization so for the benefit of (human) readability below the RDF will be represented in the TriG serialization:

```
@prefix ns1: <http://www.w3.org/ns/prov#> .
@prefix ns2: <http://www.w3.org/ns/csvw#> .
@prefix ns3: <http://schema.org/> .
@prefix ns4: <http://purl.org/dc/terms/> .
@prefix ns5: <urn:uuid:5> .
```

(continues on next page)

(continued from previous page)

```
@prefix ns6: <http://www.nanopub.org/nschema#> .
@prefix ns7: <https://iisg.amsterdam/vocab/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<https://iisg.amsterdam/imf_gdppc/pubinfo/48422b27/2018-11-14T10:59> {
    <https://iisg.amsterdam/imf_gdppc/nanopublication/48422b27/2018-11-14T10:59>
    ns1:generatedAtTime "2018-11-14T10:59:00"^^xsd:dateTime ;
        ns1:wasGeneratedBy <https://github.com/CLARIAH/wp4-converters> .
}

<https://iisg.amsterdam/imf_gdppc/provenance/48422b27/2018-11-14T10:59> {
    <https://iisg.amsterdam/imf_gdppc/assertion/48422b27/2018-11-14T10:59>
    ns1:generatedAtTime "2018-11-14T10:59:00"^^xsd:dateTime ;
        ns1:wasDerivedFrom <http://example.com/resource/imf_gdppc.csv>,
            <https://iisg.amsterdam/48422b27cba4a0e68c9c66d0f7ca614ec688dfcb> .

    <http://example.com/resource/__row_> ns1:wasDerivedFrom "http://example.com/
    resource/{_row}"^^xsd:string .

    <http://example.com/resource/imf_gdppc.csv> ns4:license <http://opendefinition.
    org/licenses/cc-by/> ;
        ns4:modified "2018-11-14"^^xsd:date ;
        ns4:publisher [ ns3:name "CLARIAH Structured Data Hub - Datalegend"@en ;
                ns3:url <http://datalegend.net/> ] ;
        ns4:title "imf_gdppc.csv"@en ;
        ns2:dialect [ ns2:delimiter ";" ;
                ns2:encoding "ascii" ;
                ns2:quoteChar "\"" ] ;
        ns2:tableSchema [ ns2:aboutUrl <http://example.com/resource/__row_> ;
                ns2:column ( <http://example.com/resource/imf_gdppc.csv/column/Rank>
    <http://example.com/resource/imf_gdppc.csv/column/Country> <http://example.com/
    resource/imf_gdppc.csv/column/GDP_Per_Capita> ) ;
                ns2:primaryKey "Rank" ] ;
        ns2:url "imf_gdppc.csv"^^xsd:anyURI .

    <http://example.com/resource/imf_gdppc.csv/column/Country> ns4:description
    "Country"@en ;
        ns2:datatype xsd:string ;
        ns2:name "Country" ;
        ns2:title "Country"@en .

    <http://example.com/resource/imf_gdppc.csv/column/GDP_Per_Capita> ns4:description
    "GDP_Per_Capita"@en ;
        ns2:datatype xsd:string ;
        ns2:name "GDP_Per_Capita" ;
        ns2:title "GDP_Per_Capita"@en .

    <http://example.com/resource/imf_gdppc.csv/column/Rank> ns4:description "Rank"@en
    ;
        ns2:datatype xsd:string ;
        ns2:name "Rank" ;
        ns2:title "Rank"@en .
}

```

(continues on next page)

(continued from previous page)

```
ns5:db490c7-50c3-4ad6-b0df-d48fe3dfa984 {
    <https://iisg.amsterdam/48422b27cba4a0e68c9c66d0f7ca614ec688dfcb> ns7:path "/tmp/V2RY7QULW9/web_interface/91a7c0a271826cf3e7e5b470dfd5e345/imf_gdppc.csv"^^xsd:string ;
        ns7:sha1_hash "48422b27cba4a0e68c9c66d0f7ca614ec688dfcb"^^xsd:string .

    <https://iisg.amsterdam/imf_gdppc/nanopublication/48422b27/2018-11-14T10:59> a ns6:Nanopublication ;
        ns6:hasAssertion <https://iisg.amsterdam/imf_gdppc/assertion/48422b27/2018-11-14T10:59> ;
        ns6:hasProvenance <https://iisg.amsterdam/imf_gdppc/provenance/48422b27/2018-11-14T10:59> ;
        ns6:hasPublicationInfo <https://iisg.amsterdam/imf_gdppc/pubinfo/48422b27/2018-11-14T10:59> .

    <https://iisg.amsterdam/imf_gdppc/assertion/48422b27/2018-11-14T10:59> a ns6:Assertion .

    <https://iisg.amsterdam/imf_gdppc/provenance/48422b27/2018-11-14T10:59> a ns6:Provenance .

    <https://iisg.amsterdam/imf_gdppc/pubinfo/48422b27/2018-11-14T10:59> a ns6:PublicationInfo .
}

<https://iisg.amsterdam/imf_gdppc/assertion/48422b27/2018-11-14T10:59> {
    <http://example.com/resource/0> ns7:Country "Qatar"^^xsd:string ;
        ns7:GDP_Per_Capita "131,063"^^xsd:string ;
        ns7:Rank "1"^^xsd:string .

    <http://example.com/resource/1> ns7:Country "Luxembourg"^^xsd:string ;
        ns7:GDP_Per_Capita "104,906"^^xsd:string ;
        ns7:Rank "2"^^xsd:string .

    <http://example.com/resource/2> ns7:Country "Macau"^^xsd:string ;
        ns7:GDP_Per_Capita "96,832"^^xsd:string ;
        ns7:Rank "3"^^xsd:string .

    <http://example.com/resource/3> ns7:Country "Singapore"^^xsd:string ;
        ns7:GDP_Per_Capita "90,249"^^xsd:string ;
        ns7:Rank "4"^^xsd:string .

    <http://example.com/resource/4> ns7:Country "Brunei Darussalam"^^xsd:string ;
        ns7:GDP_Per_Capita "83,513"^^xsd:string ;
        ns7:Rank "5"^^xsd:string .

    <http://example.com/resource/5> ns7:Country "Kuwait"^^xsd:string ;
        ns7:GDP_Per_Capita "72,675"^^xsd:string ;
        ns7:Rank "6"^^xsd:string .

    <http://example.com/resource/6> ns7:Country "Ireland"^^xsd:string ;
        ns7:GDP_Per_Capita "72,524"^^xsd:string ;
        ns7:Rank "7"^^xsd:string .

    <http://example.com/resource/7> ns7:Country "Norway"^^xsd:string ;
        ns7:GDP_Per_Capita "70,645"^^xsd:string ;
        ns7:Rank "8"^^xsd:string .
```

(continues on next page)

(continued from previous page)

```
}
```

What does this mean?

- Everything in `https://iisg.amsterdam/imf_gdppc/provenance/48422b27/2018-11-14T10:59` is the RDF representation of the CSVW JSON schema.
- Everything in `https://iisg.amsterdam/imf_gdppc/assertion/48422b27/2018-11-14T10:59` is the RDF representation of the CSV file.

  Since the global `aboutUrl` is set to `{_row}`, every row is represented in RDF as a resource with the base URI concatenated with the row number. The column names are used as predicates to relate the row resource to a string literal representation of the value of a cell in that row.
- The graph `ns5:db490c7-50c3-4ad6-b0df-d48fe3dfa984` is the default graph that contains the Nanopublication.

CHAPTER 4

The Schema

The CoW converter uses the CSWV standard syntax for defining mappings from CSV to RDF graphs. These mappings are all defined in the `tableSchema` dictionary. For a full reference of the things you can do, we refer to the CSV on the Web (CSVW) specification and in particular to the document on Generating RDF from Tabular Data on the Web.

Important: CoW does not purport to implement the full CSVW specification, nor has it been tested against the official test suite. In fact, CoW extends and deviates from the CSVW specification in several important ways.

We document the most important differences in the section below, and give a *short overview* of how schemas can be defined.

4.1 Differences and Extensions

1. While CSVW allows only for simple references to values in a column using the curly-brackets syntax (e.g. `{name}` to refer to the value of the name column at the current row), CoW interprets the strings containing these references in two ways:

    1. as Python Format Strings, and
    2. as Jinja2 Templates

    This allows for very elaborate operations on row contents (e.g. containing conditionals, loops, and string operations.)[3].

2. CSVW allows only to specify a single `null` value for a column; when the cell in that column is equal to the null value, it is ignored for RDF conversion. CoW extends the CSVW treatment of `null` values in two ways:

1. multiple potential `null` values for a column, expressed as a JSON list, and
2. conditional on values in *another* column, as a JSON-LD list (using the `@list` keyword)
3. CoW allows the use of `csvw:collectionUrl` and `csvw:schemeUrl` on column specifications. This will automatically cast the value for `valueUrl` to a `skos:Concept`, and adds it to the collection or scheme respectively indicated by these urls using a `skos:member` or `skos:inScheme` predicate.

[3] In the future we may enable the Jinja2 plugin mechanism. This will allow running custom Python functions as filters over values.

4. By default CoW skips cells that are empty (as per the CSVW specification), setting the `csvw:parseOnEmpty` attribute to `true` overrides this setting. This is useful when an empty cell has a specific meaning.
5. Column specifications with a `xsd:anyURI` datatype are converted to proper URIs rather than Literals with the `xsd:anyURI` datatype. This allows for conditionally generating URIs across multiple namespaces using Jinja2 templates, see issue #13 .
6. Column specifications in CoW should have a JSON-LD style `@id` attribute. This ensures that all predicates generated through the conversion are linked back to the RDF representation of the CSVW JSON schema that informed the conversion.
7. CoW converts column names to valid Python dictionary keys. In general this means that spaces in column names will be replaced with underscores.
8. For convenience, CoW uses a default set of namespaces, specified in the `src/converter/namespaces.yaml` file, that will be used to interpret namespace prefix use in the JSON schema. Any namespace prefixes defined in the JSON schema will override the default ones.

4.2 Short Overview

A very simple `tableSchema` may have the following structure:

```
"tableSchema": {
  "aboutUrl": "{_row}",
  "primaryKey": "Rank",
  "columns": [
    {
     "@id": "http://example.com/resource/imf_gdppc.csv/column/Rank",
     "dc:description": "Rank",
     "datatype": "string",
     "name": "Rank"
    }
  ]
}
```

For the conversion to RDF, only the `aboutUrl` and `columns` attributes are of importance.

4.2.1 aboutUrl

The `aboutUrl` attribute defines a template for all URIs that occur in the *subject* position of triples generated by the converter. It may appear in the `tableSchema` or in one of the `columns`. If defined in the `tableSchema`, it acts as a *global* template that may be overriden by individual columns.

We explain URL template expansion *here*.

4.2.2 columns

The `columns` array defines a schema for each column, and any additional `virtual` columns. The distinction between the two is important, as non-virtual columns must actually be present in the CSV (schema compliance) while virtual columns only instruct the conversion to RDF.

In the schema above, we state that the column identifiable with the `name Rank` specifies a literal value, with the `datatype` of `string` (a shorthand for `xsd:string`). The `titles` array gives a number of alternative

4.2.3 Column Attributes

Every column is a dictionary that may have the following attributes.

Table 1: Attributes usable in column specifications

Attribute	Explanation
`name`	Specifies the column to which this column specification applies. If no `propertyUrl` is defined on the column, the value for `name` will be used to generate the URL for the *predicate* position of the triple generated.
`virtual`	If set to `true`, the column specification is not taken into account when validating a CSV file against this schema.
`aboutUrl`	Overrides the *global* `aboutUrl` template defined for the schema. This template will be used to generate the *subject* URL of the triple.
`valueUrl`	If present, this template will be used to generate the *object* URL of the triple. Otherwise, the value for `name` is used to retrieve the value for that cell, to generate a URL.
`datatype`	Specifies that this column should result in a triple where the *object* is a `Literal` with the datatype specified here (for common XML Schema datatypes, it is possible to drop the `xsd:` prefix). The value of the literal is then the value of the cell in this row indicated by the value of `name`. **Special case**: when the `datatype` is `xsd:anyURI` COW creates a URI rather than a literal value.
`csvw:value`	Specifies that this column should result in a triple where the *object* is a `Literal` with the default `xsd:string` datatype (unless otherwise specified in the `datatype` attribute). The literal value for this cell is determined by applying the ref::*template expansion <template-expansion>* rule to this row. Can only be used in `virtual` columns.
`csvw:parseOnEmpty`	When set to `true`, specifies that this column should be processed even when the cell corresponding to this column in this row is empty.
`null`	Specifies that this template does not apply if the cell in this column in this row corresponds to the value specified here. Can take a single value (as per the CSVW spec) or an array of values.
`lang`	Specifies the language tag for the literal in the *object* position, but only if the `datatype` is set to be `string`.
`collectionUrl`	Specifies that the `valueUrl` (or equivalent) should be of type `skos:Concept` and that it is a `skos:member` of the URL generated by applying the `collectionUrl` template.
`schemeUrl`	Specifies that the `valueUrl` (or equivalent) should be of type `skos:Concept` and that it is `skos:inScheme` the URL generated by applying the `schemeUrl` template.

4.2.4 Template Expansion with Jinja2 templates and Python format strings

When a CSV file is processed, CoW does this row by row in the file, producing a dictionary where key/value pairs correspond to column headers and the value of the cell. So for:

```
Rank;Country;GDP_Per_Capita
1;Qatar;131063
```

the first row becomes[5]

```
row = {'Rank': 1, 'Country': 'Qatar', 'GDP_Per_Capita': 131063}
```

For each row, CoW then applies each column definition in the `columns` array in the JSON-LD file (i.e. which does not have to mean each column in the CSV file).

The URL templates in the attributes `aboutUrl`, `propertyUrl`, `valueUrl`, and the regular template in the `csvw:value` are used to generate URLs and Literal values from the values of the cells in a specific row.

[5] Assuming that you have the proper locale settings that instructs Python to interpret the comma as a thousands separator.

The values for the URL templates that the parser receives are *interpreted as URLs*. This means that they are expanded relative to the `@base` URI of the CSVW JSON schema file, unless they are explicitly preceded by a defined namespace prefix.

The names of Jinja2 or Python formatting field names should correspond to the keys of the dictionary (i.e. to the column names). CoW supports a special CSVW field name `_row` that inserts the row number. This means that our row now becomes:

```
row = {'Rank': 1, 'Country': 'Qatar', 'GDP_Per_Capita': 131063, '_row': 1}
```

With this preparation of the row data the template expansion can begin. CoW always first applies: * the Jinja2 template (see documentation), * and then the Python format strings (see documentation).

For instance (assuming a `@base` of `http://example.com/`), we define an `aboutUrl` with the special `_row` key as a Python string formatting field name, and `Country` as a Jinja2 field name:

```
"aboutUrl": "{_row}/{{Country}}"
```

the JSON-LD parser interprets the value for `aboutUrl` as the following URI:

```
"http://example.com/{_row}/{{Country}}"
```

we then apply the Jinja2 formatting (`Template("http://example.com/{_row}{{Country}}").render(**row)`):

```
"http://example.com/{_row}/Qatar"
```

followed by the Python formatting (`"http://example.com/{_row}/{{Country}}".format(**row)`):

```
"http://example.com/1/Qatar"
```

For `csvw:value` attributes this works similarly, with the exception that the JSON-LD parser will not interpret these fields as URIs:

```
"csvw:value": "{_row}/{{Country}}"
```

is parsed as:

```
"{_row}/{{Country}}"
```

This means that one can use Jinja2 conditional formatting on `csvw:value` atributes in combination with an `xsd:anyURI` value for `datatype` to generate custom URIs that do not fit within a defined namespace.

Jinja2 is a very expressive templating language. To give a small example, we could define a `virtual` column that allows us to specify whether a country is `http://example.com/rich` or `http://example.com/poor` depending on whether the GDP is over 100k.

Our virtual column may look as follows:

```
{
  "virtual": "true",
  "aboutUrl": "{Country}",
  "propertyUrl": "rdf:type",
  "valueUrl": "{% if GDP_Per_Capita > 100000 %}rich{% else %}poor{% endif %}"
}
```

This will produce, for Qatar and Singapore, the respective triples:

```
<http://example.com/Qatar>     rdf:type <http://example.com/rich> .
<http://example.com/Singapore> rdf:type <http://example.com/poor> .
```

If you happen to be a bit experienced with the Python3 or ipython shell, then you could also quickly test Jinja templates like so:

```
from jinja2 import Template
my_jinja_template = "{% if GDP_Per_Capita > 100000 %}rich{% else %}poor{% endif %}"
row = {'Rank': 1, 'Country': 'Qatar', 'GDP_Per_Capita': 131063}
Template(my_jinja_template).render(row)
# returns 'rich'
```

CHAPTER 5

FAQ: Frequently Asked Questions

Please refer to our wiki for questions on specific topics.

5.1 Commonly used Template Formatting

- Leading zeroes: `{{'%05d'|format(variable|int)}}`, where `5` is the number of digits to fill up to.
- If-else statements: `{% if conditional_variable=="something" %} value_if {% else %} value_else {% endif %}`.
- Convert to string and concatenate: `{{variable ~ 'string'}}`, e.g. if variable has value "Hello" then the result would be "Hello string". Note the double braces.
- Arithmetic: use double braces and cast as numeric first, e.g. `{{variable|float() * 1000}}`.
- Lowercase, uppercase, etc.: `{{variable|lower()}}`` . Note the double brace.
- String slices: `{{variable[n:m]}}` as described here.

CHAPTER 6

API Documentation

- code

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

7.1 Footnotes