dctap-python

Release 0.4.2

unknown

Jan 26, 2023

CONTENTS

1 DCTAP Model **3**
- 1.1 Minimal application profile . . . 3

2 DCTAP Elements **5**
- 2.1 Statement Template elements . . . 5
- 2.2 Shape elements . . . 17

3 Configuration **21**
- 3.1 Default Shape Identifier . . . 21
- 3.2 Namespace Prefix Mappings . . . 21
- 3.3 Extra Elements . . . 21
- 3.4 Picklist Elements . . . 22
- 3.5 Picklist Item Separator . . . 23
- 3.6 Extra Value Node Types . . . 24
- 3.7 Element Aliases . . . 24

4 Design principles **25**
- 4.1 Empty rows are ignored. . . . 25
- 4.2 Keywords are normalized to lowercase. . . . 25
- 4.3 The sequence of elements is normalized. . . . 26
- 4.4 Non-DCTAP elements are ignored unless configured. . . . 26
- 4.5 DCTAP elements are not repeatable. . . . 28
- 4.6 Some variants of element names are tolerated. . . . 28
- 4.7 Some element names are not allowed. . . . 28
- 4.8 Shapes may be declared on separate rows. . . . 29
- 4.9 Shape elements are set just once. . . . 29
- 4.10 Elements belong either to shapes or to statement templates, not both. . . . 30

5 Command-line tool **31**
- 5.1 Initialize a config file . . . 31
- 5.2 View a TAP as TXT, JSON, or YAML . . . 31

6 DCTAP Glossary **35**

Index **37**

dctap is a Python package for parsing and normalizing spreadsheets or *CSV File*s that follow the model for DC Tabular Application Profiles (DCTAP) (see installation instructions).

The **dctap** package includes a command-line tool for viewing the normalized contents of a given *TAP* in one of three interchangeable formats: a verbose indented-TXT format (for human users) and YAML or JSON (for machines). The tool checks a *CSV File* for potential violations of the DCTAP model and emits warnings or helpful suggestions.

An *Application Profile* describes models, vocabularies, and usage patterns that are expected or required to be found in *Instance Data*. Developing a shared profile can help data providers capture consensus models on the "shape" of data in a given domain and improve the coherence or interoperability of data in that domain. Developing that profile in a simple spreadsheet, using DCTAP, can make it easier for people to participate in that process and use its results.

An Application Profile is also commonly used as a basis for data validation. While the **dctap** package itself does not support validation (or any other operation touching on instance data), it can however serve as a preprocessor for validation applications downstream. The normalized representation of a DCTAP CSV in JSON, for example, can be converted into validation schemas expressed in Shape Expressions Language (ShEx) or Shapes Constraint Language (SHACL).

dctap aims at catching a few of the more obvious inconsistencies in a given *TAP* – malformed regular expressions, the use of literal datatypes with nonliteral values, and the like. These checks are documented below and in extensive unit tests. The checks err on the side of tolerance, and error messages are meant as helpful hints to editors of early drafts. Users are free to customize the DCTAP model with local extensions. Any part of a given *TAP* not recognized by **dctap** as a built-in or customized feature of the DCTAP model is simply ignored.

CHAPTER

ONE

DCTAP MODEL

Reality	Metadata	DCTAP	Vocabulary
Entity			Class
	(Statements about an entity)	Shape	
	Statement	Statement Template	
	Predicate	Predicate Constraint	Property
	Value	Value Constraint	

Instance Data, whether in the form of metadata records, databases, or networked graphs such as Wikidata, typically makes statements about things "in the world" — books, their authors, viruses, buildings, and the like. A single *Statement* consists of a *Property-Value* pair. A set of statements about a distinct entity in the world is referred to here as as a *Description*. Because a given body of *Instance Data* may describe multiple things in the world, it may be said to consist of multiple *Description*s.

An *Application Profile* (here: a *TAP*) enumerates the properties and characterizes the values that are expected to be found in a given body of *Instance Data*. In effect, an Application Profile is a description of a description – a notion that is inevitably somewhat confusing. To minimize this confusion, the DCTAP Model names the things "in *Instance Data*" differently from the things "in an *Application Profile*" (see table above).

In an *Application Profile*: - a *Statement* in *Instance Data* is described with a *Statement Template*; - a Property-Value pair is described with a *Predicate Constraint* and *Value Constraint*; - a set of *Statement*s in *Instance Data* about exactly one real-world *Entity* (aka *Description*) is described in a *Shape*. Where a *Description* in *Instance Data* groups a set of *Statement*s, a *Shape* groups a set of *Statement Template*s.

The DCTAP Model consists of *Shape*s and *Statement Template*s, each of which consists of *DCTAP Element*s (a generic term for the column headers in a *CSV File*).

Because the DCTAP Model was designed for compatibility with RDF and Linked Data, property constraints, shapes, literal datatypes, and some value constraints are represented in a *TAP* with *IRI*s (or *Compact IRI*s).

1.1 Minimal application profile

In the DCTAP model, the simplest possible application profile consists of just one *Statement Template* in the context of one *Shape*.

A Statement Template has, at a minimum, one **propertyID** element, and the existence of a Shape can be inferred, so in practical terms, the simplest possible application profile is a list of just one property.

Note that if a shape identifier is not explicitly assigned in a CSV, a default identifier will be assigned. (This is discussed in the the section *shapeID / shapeLabel*.) In "shape-less" applications, this shape identifier can simply be ignored.

propertyID
http://purl.org/dc/terms/title
http://purl.org/dc/terms/publisher
https://schema.org/creator
http://purl.org/dc/terms/date

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          http://purl.org/dc/terms/title
        Statement Template
            propertyID          http://purl.org/dc/terms/publisher
        Statement Template
            propertyID          https://schema.org/creator
        Statement Template
            propertyID          http://purl.org/dc/terms/date
```

CHAPTER

TWO

DCTAP ELEMENTS

In the DCTAP Model, a Shape groups a set of Statement Templates, each of which describes one type of Statement in Instance Data about a specified Entity. Each of these two components (Shapes and Statement Templates) has its own (extensible) set of DCTAP Elements:

2.1 Statement Template elements

2.1.1 propertyID / propertyLabel

The DCTAP model was designed for compatibility with the RDF model. In the RDF model, properties are identified with IRIs, and this module will issues a warning if a property identifier, based on a superficial inspection, does not look like an IRI.

Users not interested in compatibility with RDF, or users who are brainstorming a draft application profile and simply need a placeholder, can safely ignore such a warning.

propertyID
dcterms:creator
height

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:creator
        Statement Template
            propertyID          height

WARNING [default/propertyID] 'height' is not an IRI or Compact IRI.
```

Properties can have natural-language labels for use in displays and documentation.

propertyID	propertyLabel
dct:creator	Author or Creator

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dct:creator
            propertyLabel       Author or Creator
```

2.1.2 mandatory / repeatable

In the DCTAP model, the expected cardinality of a property can be expressed with the elements **mandatory** and **repeatable**. These elements take Boolean values that express "true" or "false" in one of two supported ways:

- The keywords **true** and **false** (case-insensitive).
- The integers **0** and **1**.

These supported Boolean values are handled in the following ways:

- normalized as the Boolean class instances True and False in the internal Python object.
- normalized as true and false in the JSON and YAML outputs,
- displayed as True and False in the compact text output.

Note that empty values (ie, strings of length zero) are simply interpreted as unspecified and are not assigned an explicit Boolean value.

propertyID	mandatory	repeatable
dc:creator	1	0
dc:date	0	

This is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dc:creator
            mandatory           True
            repeatable          False
        Statement Template
            propertyID          dc:date
            mandatory           False
```

Any other value for either element — including an empty string for when the element is present but left blank — has no effect on the default of **None** for each element and will be passed through as a string (or empty string) to the JSON and YAML output.

An empty string value will result in an element not being displayed at all in the compact text output; in the verbose text format, the value will be displayed as the default "None".

propertyID	mandatory	repeatable
dc:creator		N
dc:date	Y	

This is displayed as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dc:creator
            repeatable          N
        Statement Template
            propertyID          dc:date
            mandatory           Y

WARNING [default/repeatable] 'N' is not a supported Boolean value.
WARNING [default/mandatory] 'Y' is not a supported Boolean value.
```

The four possible combinations of **mandatory** with **repeatable** translate into the following minimum and maximum values when cardinality is expressed as a range (where "-1" means "many").

mandatory/repeatable		min/max	
mand	repeat	min	max
False	False	0	1
True	False	1	1
False	True	0	-1
True	True	1	-1

Users of DCTAP in areas such as biology, where more expressive cardinality is required, may want to extend the model with such ranges.

2.1.3 valueNodeType

The DCTAP model was designed for compatibility with the RDF model. In the RDF model, there are three types of node: an *IRI* (or URI), a BNode, and a Literal.

Users not interested in compatibility with RDF can safely ignore this element.

dctap issues a warning if an unsupported value is provided (here: "Concept").

propertyID	valueNodeType
dcterms:title	Literal
dcterms:creator	URI
dcterms:subject	Concept

Interpreted, with a warning, as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:title
            valueNodeType       literal
        Statement Template
            propertyID          dcterms:creator
```

(continues on next page)

(continued from previous page)

```
            valueNodeType        uri
        Statement Template
            propertyID           dcterms:subject
            valueNodeType        concept

WARNING [default/valueNodeType] 'concept' is not a valid node type.
```

2.1.4 valueDataType

The DCTAP model was designed for compatibility with the RDF model. In the RDF model, literal values can be tagged with a datatype that marks the value as a date, string, decimal number, and the like. The most commonly used datatypes are defined in the W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes specification.

Because datatypes are identified by IRI, this module issues a warning if a non-IRI keyword is encountered. Users not interested in compatibility with RDF can safely ignore such a warning.

propertyID	valueDataType
dc:creator	xsd:string
dct:date	Date

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                  default
        Statement Template
            propertyID           dc:creator
            valueDataType        xsd:string
        Statement Template
            propertyID           dct:date
            valueDataType        Date

WARNING [default/valueDataType] 'Date' is not an IRI or Compact IRI.
```

Datatypes are used only with literal values, so if a node is of type “URI”, “IRI”, or “BNode” and any datatype is provided, this will trigger a warning.

Note that if a URI is meant to be processed as a string, the node type should be “Literal”.

propertyID	valueNodeType	valueDataType
dcterms:creator	IRI	xsd:string
dcterms:subject	BNODE	xsd:string

Interpreted, with a warning, as:

```
DCTAP instance
    Shape
        shapeID                  default
        Statement Template
            propertyID           dcterms:creator
            valueNodeType        iri
```

(continues on next page)

(continued from previous page)

```
            valueDataType        xsd:string
        Statement Template
            propertyID           dcterms:subject
            valueNodeType        bnode
            valueDataType        xsd:string

WARNING [default/valueDataType] Datatypes are only for literals, so node type should not␣
↪be 'iri'.
WARNING [default/valueDataType] Datatypes are only for literals, so node type should not␣
↪be 'bnode'.
```

2.1.5 valueConstraint / valueConstraintType

A value constraint (**valueConstraint**) constrains the value associated with a property in specific ways according to its type (**valueConstraintType**). A value constraint type may define a specific interpretation of a value constraint or trigger specific techniques for processing the value constraint in an application downstream.

Value constraints with no value constraint types

When a value constraint is provided without a value constraint type, it is treated as a plain literal (unless **valueNodeType** is "IRI" or "BNode"). Typically, this is intended to close the set of possible values to one specific value and no others. In the following example, the value expected to be found with the property **:securityLevel** is "Confidential" (and no other).

propertyID	valueConstraint
:securityLevel	Confidential

This is interpreted as:

```
DCTAP instance
    Shape
        shapeID                  default
        Statement Template
            propertyID           :securityLevel
            valueConstraint      Confidential
```

Value constraint types with no value constraints

Because the value constraint type is intended to provide a context for interpreting a value constraint, a value constraint type means nothing in the absence of a value constraint. If a value is provided for **valueConstraintType** but not for **valueConstraint**, a warning will be emitted.

propertyID	valueConstraintType
:securityLevel	picklist

is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          :securityLevel
            valueConstraintType  picklist

WARNING [default/valueConstraint] Value constraint type ('picklist') but no value␣
↪constraint.
```

Built-in value constraint types

The **valueConstraintType** element is intended to serve as an extension point for implementers of the DCTAP model. As proof of concept, four commonly used value constraint types are supported by default:

Picklist

Value constraints of type "Picklist" are split into lists of literals (strings) by using the *Picklist Item Separator*, by default whitespace. Lists are rendered in the text display as quoted strings, separated by commas and enclosed in square brackets, and in the JSON and YAML outputs as list objects.

In the following example: - In the absence of valueConstraintType "picklist", "red blue green" is a string value. - With valueConstraintType "picklist", "red blue green" is parsed on whitespace into a list. - With valueConstraintType "picklist", "yellow" is parsed on whitespace into a list with a single item.

propertyID	valueConstraint	valueConstraintType
:color	red blue green	
:color	red blue green	picklist
:color	yellow	picklist

This is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          :color
            valueConstraint     red blue green
        Statement Template
            propertyID          :color
            valueConstraint     ['red', 'blue', 'green']
            valueConstraintType  picklist
        Statement Template
            propertyID          :color
            valueConstraint     ['yellow']
            valueConstraintType  picklist
```

If **dctap** is configured to use a comma as the *Picklist Item Separator*, the CSV

propertyID	valueConstraint	valueConstraintType
:color	reddish brown, greenish yellow, bluish green	picklist

is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          :color
            valueConstraint     ['reddish brown', 'greenish yellow', 'bluish green']
            valueConstraintType picklist
```

Pattern

A value constraint type of Pattern means that the corresponding value constraint is to be interpreted as a (Python) regular expression. If value constraints are empty or malformed as regular expressions they will be passed through, untouched, to the text, JSON, and YAML outputs. If a pattern does not correctly compile as a Python regular expression (or is empty), a warning will be emitted.

shapeID	propertyID	valueConstraint	valueConstraintType
:ex1	:status	approved_*	Pattern
:ex2	:status		Pattern
:ex3	:status	approved_(*	Pattern
:ex4	:status	/approved_*/	Pattern
:ex5	:status	^2020 August	Pattern

This is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 :ex1
        Statement Template
            propertyID          :status
            valueConstraint     approved_*
            valueConstraintType pattern
    Shape
        shapeID                 :ex2
        Statement Template
            propertyID          :status
            valueConstraintType pattern
    Shape
        shapeID                 :ex3
        Statement Template
            propertyID          :status
            valueConstraint     approved_(*
            valueConstraintType pattern
    Shape
        shapeID                 :ex4
        Statement Template
            propertyID          :status
            valueConstraint     /approved_*/
            valueConstraintType pattern
    Shape
```

(continues on next page)

(continued from previous page)

```
        shapeID                 :ex5
        Statement Template
            propertyID          :status
            valueConstraint     ^2020 August
            valueConstraintType pattern

WARNING [:ex2/valueConstraint] Value constraint type is 'pattern', but value constraint␣
↪is empty.
WARNING [:ex3/valueConstraint] Value constraint type is 'pattern', but 'approved_(*' is␣
↪not a valid regular expression.
```

IRIStem

A value of type IRIStem consists of one or more IRIs (or *Compact IRI*s) for matching against longer IRIs. For example, "http://lod.nal.usda.gov/nalt/" is an IRIStem that matches "http://lod.nal.usda.gov/nalt/10129".

More than one IRI stem can be provided by separating the IRIs with blank spaces. IRI stems are always output as a list, even if just one IRI stem is provided.

The following example says that values of the "dcterms:subject" property are expected to be values of the NAL Thesaurus.

propertyID	valueConstraint	valueConstraintType
dcterms:subject	http://lod.nal.usda.gov/nalt/	IRIStem

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:subject
            valueConstraint     ['http://lod.nal.usda.gov/nalt/']
            valueConstraintType iristem
```

This module will superficially check whether the value constraint looks like an IRI and, if not, emit a warning.

propertyID	valueConstraint	valueConstraintType
dcterms:subject	nalt	IRIStem

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:subject
            valueConstraint     ['nalt']
            valueConstraintType iristem
```

(continues on next page)

(continued from previous page)

```
WARNING [default/valueConstraint] Value constraint type is 'iristem', but 'nalt' does␣
↪not look like an IRI or Compact IRI.
```

LanguageTag

A *Language Tag* is an abbreviated name for a natural language, such as `fr` for French or `fr-CA` for Canadian French. Language tags are used to identify the language of a *Literal*. Standard sets of language tags serve as a controlled vocabulary of identifiers for languages.

A value constraint of type "languageTag" is processed as a picklist of one or more language tags. Specifying language tags in this manner means that the value associated with the property (in the example below, with ":status") is expected to be a string tagged with one of the language tags.

As with the value constraint type *Picklist*, a value constraint of type "LanguageTag" is split on whitespace unless another list separator has been defined (see section *Configuration*).

A string with no whitespace is parsed into a list with just one string. As the rules for well-formed language tags are quite complex, the module makes no attempt to check whether the language tags themselves are well-formed.

propertyID	valueConstraint	valueConstraintType
:status	fr	LanguageTag
:status	fr fr-CA	LanguageTag

This is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          :status
            valueConstraint     ['fr']
            valueConstraintType languagetag
        Statement Template
            propertyID          :status
            valueConstraint     ['fr', 'fr-CA']
            valueConstraintType languagetag
```

MinInclusive / MaxInclusive

A value constraint of type MinInclusive or MaxInclusive is used with a numeric value constraint (integer or float) to indicate the minimum or maximum of a numeric value. "Inclusive" means that the value provided will also match:

- MinInclusive means "greater than or equal to".
- MaxInclusive means "less than or equal to".

Note that because the columns for value constraint and value constraint type are not repeatable in the base DCTAP model, these value constraint types cannot be used to indicate ranges (eg, "-9 to -2"). Users who need to express value ranges should consider extending DCTAP, for example as follows:

- With a single column that uses an application-specific syntax for ranges (eg, "1-6").
- With two columns: one for MinInclusive and one for MaxInclusive.

propertyID	valueConstraint	valueConstraintType
:temperature	9	MinInclusive
:temperature	-9	MinInclusive
:temperature	12.2	MaxInclusive
:temperature	-2	MaxInclusive
:temperature	info@example.org	MaxInclusive

This is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          :temperature
            valueConstraint     9
            valueConstraintType  mininclusive
        Statement Template
            propertyID          :temperature
            valueConstraint     -9
            valueConstraintType  mininclusive
        Statement Template
            propertyID          :temperature
            valueConstraint     12.2
            valueConstraintType  maxinclusive
        Statement Template
            propertyID          :temperature
            valueConstraint     -2
            valueConstraintType  maxinclusive
        Statement Template
            propertyID          :temperature
            valueConstraint     info@example.org
            valueConstraintType  maxinclusive

WARNING [default/valueConstraint] Value constraint type is 'mininclusive', but
↪'info@example.org' is not numeric.
```

Note:

- When viewed with the default text display (as above), non-numeric value constraints are flagged with warnings.
- When output as JSON, numeric values are coerced to integers or floats, as appropriate. Values that are not coercable are passed through as strings:

```
{
    "shapes": [
        {
            "shapeID": "default",
            "statement_templates": [
                {
                    "propertyID": ":temperature",
                    "valueConstraint": 9,
                    "valueConstraintType": "mininclusive"
                },
```

(continues on next page)

(continued from previous page)

```
                {
                    "propertyID": ":temperature",
                    "valueConstraint": 12.2,
                    "valueConstraintType": "maxinclusive"
                },
                {
                    "propertyID": ":temperature",
                    "valueConstraint": "info@example.org",
                    "valueConstraintType": "maxinclusive"
                }
            ]
        }
    ]
}
```

MinLength / MaxLength

A value constraint of type MinLength or MaxLength defines a minimum or maximum length of a string value:

- MinLength means a string is at least X characters long.
- MaxLength means a string no longer than X characters long.

Note that because the columns for value constraint and value constraint type are not repeatable in the base DCTAP model, these value constraint types cannot be used to indicate ranges (eg, "2 to 9"). Users who need to express value ranges should consider extending DCTAP, for example as follows:

- With a single column that uses an application-specific syntax for ranges (eg, "1-6").
- With two columns: one for MinInclusive and one for MaxInclusive.

propertyID	valueConstraint	valueConstraintType
:identifier	3	MinLength
:identifier	3.1	MinLength
:identifier	-10	MaxLength
:identifier	info@example.org	MaxLength

This is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          :identifier
            valueConstraint     3
            valueConstraintType minlength
        Statement Template
            propertyID          :identifier
            valueConstraint     3.1
            valueConstraintType minlength
        Statement Template
            propertyID          :identifier
```

(continues on next page)

(continued from previous page)

```
            valueConstraint      -10
            valueConstraintType  maxlength
        Statement Template
            propertyID           :identifier
            valueConstraint      info@example.org
            valueConstraintType  maxlength

WARNING [default/valueConstraint] Value constraint type is 'minlength', but '3.1' is not␣
↪an integer.
WARNING [default/valueConstraint] Value constraint type is 'maxlength', but
↪'info@example.org' is not an integer.
```

Values of type MinLength or MaxLength must be integers. Note:

- String and float values trigger warnings but are passed through, untouched, as string values.
- Negative integers do not trigger warnings, though they may not make sense.

Recall that the element *valueDataType* is used for general datatypes of literal values, such as "string" and "date". The element *valueConstraint / valueConstraintType* is used for more specific or rarely used types of value. While every imaginable value constraint type could, in principle, have its own column in a tabular application profile, the resulting tables would be overly wide and this specification would be more longer and difficult to use. Pairing value constraint types with value constraints in just two columns helps keep tabular profiles more compact and concise.

Custom value constraint types

The built-in value constraint types are intended only as examples. Implementers are encouraged to define their own types. If a **valueConstraintType** other than the four built-in types is provided — in the following example, a hypothetical type **markdown** — **dctap** will simply pass the value through to the output, where any consuming applications will be responsible for processing the type correctly.

propertyID	valueConstraint	valueConstraintType
:tutorial	click [here](https://sphinx-rtd-tutorial.readthedocs.io)	markdown

is interpreted as:

```
DCTAP instance
    Shape
        shapeID                  default
        Statement Template
            propertyID           :tutorial
            valueConstraint      click [here](https://sphinx-rtd-tutorial.readthedocs.io)
            valueConstraintType  markdown
```

2.1.6 valueShape

By specifing the *Shape* to which the *Description* of the resource represented by the *Value* — ie, the object of a *Statement* in the *Instance Data* — is expected to conform, the **valueShape** element connects the shapes of a profile.

A value shape identifier may be a literal, blank node, or IRI, so no checks are performed on the value of this element.

The example below says:

- A book, as described according to the ":book" shape, has a creator.
- The creator of the book must be described in accordance with the ":person" shape.
- The ":person" shape says that the description of a person must include their name.

shapeID	propertyID	valueShape
:book	dct:creator	:person
:person	foaf:name	

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 :book
        Statement Template
            propertyID          dct:creator
            valueShape          :person
    Shape
        shapeID                 :person
        Statement Template
            propertyID          foaf:name
```

2.1.7 note

The "note" element is a catch-all field for annotating any aspect of a Shape or of a Statement Template.

Users requiring annotations that are more specific, for example to generate forms or displays, may want to extend the DCTAP model with more precisely defined annotation elements.

2.2 Shape elements

There are two Shape elements. If the **shapeID** element is not used in a given DCTAP instance, it will be assigned a default value (which can be customized in the config file - see *Default Shape Identifier*).

2.2.1 shapeID / shapeLabel

In the DCTAP model, all *Statement Template*s are seen as grouped into *Shape*s, where a Shape is about a *Description* in *Instance Data* — a set of statements about just one *Entity* in the real world.

A shape identifier is typically a plain *Literal* or an *IRI*.

If no **shapeID** is provided in the CSV or in a configuration file (see *Configuration*), a default shape identifier will be assigned ("default"). A different default shape identifier may be configured, as described in the section *Configuration*. For example:

propertyID
dcterms:creator
dcterms:date

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:creator
        Statement Template
            propertyID          dcterms:date
```

Users with metadata about a single *Entity*, or whose downstream applications do not make use of shapes, can safely ignore this default identifier.

A **shapeID**, once declared, will apply to any immediately subsequent rows where the **shapeID** is left blank. However, a shape ID may be declared explicitly for any or for every row. When shape IDs are explicitly declared, they can be presented in any arbitrary sequence without compromising their proper grouping as shapes. Declaring shape IDs explicitly makes it possible to combine statement templates from multiple sources without regard for their sequential order.

shapeID	propertyID
:book	dcterms:creator
	dcterms:date
:author	foaf:name
:book	dcterms:language

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 :book
        Statement Template
            propertyID          dcterms:creator
        Statement Template
            propertyID          dcterms:date
        Statement Template
            propertyID          dcterms:language
    Shape
        shapeID                 :author
```

(continues on next page)

(continued from previous page)

```
        Statement Template
            propertyID          foaf:name
```

If a shape identifier is not provided for the first rows processed but is provided for rows processed thereafter, only the shape identifier for the first statement templates will be the default.

shapeID	propertyID
	dcterms:creator
	dcterms:date
:author	foaf:name

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:creator
        Statement Template
            propertyID          dcterms:date
    Shape
        shapeID                 :author
        Statement Template
            propertyID          foaf:name
```

Shapes can also have labels for use in displays and documentation.

shapeID	shapeLabel	propertyID
:book	Book	dcterms:creator

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 :book
        shapeLabel              Book
        Statement Template
            propertyID          dcterms:creator
```

Note that a shape label does not function as a shape identifier. If no value is provided for **shapeID** it will be assigned a (configurable) default. Only the assignment of a new **shapeID** will trigger the creation of a new shape. In the example below, the second **shapeLabel** ("Libro") is simply ignored.

shapeLabel	propertyID
Book	dcterms:creator
Libro	dcterms:creator

Interpreted as:

```
DCTAP instance
    Shape
```

(continues on next page)

(continued from previous page)

```
        shapeID                 default
        shapeLabel              Book
        Statement Template
            propertyID          dcterms:creator
        Statement Template
            propertyID          dcterms:creator
```

CHAPTER

THREE

CONFIGURATION

dctap has built-in default config settings (see defaults.py). By generating and editing a config file (see *Initialize a config file*), the following defaults can be tweaked:

3.1 Default Shape Identifier

When shape identifiers are not provided in a CSV, a configurable default shape name is used (see section *shapeID / shapeLabel*).

3.2 Namespace Prefix Mappings

As explained in the section *View a TAP as TXT, JSON, or YAML*, the *Compact IRI*s can be expanded into full *IRI*s by replacing the short prefix with the full IRI of the namespace. The default configuration settings provide a starter set of prefix mappings for frequently used namespaces. This list can be customized with locally defined namespaces or with namespaces listed in services such as prefix.cc <http://prefix.cc/'_ or `Linked Open Vocabularies.

3.3 Extra Elements

By default, **dctap** ignores elements that are not part of the DCTAP model. As explained in the section "*Non-DCTAP elements are ignored unless configured.*", **dctap** can be configured to recognize extra elements as belonging either to a shape or to a statement template. In the absence of such configuration, **dctap** has no basis for handling a given element as a shape constraint or a statement constraint. Columns with unrecognized headers are simply ignored and passed through, unchanged to text, JSON, or YAML output.

3.3.1 Extra shape elements

Extra CSV columns (elements) can be configured as shape constraints by enumerating the column headers (element names) as follows:

```
extra_shape_elements:
- closed
- start
```

3.3.2 Extra statement template elements

Extra CSV columns (elements) can be configured as statement template elements by enumerating the column headers (element names) as follows:

```
extra_statement_template_elements:
- min
- max
```

3.4 Picklist Elements

Some statement template elements can be configured as picklist elements. Cell values of picklist elements are split into lists of multiple values on the basis of a configurable *Picklist Item Separator*. Value lists may be used or interpreted differently in applications downstream of a DCTAP instance. The semantic implications of using list values with given elements in particular applications is out of scope for DCTAP.

There are two cases where a list may be used as the value of an element:

- In the context of a specific statement constraint, a **valueConstraint** is provided together with a **valueConstraintType** of "picklist".
- An element has been declared in the config file as a picklist element - ie, all values in that given column are to be treated as lists.

Note that the following types of statement template element cannot sensibly be configured for use with multiple values:

- Elements with numeric values: **min**, **max**
- Elements with Boolean values: **closed**, **start**, **mandatory**, **repeatable**

Elements used purely for annotation, such as **shapeLabel**, **propertyLabel**, and **note**, could in principle be configured for use with multiple values (eg, with labels in multiple languages).

On the example of **propertyID**, given:

propertyID
dc:creator foaf:maker

In the following example, the value of **propertyID** would by default be interpreted as including an (illegal) space:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dc:creator foaf:maker
```

However, if **dctap** were so configured:

```
picklist_elements:
- propertyID
```

The value would be interpreted as a list:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          ['dc:creator', 'foaf:maker']
```

Note that a column can be either a regular column or a list column, but not both - ie, all cells in a given column will be treated either as single values or as lists. In the following table:

propertyID
dc:creator,foaf:maker
dc:date

the value "dc:date" is treated as an item a list that has just one value:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          ['dc:creator', 'foaf:maker']
        Statement Template
            propertyID          ['dc:date']
```

3.5 Picklist Item Separator

The value of a picklist element, a string, is parsed into a list of substrings on the basis of a list item separator, by default a single space. This default separator can be changed to a different character, such as a comma or pipe (orbar). For example, the element **propertyID** can be configured as a picklist element with a non-default list item separator, such as a comma:

```
picklist_elements:
- propertyID
picklist_item_separator: ','
```

In this case, a **propertyID** containing a comma, such as:

propertyID
dc:creator,foaf:maker

would be parsed as a list with two values, as in the example shown in the section *Picklist Elements*.

Note, however, that because columns in CSVs are, by definition, separated by commas, a value with an embedded comma, as above, must be enclosed in quotes. Exporting to CSV from an Excel spreadsheet yields a result such as the following, where the multiple values in cell A2 are enclosed in quotes:

```
propertyID,valueNodeType
"dc:creator,foaf:maker",iri
```

3.6 Extra Value Node Types

According to the DCTAP Primer, DCTAP supports the three node types of the graph-based data model as defined in RDF 1.1 Concepts and Abstract Syntax: IRI, literal, and blank node. These are represented in a *TAP* with "IRI", "Literal", and "BNode" as keywords for the element *valueNodeType*. Users can extend this list of supported keywords with aliases for supported node types, such as "URI" (for "IRI") or with combinations of node types that will be understood by applications downstream.

For example, it is often necessary to say that the value is "not a literal" or, in other words, that it is an "IRI or bnode". It is of course possible to handle this by declaring **valueNoteType** to be a picklist element as described in the section *Picklist Elements* above. It can however be convenient to coin extra node types to cover the most common use cases. ShEx covers this case with the value "nonliteral", while SHACL provides three additional pairwise combinations, "sh:BlankNodeOrIRI", "sh:BlankNodeOrLiteral", and "sh:IRIOrLiteral". One may also want to use "URI" instead of "IRI".

Any or all of these options can be activated by editing the configuration file accordingly:

```
extra_value_node_types:
- uri
- nonliteral
- IRIOrLiteral
```

3.7 Element Aliases

The width of CSVs can be reduced by creating aliases for headers. For aliases, case, whitespace, and punctuation are ignored, but the canonical element names to which they map must exactly match those presented in the section *DCTAP Elements*. Aliases will be expanded to the canonical element names in text, JSON, and YAML output. For example, given the following configuration file ("dctap.yaml"):

```
prefixes:
    ":":        "http://example.org/"
    "dc:":      "http://purl.org/dc/elements/1.1/"

extra_element_aliases:
    "PropID": "propertyID"
```

The following table:

SID	PropertyID	Mand	Rep
:book	dc:creator	1	0

Is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dc:creator
```

Aliases can also be used for translations of CSV headers into other languages.

CHAPTER

FOUR

DESIGN PRINCIPLES

The following principles govern how **dctap** processes a CSV file. Comments are welcome in the Github issue tracker.

4.1 Empty rows are ignored.

For the purposes of **dctap**, a row is "empty" if it does not have a value either for **shapeID** or for **propertyID**.

The CSV:

shapeID	propertyID	valueNodetype
book		
		literal
	dc:creator	uri
book	dc:date	literal

is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 book
        Statement Template
            propertyID          dc:creator
            valueNodeType       uri
        Statement Template
            propertyID          dc:date
            valueNodeType       literal
```

4.2 Keywords are normalized to lowercase.

Value constraint types and value node types are normalized to lowercase. In the example below, "LITERAL", "Literal", and "lITERAL" are normalized to "literal", while "Picklist", "PICKLIST", and "pICKLIST" are normalized to "picklist".

Property ID	Value_Node_type	Value_Constraint	Value_Constraint_Type
dc:subject	Literal	Kish Uruk Nuzi	Picklist
dc:subject	LITERAL	Kish Uruk Nuzi	PICKLIST
dc:subject	lITERAL	Kish Uruk Nuzi	pICKLIST

Interpreted as:

```
DCTAP instance
Shape
    shapeID                 default
    Statement Template
        propertyID          dc:subject
        valueNodeType       literal
        valueConstraint     ['Kish', 'Uruk', 'Nuzi']
        valueConstraintType picklist
    Statement Template
        propertyID          dc:subject
        valueNodeType       literal
        valueConstraint     ['Kish', 'Uruk', 'Nuzi']
        valueConstraintType picklist
    Statement Template
        propertyID          dc:subject
        valueNodeType       literal
        valueConstraint     ['Kish', 'Uruk', 'Nuzi']
        valueConstraintType picklist
```

4.3 The sequence of elements is normalized.

In order to improve the consistency and readability of results, he order of *DCTAP Element*s will be normalized in text, JSON, and YAML outputs irrespective of their sequence in a CSV,

valueShape	propertyID	shapeLabel	shapeID
:author	dcterms:creator	Book	:book

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 :book
        shapeLabel              Book
        Statement Template
            propertyID          dcterms:creator
            valueShape          :author
```

4.4 Non-DCTAP elements are ignored unless configured.

Columns in a CSV that are not part of the DCTAP model are not automatically passed through to text, YAML, or JSON output because unrecognized elements, in principle, bear an undefined relationship to *Shape*s and *Statement Template*s.

propertyID	Status
dcterms:creator	ignotus

Interpreted (with warnings enabled) as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:creator

WARNING [csv/header] Non-DCTAP element 'Status' not configured as extra element.
```

Users wishing to use columns in their CSV that are not part of the DCTAP model, for example to specify that a shape is "closed" or to specify "severity" of validation errors, can generate a configuration file (see section *Initialize a config file*) and list their extra column headers in the configuration file under the sections "extra_shape_elements" or "extra_statement_template_elements". This will ensure that the extra columns will be passed through to the text, JSON, and YAML outputs.

For example, if the configuration file includes:

```
extra_statement_template_elements:
- status
```

The text output, intended as an aid in debugging, includes the extra element but marks it as "extra" with brackets:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dcterms:creator
            [status]            ignotus
```

The JSON (or YAML) output includes the extra element "as is":

```
{
    "shapes": [
        {
            "shapeID": "default",
            "statement_templates": [
                {
                    "propertyID": "dcterms:creator",
                    "status": "ignotus"
                }
            ]
        }
    ]
}
```

4.5 DCTAP elements are not repeatable.

Elements cannot be repeated, i.e., used as a header for more than one column in a CSV. This module ignores all but the last column with a given header.

propertyID	note	note
dc:creator	Writer of the report.	Typically, the person listed on the cover page.

Interpreted as:

```
DCTAP instance
    Shape
        shapeID                 default
        Statement Template
            propertyID          dc:creator
            note                Typically, the person listed on the cover page.
```

4.6 Some variants of element names are tolerated.

When processing CSV headers, the module ignores case, whitespace, and dashes and understores. All of the following variants of "propertyID" are normalized to "propertyID":

- "Property ID"
- "Property-ID"
- "Property_ID"
- "propertyid"
- "p r o p e r t y i d"

4.7 Some element names are not allowed.

Some keywords may not be used as names of elements (i.e., of CSV column headers):

- "state_list"
- "shape_warns"
- "state_warns"
- "shape_extras"
- "state_extras"

Note that in processing headers, the module ignores case, certain punctuation (dashes and understores), and whitespace, so none of the following variants of "state_list" may be used as element names (see *Some variants of element names are tolerated.*):

- "SC List"
- "SC-List"
- "SCLIST"

4.8 Shapes may be declared on separate rows.

Shapes, if declared on a row separately from statement templates, will apply to all subsequent statement templates - until a new **shapeID** is encountered. For example, given the following configuration file settings:

extra_shape_elements: - “closed” - “start”

The CSV:

shapeID	propertyID	valueNodetype	closed	start
book			True	True
	dc:creator	uri		
	dc:subject	literal		
author				
	foaf:name	literal		

is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 book
        [closed]                True
        [start]                 True
        Statement Template
            propertyID          dc:creator
            valueNodeType       uri
        Statement Template
            propertyID          dc:subject
            valueNodeType       literal
    Shape
        shapeID                 author
        Statement Template
            propertyID          foaf:name
            valueNodeType       literal
```

4.9 Shape elements are set just once.

Values for shape elements are set from the row where a new shape is first encountered. Shape element values asserted in subsequent rows are ignored. For example, given the following configuration file settings:

extra_shape_elements: - “closed” - “start”

The CSV:

shapeID	propertyID	valueNodetype	closed	start
book	dc:creator	uri		True
book	dc:date	literal	False	False

is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 book
        [start]                 True
        Statement Template
            propertyID          dc:creator
            valueNodeType       uri
        Statement Template
            propertyID          dc:date
            valueNodeType       literal
```

4.10 Elements belong either to shapes or to statement templates, not both.

A given element is defined either as an element of a shape or an element of a statement template. At one's own risk, one can configure a statement constraint element as an "extra shape element" (or vice versa), for example with:

```
extra_shape_elements:
- "note"
```

However, the results may be unexpected. The CSV:

shapeID	propertyID	note
book		Note on a Shape
	dc:creator	Note on a Statement Template
author	foaf:name	Where does this note belong?

is interpreted as:

```
DCTAP instance
    Shape
        shapeID                 book
        [note]                  Note on a Shape
        Statement Template
            propertyID          dc:creator
            note                Note on a Statement Template
    Shape
        shapeID                 author
        [note]                  Where does this note belong?
        Statement Template
            propertyID          foaf:name
            note                Where does this note belong?
```

This ambiguity could be solved simply by coining an extra element, eg **shapeNote**:

```
extra_shape_elements:
- "shapeNote"
```

CHAPTER

FIVE

COMMAND-LINE TOOL

With the command-line tool **dctap**, one can:

5.1 Initialize a config file

The command **dctap read** works out of the box, with no options, but its behavior can be customized with an optional configuration file (see *Configuration*).

5.1.1 Per-directory config files

The subcommand **dctap init** writes a starter configuration file, **dctap.yaml**, in the working directory. Thereafter, whenever **dctap read** is run, the program will look in the working directory for **dctap.yaml** or, if it is not found, will use built-in defaults.

```
cd /home/tombaker/myproject/data/
dctap init                  # Write default dctap.yaml
dctap read x.csv            # Looks for dctap.yaml or reads defaults.
```

5.1.2 Global config files

Once generated, config files may be moved to arbitrary locations or even renamed. As described in the section *View a TAP as TXT, JSON, or YAML*, config files at arbitrary locations may be referenced by their absolute or relative pathnames with the option **–config [path-to-configfile]**. In this way, one central config file can be referenced from anywhere on the file system or multiple config files can be created with alternative settings.

5.2 View a TAP as TXT, JSON, or YAML

The subcommand **dctap read**:

- reads a CSV file - alternatively, reads CSV file contents from stdin (eg, **cat example.csv | dctap read -**)
- sends a lightly normalized view of a TAP to stdout - by default, outputs TXT for on-screen debugging, without showing prefixes - with option **–json**, outputs JSON, with namespace prefixes - with option **–yaml**, outputs YAML, with namespace prefixes

The option **–expand-prefixes** expands any *Compact IRI* into a full *IRI* using prefixes found in the built-in defaults or as overridden by a *configuration file*.

The file **example.csv**:

shapeID	propertyID	valueNodeType
:a	dcterms:creator	IRI

can be read as TXT, with full IRIs, with **dctap read –expand-prefixes example.csv**:

```
DCTAP instance
    Shape
        shapeID                 http://example.org/a
        Statement Template
            propertyID          http://purl.org/dc/terms/creator
            valueNodeType       iri
```

Or as JSON with **dctap read –json example.csv**:

```
{
  "shapes": [
    {
      "shapeID": ":a",
      "statement_templates": [
        {
          "propertyID": "dcterms:creator",
          "valueNodeType": "iri"
        }
      ]
    }
  ],
  "namespaces": {
    ":": "http://example.org/",
    "dcterms:": "http://purl.org/dc/terms/"
  }
}
```

Or as YAML, with full IRIs, with **dctap read –yaml example.csv**:

```
shapes:
  - shapeID: :a
    statement_templates:
      - propertyID: dcterms:creator
        valueNodeType: iri
namespaces:
  ':': http://example.org/
  'dcterms:': http://purl.org/dc/terms/
```

5.2.1 View warnings generated

As an aid for debugging, **dctap read –warnings** generates warnings for any obvious inconsistencies or errors found in the TAP.

Specific consistency checks are explained in the descriptions of individual *DCTAP elements*; see section *DCTAP Elements*.

dctap read –warnings example2.csv sends warnings in plain text to stderr:

```
DCTAP instance
    Shape
        shapeID                 :a
        Statement Template
            propertyID          dcterms:date
            valueNodeType       noodles

WARNING [:a/valueNodeType] 'noodles' is not a valid node type.
```

dctap read –warnings –json example2.csv includes warnings in the JSON dictionary:

```
{
  "shapes": [
    {
      "shapeID": "default",
      "statement_templates": [
        {
          "propertyID": "dcterms:date",
          "valueNodeType": "noodles"
        }
      ]
    }
  ],
  "namespaces": {
    "dcterms:": "http://purl.org/dc/terms/"
  },
  "warnings": {
    "default": {
      "valueNodeType": [
        "'noodles' is not a valid node type."
      ]
    }
  }
}
```

dctap read –warnings –yaml example2.csv includes warnings in the YAML output:

```
shapes:
  - shapeID: default
    statement_templates:
      - propertyID: dcterms:date
        valueNodeType: noodles
namespaces:
  'dcterms:': http://purl.org/dc/terms/
warnings:
```

(continues on next page)

(continued from previous page)

```
  default:
    valueNodeType:
      - "'noodles' is not a valid node type."
```

5.2.2 Read settings from nondefault config file

The option **–configfile** can point to non-default *configuration files*.

A starter configuration file can be generated with **dctap init**, as described in the section *Initialize a config file*. As discussed in the section *Configuration*, settings such as the default shape name and namespace prefix mappings can be tweaked in this file.

```
$ dctap read --configfile /home/tbaker/dctap.yaml example.csv
```

CHAPTER

SIX

DCTAP GLOSSARY

Application Profile
: A description of the models, vocabularies, and usage patterns that are expected or required to be found in *Instance Data*. An application profile that follows the *DCTAP Model* is documented in a *TAP*.

Blank Node
: In RDF, a blank node is a unique identifier used, typically, within the local scope of a specific file or RDF store. As described in RDF 1.1 Concepts and Abstract Syntax, a blank node is distinct both from an *IRI* and a *Literal*. Blank nodes are of interest only to users or creators of RDF applications.

Compact IRI
: An IRI represented by an abbreviated syntax in which a label associated with a namespace (the prefix) is followed by a colon and by a local name which, taken together, can be expanded into a full IRI. For example, if the prefix "dcterms:" is associated with the namespace "http://purl.org/dc/terms/", then the prefixed name "dcterms:creator" can be expanded into "http://purl.org/dc/terms/creator".

CSV File
: A text file in which data values are delimited with commas or with other standard punctuation.

Datatype
: As per RDF 1.1 Concepts and Abstract Syntax, a datatype is used to tag a *Literal* as being a specific type of date or number or, by default, just a plain string. In RDF, datatypes are identified with *IRI*s.

DCTAP Element
: One of a dozen or so labels defined in the DCTAP Model, such as *propertyID*, *valueConstraint*, and *shapeLabel*, used as column headers in a CSV.

Description
: A set of Statements in *Instance Data* used to describe just one real-world *Entity*.

Entity
: Something, typically in the real world, that is described by *Instance Data*.

Instance Data
: Records or, more recently, "graphs" that carry Descriptions, traditionally on paper but now, more typically, on the Web.

IRI
: An Internationalized Resource Identifier is a Web-based identifier that builds on and expands the Uniform Resource Identifier (URI), and is used, for our purposes, to provide the Properties, Entities, and other components of Instance Data, with identity within the globally managed context of the Web.

Language Tag
: A language tag is an abbreviated name for a natural language, such as `fr` for French or `fr-CA` for Canadian French. Language tags are used to identify the language of a *Literal*. Standard sets of language tags serve as a controlled vocabulary of identifiers for languages.

Literal
: Along with *IRI* and *Blank Node*, Literal is one of the three allowable node types defined in the abstract syntax of RDF. For the purposes of DCTAP, it is close enough to think of literals as strings. Literals are used for values such as strings, numbers, and dates. Interested readers can learn more about how literals relate to "lexical forms", *Datatype*s, and *Language Tag*s by consulting RDF 1.1 Concepts and Abstract Syntax.

Picklist
: A controlled list of valid options, one of which can be picked.

Picklist Element
: A *DCTAP Element*, the values of which must be selected from a *Picklist*.

Property
: A controlled term in *Instance Data* denoting an attribute of an Entity referenced in a Statement.

Predicate Constraint
: A pattern in an *Application Profile* descriptive of how a given *Property* is expected to be used in *Instance Data*. Also commonly referred to as a Property Constraint.

Shape
: A component in an *Application Profile* (aka *TAP*) that holds a set of *Statement Template*s. In the now-superseded DCMI Abstract Model of 2007, these were called Description Templates.

Statement
: A property-value pair in *Instance Data* used in a Description to make claims about an Entity.

Statement Template
: A component in an *Application Profile* that describes a *Statement* expected to be found in *Instance Data*.

TAP
: A "TAP" (for "tabular application profile") is a single instance of an *Application Profile* that follows the *DCTAP Model* and is typically serialized as a spreadsheet or *CSV File*.

URI
: See *IRI*.

Value
: A value in *Instance Data* associated with a *Property* in the context of a *Statement*.

Value Constraint
: A pattern in an *Application Profile* descriptive of *Value*s expected in *Instance Data*.

Vocabulary
: A set of Properties and other terms used in *Instance Data* and referred to in constraints defined in an *Application Profile*. By convention, all properties referenced in a Dublin-Core-style Application Profile are defined and documented separately from the profile itself.

INDEX

A

Application Profile, 35

B

Blank Node, 35

C

Compact IRI, 35
CSV File, 35

D

Datatype, 35
DCTAP Element, 35
Description, 35

E

Entity, 35

I

Instance Data, 35
IRI, 35

L

Language Tag, 35
Literal, 36

P

Picklist, 36
Picklist Element, 36
Predicate Constraint, 36
Property, 36

S

Shape, 36
Statement, 36
Statement Template, 36

T

TAP, 36

U

URI, 36

V

Value, 36
Value Constraint, 36
Vocabulary, 36