
pronto

Release 1.1.3

Martin Larralde

Nov 10, 2019

CONTENTS

1	Setup	3
2	Library	5
2.1	Installation	5
2.2	Examples	6
2.3	API Reference	9
2.4	To-do	35
2.5	Changelog	35
2.6	About	38
3	Indices and tables	39
	Python Module Index	41
	Index	43

A Python frontend to ontologies.

`pronto` is a Python agnostic library designed to work with ontologies. At the moment, it can parse [OBO](#), [OBO Graphs](#) or [OWL in RDF/XML format](#), ntologies on the local host or from an network location, and export ontologies to [OBO](#) or [OBO Graphs](#) (in [JSON](#) format).

CHAPTER ONE

SETUP

Run `pip install pronto` in a shell to download the latest release and all its dependencies from PyPi, or have a look at the [Installation page](#) to find other ways to install `pronto`.

Note: `pronto` requires `fastobo`, an efficient and faultless parser for the OBO language implemented in Rust. Most platforms, such as Linux x86-64, OSX and Windows x86-64 provide precompiled packages, but other less frequent platforms will require a working Rust toolchain. See the `fastobo` [Installation page](#) and the [Rust Forge tutorial](#) for more information about this topic.

2.1 Installation

2.1.1 PyPi

`pronto` is hosted on GitHub, but the easiest way to install it is to download the latest release from its [PyPi repository](#). It will install all dependencies then install the `pronto` module:

```
$ pip install --user pronto
```

2.1.2 Conda

Pronto is also available as a [recipe](#) in the [bioconda](#) channel. To install, simply use the `conda` installer:

```
$ conda install -c bioconda pronto
```

2.1.3 GitHub + pip

If, for any reason, you prefer to download the library from GitHub, you can clone the repository and install the repository by running (with the admin rights):

```
$ pip install --user https://github.com/althonos/pronto/archive/master.zip
```

Keep in mind this will install the development version of the library, so not everything may work as expected compared to a stable versioned release.

2.1.4 GitHub + setuptools

If you do not have `pip` installed (the Makefile uses `pip` to install dependencies and then install `pronto`), you can do the following (after having properly installed all the dependencies):

```
$ git clone https://github.com/althonos/pronto
$ cd pronto
# python setup.py install
```

2.2 Examples

2.2.1 Exploring MzML files with the MS Ontology

In this example, we will learn how to use `pronto` to extract a hierarchy from the MS Ontology, a controlled vocabulary developed by the Proteomics Standards Initiative to hold metadata about Mass Spectrometry instrumentation and other Protein Identification and Quantitation software. This example is taken from a real situation that kickstarted the development of `pronto` to extract metadata from [MzML files](#), a file format for Mass Spectrometry data based on XML.

Loading `ms.obo`

The MS ontology is available online on the [OBO Foundry](#), so unless we are using a local version we can simply use the version found online to load the OBO file. We may get some encoding warnings since `ms.obo` imports some legacy ontologies, but we should be OK for the most part since we are only querying terms directly.

```
[1]: import pronto
ms = pronto.Ontology.from_obo_library("ms.obo")

/home/althonos/.local/lib/python3.7/site-packages/pronto/parsers/base.py:48:
↳UnicodeWarning: could not find encoding, assuming UTF-8
    self.ont.timeout,
```

Displaying a class hierarchy with Vega

The MS ontology contains a catalog of several instruments, grouped by instrument manufacturers, but not all instruments are at the same depth level. We can easily use the `Term.subclasses` method to find all instruments defined in the controlled vocabulary. Let's then build a tree from all the subclasses of `MS:1000031`:

```
[2]: instruments = set(ms['MS:1000031'].subclasses())
data = []

for term in instruments:
    value = {"id": int(term.id[3:]), "name": term.id, "desc": term.name}
    parents = instruments.intersection(term.relationships.get(ms['is_a'], set()))
    if parents:
        value['parent'] = int(parents.pop().id[3:])
    data.append(value)
```

Now that we have our tree structure, we can render it simply with [Vega](#) to get a better idea of the classes we are inspecting:

```
[3]: import json
import urllib.request
from vega import Vega

# Let's use the Vega radial tree example as a basis of the visualization
view = json.load(urllib.request.urlopen("https://vega.github.io/vega/examples/radial-
↳tree-layout.vg.json"))

# First replace the default data with our own
view['data'][0].pop('url')
view['data'][0]['values'] = data
view['marks'][1]['encode']['enter']['tooltip'] = {"signal": "datum.desc"}
```

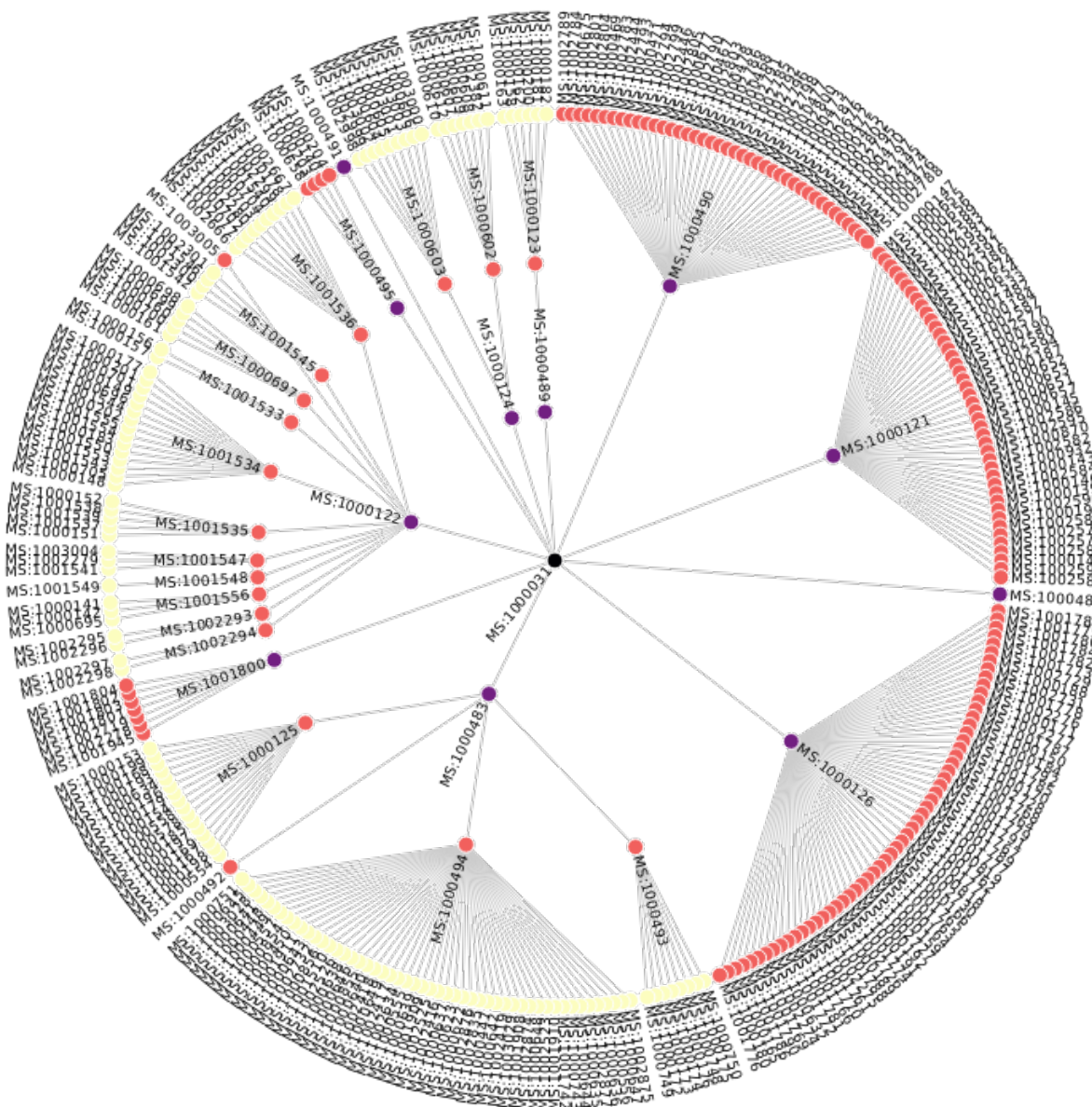
(continues on next page)

(continued from previous page)

```
view['signals'][4]['value'] = 'cluster'

# Render the clustered tree
Vega(view)
```

```
[3]: <vega.vega.Vega at 0x7f2f5f0ba7d0>
```



Extracting the instruments from an MzML file

MzML files store the metadata corresponding to one or several MS scans using the MS controlled vocabulary, but the location and type of metadata can vary and needs to be extracted from a term subclassing hierarchy. Let's download an example file from the MetaboLights library and parse it with `xml.etree`:

```
[4]: import urllib.request
import xml.etree.ElementTree as etree

URL = "http://ftp.ebi.ac.uk/pub/databases/metabolights/studies/public/MTBLS341/pos_
↳Exp2-K3_2-E,5_01_7458.d.mzML"
mzml = etree.parse(urllib.request.urlopen(URL))
```

Now we want to extract the instruments that were used in the MS scan, that are stored as `mzml:cvParam` elements: we build a set of all the instruments in the MS ontology, and we iterate over the instruments `mzml:cvParam` to find the ones that refer to instruments:

```
[5]: instruments = {term.id for term in ms["MS:1000031"].subclasses()}
study_instruments = []

path = "mzml:instrumentConfigurationList/mzml:instrumentConfiguration/mzml:cvParam"
for element in mzml.iterfind(path, {'mzml': 'http://psi.hupo.org/ms/mzml'}):
    if element.attrib['accession'] in instruments:
        study_instruments.append(ms[element.attrib['accession']])

print(study_instruments)

[Term('MS:1000703', name='microTOF-Q')]
```

Finally we can extract the manufacturer of the instruments we found by checking which one of its superclasses is a direct child of the `MS:1000031` term. We use the `distance` argument of `subclasses` to get the direct subclasses of `instrument model`, which are the manufacturers, and we use set operations to select manufacturers from the superclasses of each instrument we found.

```
[6]: manufacturers = set(ms['MS:1000031'].subclasses(distance=1))
manufacturers.discard(ms['MS:1000031'])
study_manufacturers = []

for instrument in study_instruments:
    study_manufacturers.extend(manufacturers & set(instrument.superclasses()))

print(study_manufacturers)

[Term('MS:1000122', name='Bruker Daltonics instrument model')]
```

Validating the controlled vocabulary terms in an MzML file

All `mzml:cvParam` XML elements are required to have the 3 following attributes: * `accession`, which is the identifier of the term in one of the ontologies imported in the file * `cvRef`, which is the identifier of the ontology imported in the file * `name`, which is the textual definition of the term

`name` in particular is redundant with respect to the actual ontology file, but can help rendering the XML elements. However, some MzML files can have a mismatch between the `name` and `accession` attributes. In order to check these mismatches we can use `pronto` to retrieve the name of all of these controlled vocabulary terms.

```
[7]: mismatches = [
    element
    for element in mzml.iter()
    if element.tag == "{http://psi.hupo.org/ms/mzml}cvParam"
    if element.get('accession') in ms
    if ms[element.get('accession')].name != element.get('name')
]
```

```
[8]: for m in mismatches:
      print(f"{m.get('accession')}: {m.get('name')!r} (should be {ms[m.get('accession')
      ↪'].name!r})")
```

```
MS:1000815: 'Bruker BAF file' (should be 'Bruker BAF format')
```

```
MS:1000590: 'contact organization' (should be 'contact affiliation')
```

2.3 API Reference

A Python frontend to ontologies.

pronto is a Python agnostic library designed to work with ontologies. At the moment, it can parse ontologies in the OBO, OBO Graphs or OWL in RDF/XML format, on either the local host or from an network location, and export ontologies to OBO or OBO Graphs (in JSON format).

Caution: Only classes and modules reachable from the top-level package `pronto` are considered public and are guaranteed stable over [Semantic Versioning](#). Use submodules (other than `warnings`) at your own risk!

Note: `pronto` implements proper *type checking* for most of the methods and properties exposed in the public classes. This reproduces the behaviour of the Python standard library, to avoid common errors. This feature does however increase overhead, but can be disabled by executing Python in optimized mode (with the `-O` flag). **The time to parse an OBO file is reduced by about 60% in optimized mode.**

2.3.1 Ontology

An abstraction over a $SHOIN^{(D)}$ ontology.

`pronto.Ontology`

An ontology storing terms and the relationships between them.

Ontology

class `pronto.Ontology`

An ontology storing terms and the relationships between them.

Ontologies can be loaded with `pronto` if they are serialized in any of the following ontology languages and formats at the moment:

- [Ontology Web Language 2](#) in RDF/XML format.
- [Open Biomedical Ontologies 1.4](#).
- [OBO graphs](#) in JSON format.

metadata

A data structure storing the metadata about the current ontology, either extracted from the `owl:Ontology` XML element or from the header of the OBO file.

Type *Metadata*

timeout

The timeout in seconds to use when performing network I/O, for instance when connecting to the OBO library to download imports. This is kept for reference, as it is not used after the initialization of the ontology.

Type `int`

imports

A dictionary mapping references found in the import section of the metadata to resolved *Ontology* instances.

Type `Dict[str, Ontology]`

Caution: *Ontology* instances share all the data about a deserialized ontology file as well as all of its imports, and contains entities which can be created or viewed with several dedicated methods. These instances, however, **must not outlive the ontology object instead**, as they are only view-models (in the design pattern sense of the definition).

classmethod `from_obo_library` (*slug: str, import_depth: int = -1, timeout: int = 5*) → `pronto.ontology.Ontology`

Create an *Ontology* from a file in the OBO Library.

This is basically just a shortcut constructor to avoid typing the full OBO Library URL each time.

Parameters

- **slug** (*str*) – The filename of the ontology release to download from the OBO Library, including the file extension (should be one of `.obo`, `.owl` or `.json`).
- **import_depth** (*int*) – The maximum depth of imports to resolve in the ontology tree. *Note that the library may not behave correctly when not importing the complete dependency tree, so you should probably use the default value and import everything.*
- **timeout** (*int*) – The timeout in seconds to use when performing network I/O, for instance when connecting to the OBO library to download imports.

Example

```
>>> ms = pronto.Ontology.from_obo_library("apo.obo")
>>> ms.metadata.ontology
'apo'
>>> ms.path
'http://purl.obolibrary.org/obo/apo.obo'
```

__init__ (*handle: Union[BinaryIO, str, None] = None, import_depth: int = -1, timeout: int = 5, cache: bool = True*)

Create a new *Ontology* instance.

Parameters

- **handle** (*str, BinaryIO, or None*) – Either the path to a file or a binary file handle that contains a serialized version of the ontology. If `None` is given, an empty *Ontology* is returned and can be populated manually.
- **import_depth** (*int*) – The maximum depth of imports to resolve in the ontology tree. *Note that the library may not behave correctly when not importing the complete dependency tree, so you should probably use the default value and import everything.*

- **timeout** (*int*) – The timeout in seconds to use when performing network I/O, for instance when connecting to the OBO library to download imports.
- **cache** (*bool*) – Enable caching of subclassing relationships to make *Term.subclasses* much faster, at the cost of a slightly longer parsing step (about 5% of the total time).

Raises

- **TypeError** – When the given *handle* could not be used to parse and ontology.
- **ValueError** – When the given *handle* contains a serialized ontology not supported by any of the builtin parsers.

__len__ () → int

Return the number of entities in the ontology.

This method takes into accounts the terms and the relationships defined in the current ontology as well as all of its imports. To only count terms or relationships, use *len* on the iterator returned by the dedicated methods (e.g. *len(ontology.terms())*).

Example

```
>>> ms = pronto.Ontology.from_obo_library("ms.obo")
>>> len(ms)
6023
>>> len(ms.terms())
5995
>>> len(ms.relationships())
28
```

__iter__ () → *pronto.utils.iter.SizedIterator*[str][str]

Yield the identifiers of all the entities part of the ontology.

__getitem__ (*id: str*) → *Union*[*pronto.term.Term*, *pronto.relationship.Relationship*]

Get any entity in the ontology graph with the given identifier.

__eq__ (*other*)

Return *self==value*.

__ge__ ()

Return *self>=value*.

__gt__ ()

Return *self>value*.

__le__ ()

Return *self<=value*.

__lt__ ()

Return *self<value*.

__repr__ ()

Return a textual representation of *self* that should roundtrip.

get (*k*[, *d*]) → *D*[*k*] if *k* in *D*, else *d*. *d* defaults to *None*.

items () → a set-like object providing a view on *D*'s items

keys () → a set-like object providing a view on *D*'s keys

values () → an object providing a view on *D*'s values

dump (*file*: *BinaryIO*, *format*: *str* = 'obo')
Serialize the ontology to a given file-handle.

Parameters

- **file** (*BinaryIO*) – A binary file handle open in reading mode to write the serialized ontology into.
- **format** (*str*) – The serialization format to use. Currently supported formats are: **obo**, **obojson**.

Example

```
>>> ms = pronto.Ontology.from_obo_library("ms.obo")
>>> with open("ms.json", "wb") as f:
...     ms.dump(f)
```

dumps (*format*: *str* = 'obo') → *str*
Get a textual representation of the serialization ontology.

terms () → *pronto.utils.iter.SizedIterator*[*pronto.term.Term*][*pronto.term.Term*]
Iterate over the terms of the ontology graph.

relationships () → *pronto.utils.iter.SizedIterator*[*pronto.relationship.Relationship*][*pronto.relationship.Relationship*]
Iterate over the relationships of the ontology graph.

Builtin relationships (*is_a*) are not part of the yielded entities, yet they can still be accessed with the *Ontology.get_relationship* method.

create_term (*id*: *str*) → *pronto.term.Term*
Create a new term with the given identifier.

Returns *Term* – the newly created term view, which attributes can be modified directly.

Raises **ValueError** – if the provided *id* already identifies an entity in the ontology graph, or if it is not a valid OBO identifier.

create_relationship (*id*: *str*) → *pronto.relationship.Relationship*
Create a new relationship with the given identifier.

Raises **ValueError** – if the provided *id* already identifies an entity in the ontology graph.

get_term (*id*: *str*) → *pronto.term.Term*
Get a term in the ontology graph from the given identifier.

Raises **KeyError** – if the provided *id* cannot be found in the terms of the ontology graph.

get_relationship (*id*: *str*) → *pronto.relationship.Relationship*
Get a relationship in the ontology graph from the given identifier.

Builtin ontologies (*is_a* and *has_subclass*) can be accessed with this method.

Raises **KeyError** – if the provided *id* cannot be found in the relationships of the ontology graph.

2.3.2 View Layer

The following classes are part of the view layer, and store references to the ontology/entity they were declared in for verification purposes. For instance, this let *pronto* check that a *Synonym* type can only be changed for a type declared in the *Ontology* header.

Because of this reason, none of these classes should be created manually, but obtained from methods of existing *Ontology* or *Entity* instances, such as *Ontology.get_term* to get a new *Term*.

<code>pronto.Entity</code>	An entity in the ontology graph.
<code>pronto.Relationship</code>	A relationship, constitute the edges of the ontology graph.
<code>pronto.Synonym</code>	
<code>pronto.Term</code>	A term, corresponding to a node in the ontology graph.

Entity

class `pronto.Entity`

An entity in the ontology graph.

With respects to the OBO semantics, an *Entity* is either a term or a relationship in the ontology graph. Any entity has a unique identifier as well as some common properties.

`__eq__` (*other: Any*) → bool
Return self==value.

`__lt__` (*other*)
Return self<value.

`__le__` (*other*)
Return self<=value.

`__gt__` (*other*)
Return self>value.

`__ge__` (*other*)
Return self>=value.

`__hash__` ()
Return hash(self).

`__repr__` ()
Return repr(self).

property `alternate_ids`

A set of alternate IDs for this entity.

Type `frozenset` of `str`

property `annotations`

Annotations relevant to the entity.

Type `frozenset` of `PropertyValue`

property `anonymous`

Whether or not the entity has an anonymous id.

Semantics of anonymous entities are the same as B-Nodes in RDF.

Type `bool`

property `builtin`

Whether or not the entity is built-in to the OBO format.

`pronto` uses this tag on the `is_a` relationship, which is the axiomatic to the OBO language but treated as a relationship in the library.

Type `bool`

property comment

A comment about the current entity.

Comments in `comment` clauses are guaranteed to be conserved by OBO parsers and serializers, unlike bang comments. A non `None comment` is semantically equivalent to a `rdfs:comment` in OWL2. When parsing from OWL, several RDF comments will be merged together into a single `comment` clause spanning over multiple lines.

Type `str` or `None`

property created_by

the name of the creator of the entity, if any.

This property gets translated to a `dc:creator` annotation in OWL2, which has very broad semantics. Some OBO ontologies may instead use other annotation properties such as the ones found in [Information Interchange Ontology](#), which can be accessed in the `annotations` attribute of the entity, if any.

Type `str` or `None`

property creation_date

the date the entity was created.

Type `datetime` or `None`

property definition

the textual definition of the current entity.

Definitions in OBO are intended to be human-readable text describing the entity, with some additional cross-references if possible.

Type `str` or `None`

property id

The OBO identifier of the entity.

Identifiers can be either prefixed (e.g. `MS:1000031`), unprefixd (e.g. `part_of`) or given as plain URLs. Identifiers cannot be edited.

Type `str`

property name

The name of the entity.

Names are formally equivalent to `rdf:label` in OWL2. The OBO format version 1.4 made names optional to improve OWL interoperability, as labels are optional in OWL.

Type `str` or `None`

property namespace

the namespace this entity is defined in.

Type `str` or `None`

property obsolete

whether or not the entity is obsolete.

Type `bool`

property subsets

the subsets containing this entity.

Type `frozenset` of `str`

property synonyms

a set of synonyms for this entity.

Type `frozenset` of `Synonym`

property xrefs

a set of database cross-references.

Xrefs can be used to describe an analogous entity in another vocabulary, such as a database or a semantic knowledge base.

Type `frozenset` of `Xref`

add_synonym (*description*: `str`, *scope*: `Optional[str]` = `None`, *type*: `Optional[pronto.synonym.SynonymType]` = `None`, *xrefs*: `Optional[Iterable[pronto.xref.Xref]]` = `None`) → `pronto.synonym.Synonym`

Add a new synonym to the current entity.

Parameters

- **description** (`str`) – The alternate definition of the entity, or a related human-readable synonym.
- **scope** (`str` or `None`) – An optional synonym scope. Must be either **EXACT**, **RELATED**, **BROAD** or **NARROW** if given.
- **type** (`SynonymType` or `None`) – An optional synonym type. Must be declared in the header of the current ontology.
- **xrefs** (iterable of `Xref`, or `None`) – A collections of database cross-references backing the origin of the synonym.

Raises `ValueError` – when given an invalid synonym type or scope.

Returns `Synonym` – A new synonym for the terms. The synonym is already added to the `Entity.synonyms` collection.

Relationship**class** `pronto.Relationship`

A relationship, constitute the edges of the ontology graph.

Also sometimes referred as typedefs, relationship types, properties or predicates. Formally equivalent to a property (either `ObjectProperty` or `AnnotationProperty`) in OWL2.

__eq__ (*other*: `Any`) → `bool`
Return `self==value`.

__ge__ (*other*)
Return `self>=value`.

__gt__ (*other*)
Return `self>value`.

__hash__ ()
Return `hash(self)`.

__le__ (*other*)
Return `self<=value`.

__lt__ (*other*)
Return `self<value`.

__repr__()

Return repr(self).

add_synonym (*description: str, scope: Optional[str] = None, type: Optional[pronto.synonym.SynonymType] = None, xrefs: Optional[Iterable[pronto.xref.Xref]] = None*) → pronto.synonym.Synonym

Add a new synonym to the current entity.

Parameters

- **description** (*str*) – The alternate definition of the entity, or a related human-readable synonym.
- **scope** (*str* or *None*) – An optional synonym scope. Must be either **EXACT**, **RELATED**, **BROAD** or **NARROW** if given.
- **type** (*SynonymType* or *None*) – An optional synonym type. Must be declared in the header of the current ontology.
- **xrefs** (iterable of *Xref*, or *None*) – A collections of database cross-references backing the origin of the synonym.

Raises **ValueError** – when given an invalid synonym type or scope.

Returns *Synonym* – A new synonym for the terms. The synonym is already added to the *Entity.synonyms* collection.

property alternate_ids

A set of alternate IDs for this entity.

Type *frozenset* of *str*

property annotations

Annotations relevant to the entity.

Type *frozenset* of *PropertyValue*

property anonymous

Whether or not the entity has an anonymous id.

Semantics of anonymous entities are the same as B-Nodes in RDF.

Type *bool*

property builtin

Whether or not the entity is built-in to the OBO format.

pronto uses this tag on the *is_a* relationship, which is the axiomatic to the OBO language but treated as a relationship in the library.

Type *bool*

property comment

A comment about the current entity.

Comments in *comment* clauses are guaranteed to be conserved by OBO parsers and serializers, unlike bang comments. A non *None comment* is semantically equivalent to a *rdfs:comment* in OWL2. When parsing from OWL, several RDF comments will be merged together into a single *comment* clause spanning over multiple lines.

Type *str* or *None*

property created_by

the name of the creator of the entity, if any.

This property gets translated to a `dc:creator` annotation in OWL2, which has very broad semantics. Some OBO ontologies may instead use other annotation properties such as the ones found in [Information Interchange Ontology](#), which can be accessed in the `annotations` attribute of the entity, if any.

Type `str` or `None`

property creation_date

the date the entity was created.

Type `datetime` or `None`

property definition

the textual definition of the current entity.

Definitions in OBO are intended to be human-readable text describing the entity, with some additional cross-references if possible.

Type `str` or `None`

property id

The OBO identifier of the entity.

Identifiers can be either prefixed (e.g. `MS:1000031`), unprefixes (e.g. `part_of`) or given as plain URLs. Identifiers cannot be edited.

Type `str`

property name

The name of the entity.

Names are formally equivalent to `rdf:label` in OWL2. The OBO format version 1.4 made names optional to improve OWL interoperability, as labels are optional in OWL.

Type `str` or `None`

property namespace

the namespace this entity is defined in.

Type `str` or `None`

property obsolete

whether or not the entity is obsolete.

Type `bool`

property subsets

the subsets containing this entity.

Type `frozenset` of `str`

property synonyms

a set of synonyms for this entity.

Type `frozenset` of `Synonym`

property xrefs

a set of database cross-references.

Xrefs can be used to describe an analogous entity in another vocabulary, such as a database or a semantic knowledge base.

Type `frozenset` of `Xref`

property antisymmetric

whether this relationship is anti-symmetric.

Type `bool`

property **asymmetric**

whether this relationship is asymmetric.

Type `bool`

Synonym

class `pronto.Synonym`

```
__ge__ (other, NotImplemented=NotImplemented)
    Return a >= b. Computed by @total_ordering from (not a < b).

__gt__ (other, NotImplemented=NotImplemented)
    Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__le__ (other, NotImplemented=NotImplemented)
    Return a <= b. Computed by @total_ordering from (a < b) or (a == b).

__eq__ (other)
    Return self==value.

__lt__ (other)
    Return self<value.

__hash__ ()
    Return hash(self).

__repr__ ()
    Return repr(self).
```

Term

class `pronto.Term`

A term, corresponding to a node in the ontology graph.

Formally a *Term* frame is equivalent to an `owl:Class` declaration in OWL2 language. However, some constructs may not be possible to express in both OBO and OWL2.

Term should not be manually instantiated, but obtained from an existing *Ontology* instance, using either the *create_term* or the *get_term* method.

```
__eq__ (other: Any) → bool
    Return self==value.

__ge__ (other)
    Return self>=value.

__gt__ (other)
    Return self>value.

__hash__ ()
    Return hash(self).

__le__ (other)
    Return self<=value.

__lt__ (other)
    Return self<value.
```

`__repr__()`

Return repr(self).

add_synonym (*description: str, scope: Optional[str] = None, type: Optional[pronto.synonym.SynonymType] = None, xrefs: Optional[Iterable[pronto.xref.Xref]] = None*) → `pronto.synonym.Synonym`

Add a new synonym to the current entity.

Parameters

- **description** (`str`) – The alternate definition of the entity, or a related human-readable synonym.
- **scope** (`str` or `None`) – An optional synonym scope. Must be either **EXACT**, **RELATED**, **BROAD** or **NARROW** if given.
- **type** (`SynonymType` or `None`) – An optional synonym type. Must be declared in the header of the current ontology.
- **xrefs** (iterable of `Xref`, or `None`) – A collections of database cross-references backing the origin of the synonym.

Raises `ValueError` – when given an invalid synonym type or scope.

Returns `Synonym` – A new synonym for the terms. The synonym is already added to the `Entity.synonyms` collection.

property alternate_ids

A set of alternate IDs for this entity.

Type `frozenset` of `str`

property annotations

Annotations relevant to the entity.

Type `frozenset` of `PropertyValue`

property anonymous

Whether or not the entity has an anonymous id.

Semantics of anonymous entities are the same as B-Nodes in RDF.

Type `bool`

property builtin

Whether or not the entity is built-in to the OBO format.

pronto uses this tag on the `is_a` relationship, which is the axiomatic to the OBO language but treated as a relationship in the library.

Type `bool`

property comment

A comment about the current entity.

Comments in `comment` clauses are guaranteed to be conserved by OBO parsers and serializers, unlike bang comments. A non `None comment` is semantically equivalent to a `rdfs:comment` in OWL2. When parsing from OWL, several RDF comments will be merged together into a single `comment` clause spanning over multiple lines.

Type `str` or `None`

property created_by

the name of the creator of the entity, if any.

This property gets translated to a `dc:creator` annotation in OWL2, which has very broad semantics. Some OBO ontologies may instead use other annotation properties such as the ones found in [Information Interchange Ontology](#), which can be accessed in the `annotations` attribute of the entity, if any.

Type `str` or `None`

property creation_date

the date the entity was created.

Type `datetime` or `None`

property definition

the textual definition of the current entity.

Definitions in OBO are intended to be human-readable text describing the entity, with some additional cross-references if possible.

Type `str` or `None`

property id

The OBO identifier of the entity.

Identifiers can be either prefixed (e.g. `MS:1000031`), unprefixes (e.g. `part_of`) or given as plain URLs. Identifiers cannot be edited.

Type `str`

property name

The name of the entity.

Names are formally equivalent to `rdf:label` in OWL2. The OBO format version 1.4 made names optional to improve OWL interoperability, as labels are optional in OWL.

Type `str` or `None`

property namespace

the namespace this entity is defined in.

Type `str` or `None`

property obsolete

whether or not the entity is obsolete.

Type `bool`

property subsets

the subsets containing this entity.

Type `frozenset` of `str`

property synonyms

a set of synonyms for this entity.

Type `frozenset` of `Synonym`

property xrefs

a set of database cross-references.

Xrefs can be used to describe an analogous entity in another vocabulary, such as a database or a semantic knowledge base.

Type `frozenset` of `Xref`

objects (*r*: `pronto.relationship.Relationship`) \rightarrow `Iterator[pronto.term.Term]`

Iterate over the terms *t* verifying `self · r · t`.

Example

```
>>> go = pronto.Ontology.from_obo_library("go.obo")
>>> go['GO:0048870']
Term('GO:0048870', name='cell motility')
>>> list(go['GO:0048870'].objects(go['part_of']))
[Term('GO:0051674', name='localization of cell')]
```

Todo: Make `Term.objects` take in account `holds_over_chain` and `transitive_over` values of the relationship it is building an iterator with.

superclasses (*distance: Optional[int] = None*) → Iterator[pronto.term.Term]

Get an iterator over the superclasses of this `Term`.

In order to follow the semantics of `rdf:subClassOf`, which in turn respects the mathematical inclusion of subset inclusion, `is_a` is defined as a transitive relationship, hence `has_subclass` is also transitive by closure property. Therefore `self` is always yielded first when calling this method.

Parameters `distance` (*int, optional*) – The maximum distance between this node and the yielded superclass (0 for the term itself, 1 for its immediate superclasses, etc.). Use `None` to explore transitively the entire directed graph.

Yields `Term` – Superclasses of the selected term, breadth-first. The first element is always the term itself, use `itertools.islice` to skip it.

Example

```
>>> ms = pronto.Ontology.from_obo_library("ms.obo")
>>> sup = ms['MS:1000143'].superclasses()
>>> next(sup)
Term('MS:1000143', name='API 150EX')
>>> next(sup)
Term('MS:1000121', name='SCIEX instrument model')
>>> next(sup)
Term('MS:1000031', name='instrument model')
```

Note: The time complexity for this algorithm is in $O(n)$, where n is the number of terms in the source ontology.

See also:

The [RDF Schema 1.1](#) specification, defining the `rdfs:subClassOf` property, which the `is_a` relationship is translated to in OWL2 language.

subclasses (*distance: Optional[int] = None*) → Iterator[pronto.term.Term]

Get an iterator over the subclasses of this `Term`.

Yields `Term` – Subclasses of the selected term, breadth-first. The first element is always the term itself, use `itertools.islice` to skip it.

Example

```
>>> ms = pronto.Ontology.from_obo_library("ms.obo")
>>> sub = ms['MS:1000031'].subclasses()
>>> next(sub)
Term('MS:1000031', name='instrument model')
>>> next(sub)
Term('MS:1000121', name='SCIEX instrument model')
>>> next(sub)
Term('MS:1000122', name='Bruker Daltonics instrument model')
```

Note: This method has a runtime of $O(n^2)$ where n is the number of terms in the source ontology in the worst case. This is due to the fact that OBO and OWL only explicit *superclassing* relationship, so we have to build the graph of *subclasses* from the knowledge graph. By caching the graph however, this can be reduced to an $O(n)$ operation.

is_leaf() → bool

Check whether the term is a leaf in the ontology.

We define leaves as nodes in the ontology which do not have subclasses since the subclassing relationship is directed and can be used to create a DAG of all the terms in the ontology.

Example

```
>>> ms = pronto.Ontology.from_obo_library("ms.obo")
>>> ms['MS:1000031'].is_leaf()    # instrument model
False
>>> ms['MS:1001792'].is_leaf()    # Xevo TQ-S
True
```

property disjoint_from

The terms declared as disjoint from this term.

Two terms are disjoint if they have no instances or subclasses in common.

property intersection_of

The terms or term relationships this term is an intersection of.

2.3.3 Model Layer

The following classes are technically part of the data layer, but because they can be lightweight enough to be instantiated directly, they can also be passed to certain functions or properties of the view layer. *Basically, these classes are not worth to implement following the view-model pattern so they can be accessed and mutated directly.*

<code>pronto.Metadata</code>	A mapping containing metadata about the current ontology.
<code>pronto.Definition</code>	A human-readable text definition of an entity.
<code>pronto.Subset</code>	A definition of a subset in an ontology.
<code>pronto.SynonymType</code>	A user-defined synonym type.
<code>pronto.LiteralPropertyValue</code>	A property-value which adds a literal annotation to an entity.

Continued on next page

Table 3 – continued from previous page

<code>pronto.ResourcePropertyValue</code>	A property-value which adds a resource annotation to an entity.
<code>pronto.Xref</code>	A cross-reference to another document or resource.

Metadata

class `pronto.Metadata`

A mapping containing metadata about the current ontology.

format_version

The OBO format version of the referenced ontology. **1.4** is the default since `pronto` can only parse and write OBO documents of that format version.

Type `str`

data_version

The OBO data version of the ontology, which is then expanded to the `versionIRI` if translated to OWL.

Type `str` or `None`

ontology

The identifier of the ontology, either as a short OBO identifier or as a full IRI.

Type `str` or `None`

date

The date the ontology was last modified, if any.

Type `datetime` or `None`

default_namespace

The default namespace to use for entity frames lacking a namespace clause.

Type `str` or `None`

namespace_id_rule

The production rule for identifiers in the current ontology document. *soft-deprecated, used mostly by OBO-Edit or other outdated tools.*

Type `str` or `None`

owl_axioms

A list of OWL axioms that cannot be expressed in OBO language, serialized in OWL2 Functional syntax.

Type `list` of `str`

saved_by

The name of the person that last saved the ontology file.

Type `str` or `None`

auto_generated_by

The name of the software that was used to generate the file.

Type `str` or `None`

subsetdefs

A set of ontology subsets declared in the ontology files.

Type `set` of `str`

imports

A set of references to other ontologies that are imported by the current ontology. OBO requires all entities referenced in the file to be reachable through imports (excluding databases cross-references).

Type `set of str`

synonymtypedefs

A set of user-defined synonym types including a description and an optional scope.

Type `set of SynonymType`

idspace

A mapping between a local ID space and a global ID space, with an optional description of the mapping.

Type `dict of str to couple of str`

remarks

A set of general comments for this file, which will be preserved by a parser/serializer as opposed to inline comments using !.

Type `set of str`

annotations

A set of annotations relevant to the whole file. OBO property-values are semantically equivalent to `owl:AnnotationProperty` in OWL2.

Type `set of PropertyValue`

unreserved

A mapping of unreserved tags to values found in the ontology header.

Type `dict of str to set of str`

__eq__()

Return `self==value`.

__ge__()

Return `self>=value`.

__gt__()

Return `self>value`.

__hash__()

Return `hash(self)`.

__le__()

Return `self<=value`.

__lt__()

Return `self<value`.

__repr__()

Return `repr(self)`.

__bool__() → bool

Return `False` if the instance does not contain any metadata.

Definition

class `pronto.Definition`

A human-readable text definition of an entity.

Definitions are human-readable descriptions of an entity in the ontology graph, with some optional cross-references to support the definition.

Example

Simply create a *Definition* instance by giving it a string:

```
>>> def1 = pronto.Definition('a structural anomaly')
```

Additional cross-references can be passed as arguments, or added later to the `xrefs` attribute of the *Definition*:

```
>>> def2 = pronto.Definition('...', xrefs={pronto.Xref('MGI:Anna')})
>>> def2.xrefs.add(pronto.Xref('ORCID:0000-0002-3947-4444'))
```

The text content of the definition can be accessed by casting the definition object to a plain string:

```
>>> str(def1)
'a structural anomaly'
```

Caution: A *Definition* compare only based on its textual value, independently of the *Xref* it may contains:

```
>>> def2 == pronto.Definition('...')
True
```

Note: Some ontologies use the xrefs of a description to attribute the authorship of that definition:

```
>>> cio = pronto.Ontology.from_obo_library("cio.obo")
>>> sorted(cio['CIO:0000011'].definition.xrefs)
[Xref('Bgee:fbf')]
```

The common usecase however is to refer to the source of a definition using persistent identifiers like ISBN book numbers or PubMed IDs.

```
>>> pl = pronto.Ontology.from_obo_library("plana.obo")
>>> sorted(pl['PLANA:0007518'].definition.xrefs)
[Xref('ISBN:0-71677033-4'), Xref('PMID:4853064')]
```

`__eq__()`
Return self==value.

`__ge__()`
Return self>=value.

`__gt__()`
Return self>value.

`__hash__()`
Return hash(self).

`__le__()`
Return self<=value.

__lt__()
Return self<value.

__repr__()
Return repr(self).

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center()
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(sub[, start[, end]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(*args, **kwargs) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(mapping) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(sub[, start[, end]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as “def” and “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

ljust()

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip()

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()

Return a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition()

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

replace()

Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(sub[, start[, end]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(sub[, start[, end]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises `ValueError` when the substring is not found.

rjust()

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()

Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split()

Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(prefix[, start[, end]]) → bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()

Return a copy of the string with leading and trailing whitespace remove.

If chars is given and not None, remove characters in chars instead.

swapcase()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()

Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper()
Return a copy of the string converted to uppercase.

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.
The string is never truncated.

Subset

class pronto.**Subset**
A definition of a subset in an ontology.

name
The name of the subset, as an OBO short identifier.
Type `str`

description
A description of the subset, as defined in the metadata part of the ontology file.
Type `str`

__eq__(*other: object*) → bool
Return self==value.

__lt__(*other: object*) → bool
Return self<value.

__hash__() → int
Return hash(self).

__ge__(*other, NotImplemented=NotImplemented*)
Return a >= b. Computed by @total_ordering from (not a < b).

__gt__(*other, NotImplemented=NotImplemented*)
Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__le__(*other, NotImplemented=NotImplemented*)
Return a <= b. Computed by @total_ordering from (a < b) or (a == b).

__repr__()
A roundtripping implementation for `repr`. Computed by @roundrepr.

SynonymType

class pronto.**SynonymType**
A user-defined synonym type.

__eq__(*other*)
Return self==value.

__lt__(*other*)
Return self<value.

__hash__()
Return hash(self).

__ge__(*other, NotImplemented=NotImplemented*)
Return a >= b. Computed by @total_ordering from (not a < b).

```

__gt__(other, NotImplemented=NotImplemented)
    Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__le__(other, NotImplemented=NotImplemented)
    Return a <= b. Computed by @total_ordering from (a < b) or (a == b).

__repr__()
    A roundtripping implementation for repr. Computed by @roundrepr.

```

LiteralPropertyValue

class pronto.**LiteralPropertyValue**

A property-value which adds a literal annotation to an entity.

```

__eq__(other: object) → bool
    Return self==value.

__lt__(other: object) → bool
    Return self<value.

__hash__() → int
    Return hash(self).

__ge__(other, NotImplemented=NotImplemented)
    Return a >= b. Computed by @total_ordering from (not a < b).

__gt__(other, NotImplemented=NotImplemented)
    Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__le__(other, NotImplemented=NotImplemented)
    Return a <= b. Computed by @total_ordering from (a < b) or (a == b).

__repr__()
    A roundtripping implementation for repr. Computed by @roundrepr.

```

ResourcePropertyValue

class pronto.**ResourcePropertyValue**

A property-value which adds a resource annotation to an entity.

```

__eq__(other: object) → bool
    Return self==value.

__ge__(other, NotImplemented=NotImplemented)
    Return a >= b. Computed by @total_ordering from (not a < b).

__gt__(other, NotImplemented=NotImplemented)
    Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__le__(other, NotImplemented=NotImplemented)
    Return a <= b. Computed by @total_ordering from (a < b) or (a == b).

__repr__()
    A roundtripping implementation for repr. Computed by @roundrepr.

__lt__(other: object) → bool
    Return self<value.

__hash__() → int
    Return hash(self).

```

Xref

`class pronto.Xref`

A cross-reference to another document or resource.

Cross-references (xrefs for short) can be used to back-up definitions of entities, synonyms, or to link ontological entities to other resources they may have been derived from. Although originally intended to provide links to databases, cross-references in OBO ontologies gained additional purposes, such as helping for header macros expansion, or being used to alias external relationships with local unprefix IDs.

The OBO format version 1.4 expects references to be proper OBO identifiers that can be translated to actual IRIs, which is a breaking change from the previous format. Therefore, cross-references are encouraged to be given as plain IRIs or as prefixed IDs using an ID from the IDspace mapping defined in the header.

Example

A cross-reference in the Mammalian Phenotype ontology linking a term to some related Web resource:

```
>>> mp = pronto.Ontology.from_obo_library("mp.obo")
>>> mp["MP:0030151"].name
'abnormal buccinator muscle morphology'
>>> mp["MP:0030151"].xrefs
frozenset({Xref('https://en.wikipedia.org/wiki/Buccinator_muscle')})
```

Caution: `Xref` instances compare only using their identifiers; this means it is not possible to have several cross-references with the same identifier and different descriptions in the same set.

Todo: Make sure to resolve header macros for xrefs expansion (such as `treat-xrefs-as-is_a`) when creating an ontology, or provide a method on `Ontology` doing so when called.

`__eq__` (*other: object*) → bool
Return self==value.

`__gt__` (*other: object*) → bool
Return self>value.

`__ge__` (*other: object*) → bool
Return self>=value.

`__lt__` (*other: object*) → bool
Return self<value.

`__le__` (*other: object*) → bool
Return self<=value.

`__repr__` ()
A roundtripping implementation for `repr`. Computed by `@roundrepr`.

`__hash__` ()
Return hash(self).

2.3.4 Data Layer

The following classes are from the data layer, and store the data extracted from ontology files. There is probably no point in using them directly, with the exception of custom parser implementations.

<code>pronto.RelationshipData</code>	
<code>pronto.SynonymData</code>	
<code>pronto.TermData</code>	Internal data storage of <i>Term</i> information.

RelationshipData

class `pronto.RelationshipData`

```

__eq__ ()
    Return self==value.

__ge__ ()
    Return self>=value.

__gt__ ()
    Return self>value.

__hash__ ()
    Return hash(self).

__le__ ()
    Return self<=value.

__lt__ ()
    Return self<value.

__repr__ ()
    Return repr(self).
```

SynonymData

class `pronto.SynonymData`

```

__eq__ (other)
    Return self==value.

__lt__ (other)
    Return self<value.

__hash__ ()
    Return hash(self).

__ge__ (other, NotImplemented=NotImplemented)
    Return a >= b. Computed by @total_ordering from (not a < b).

__gt__ (other, NotImplemented=NotImplemented)
    Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__le__ (other, NotImplemented=NotImplemented)
    Return a <= b. Computed by @total_ordering from (a < b) or (a == b).
```

__repr__()
A roundtripping implementation for `repr`. Computed by `@roundrepr`.

TermData

class `pronto.TermData`
Internal data storage of *Term* information.

__eq__()
Return `self==value`.

__ge__()
Return `self>=value`.

__gt__()
Return `self>value`.

__hash__()
Return `hash(self)`.

__le__()
Return `self<=value`.

__lt__()
Return `self<value`.

__repr__()
Return `repr(self)`.

2.3.5 Warnings

Warnings

exception `pronto.warnings.ProntoWarning`
The class for all warnings raised by *pronto*.

exception `pronto.warnings.NotImplementedWarning`
Some part of the code is yet to be implemented.

exception `pronto.warnings.SyntaxWarning(*args, **kwargs)`
The parsed document contains incomplete or unsound constructs.

exception `pronto.warnings.UnstableWarning`
The behaviour of the executed code might change in the future.

<code>pronto.warnings.ProntoWarning</code>	The class for all warnings raised by <i>pronto</i> .
<code>pronto.warnings. NotImplementedWarning</code>	Some part of the code is yet to be implemented.
<code>pronto.warnings.SyntaxWarning</code>	The parsed document contains incomplete or unsound constructs.
<code>pronto.warnings.UnstableWarning</code>	The behaviour of the executed code might change in the future.

2.4 To-do

Todo: Make `Term.objects` take in account `holds_over_chain` and `transitive_over` values of the relationship it is building an iterator with.

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/pronto/checkouts/latest/pronto/term.py:docstring of pronto.Term.objects`, line 11.)

Todo: Make sure to resolve header macros for xrefs expansion (such as `treat-xrefs-as-is_a`) when creating an ontology, or provide a method on `Ontology` doing so when called.

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/pronto/checkouts/latest/pronto/xref.py:docstring of pronto.Xref`, line 33.)

2.5 Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#) and this project adheres to [Semantic Versioning](#).

2.5.1 Unreleased

2.5.2 1.1.3 - 2019-11-10

Fixed

- Handling of some clauses in `FastoboParser`.
- `OboSerializer` occasionally missing lines between term and typedef frames.

Added

- Missing docstrings to some `Entity` properties.

2.5.3 1.1.2 - 2019-10-30

Fixed

- `RdfXMLParser` crashing on entities with `rdf:label` elements without literal content.

2.5.4 1.1.1 - 2019-10-29

Fixed

- `pronto.serializers` module not being embedded in Wheel distribution.

2.5.5 1.1.0 - 2019-10-24

Added

- `Entity.add_synonym` method to create a new synonym and add it to an entity.
- `@roundrepr` now adds a minimal docstring to the generated `__repr__` method.
- `Ontology` caches subclassing relationships to greatly improve performance of `Term.subclasses`.

Changed

- `Entity` subclasses now store their `id` directly to improve performance.
- `Term.subclasses` and `Term.superclasses` use `collections.deque` instead of `queue.Queue` as a LIFO structure since thread-safety is not needed.
- `chardet` result is now used even when prediction confidence is under 100% to detect encoding of the handle passed to `Ontology`.

Fixed

- `SynonymType` comparison implementation.
- `Synonym.type` getter crashing on `type` not being `None`.
- `RdfXMLParser` crashing on `synonymtypedefs` without scope specifiers.

2.5.6 1.0.0 - 2019-10-11

Fixed

- Issues with `typedef` serialization in `FastoboSerializer`.
- `Ontology.create_term` and `Ontology.create_relationship` not raising `ValueError` when given an identifier already in the knowledge graph.
- Signature of `BaseSerializer.dump` to remove encoding argument.
- Missing `__slots__` in `Entity` in non-typechecking runtime.

Changed

- Bumped `fastobo` requirement to `v0.6.0`.

2.5.7 1.0.0-alpha.3 - 2019-10-10

Added

- Extraction of `oboInOwl:consider` annotation in `RdfXMLParser`.
- Extraction of `oboInOwl:savedBy` annotation in `RdfXMLParser`.
- Extraction of `subsetdef` and `synonymtypedef` as annotation properties in `RdfXMLParser`.
- Support for `doap:Version` instead of `owl:VersionIri` for specification of ontology data version.

- Proper comparison of `PropertyValue` classes, based on the lexicographic order of their serialization.
- `Ontology.dump` and `Ontology.dumps` methods to serialize an ontology in **obo** or **obojson** format.

Fixed

- Metadata not storing optional description of ID spaces if any.
- Wrong type hints in `RelationshipData.equivalent_to_chain`.

Changed

- Added type checking to some more property setters.
- Avoid using `networkx` in `Term.subclasses`.
- `fastobo`-derived parsers will not create a new entity if one exists in the graph of dependencies already.
- Exposed `pronto.warnings` and the complete warnings hierarchy.

2.5.8 1.0.0-alpha.2 - 2019-10-03

Added

- Support for extraction of relationships from OWL/XML files to `OwlXMLParser`.

Fixed

- Type hints of `RelationshipData.synonyms` attribute.

2.5.9 1.0.0-alpha.1 - 2019-10-02

Changed

- Dropped support for Python earlier than 3.6.
- Brand new data model that follow the OBO 1.4 object model.
- Partial OWL XML parser implementation using the OBO 1.4 semantics.
- New OBO parser implementation based on `fastobo`.
- Imports are properly separated from the top-level ontology.
- `Ontology.__getitem__` can also access entities from imports.
- `Term`, `Relationship`, `Xref`, `SynonymType` compare only based on their ID.
- `Subset`, `Definition` compare only based on their textual value.

Added

- Support for OBO JSON parser based on `fastobo`.
- Provisional `mypy` type hints.
- Type checking for most properties in `__debug__` mode.
- Proper `repr` implementation that should roundtrip most of the time.
- Detection of file format and encoding based on buffer content.


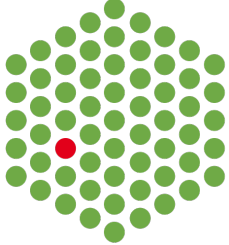
Removed

- OBO and JSON serialization support (for now).
- `Term.rchildren` and `Term.rparents` and stop making direction assumptions on relationships.

2.6 About

2.6.1 Authors

`pronto` is developped and maintained by:

 	<p>Martin Larralde PhD Candidate, EMBL Heidelberg, Germany martin.larralde@embl.de</p>
--	--

2.6.2 Contributors

The following developers contributed both code and time to this project:

- `tatsu` ([@ttyskg](https://github.com/ttyskg))

2.6.3 Reference

If you wish to cite this library, please make sure you are using the latest version of the code, and use the DOI shown on the [Zenodo record](#).

INDICES AND TABLES

- `genindex`
- `search`

PYTHON MODULE INDEX

p

pronto, [9](#)

Symbols

- `__bool__` () (*pronto.Metadata* method), 24
- `__eq__` () (*pronto.Definition* method), 25
- `__eq__` () (*pronto.Entity* method), 13
- `__eq__` () (*pronto.LiteralPropertyValue* method), 31
- `__eq__` () (*pronto.Metadata* method), 24
- `__eq__` () (*pronto.Ontology* method), 11
- `__eq__` () (*pronto.Relationship* method), 15
- `__eq__` () (*pronto.RelationshipData* method), 33
- `__eq__` () (*pronto.ResourcePropertyValue* method), 31
- `__eq__` () (*pronto.Subset* method), 30
- `__eq__` () (*pronto.Synonym* method), 18
- `__eq__` () (*pronto.SynonymData* method), 33
- `__eq__` () (*pronto.SynonymType* method), 30
- `__eq__` () (*pronto.Term* method), 18
- `__eq__` () (*pronto.TermData* method), 34
- `__eq__` () (*pronto.Xref* method), 32
- `__ge__` () (*pronto.Definition* method), 25
- `__ge__` () (*pronto.Entity* method), 13
- `__ge__` () (*pronto.LiteralPropertyValue* method), 31
- `__ge__` () (*pronto.Metadata* method), 24
- `__ge__` () (*pronto.Ontology* method), 11
- `__ge__` () (*pronto.Relationship* method), 15
- `__ge__` () (*pronto.RelationshipData* method), 33
- `__ge__` () (*pronto.ResourcePropertyValue* method), 31
- `__ge__` () (*pronto.Subset* method), 30
- `__ge__` () (*pronto.Synonym* method), 18
- `__ge__` () (*pronto.SynonymData* method), 33
- `__ge__` () (*pronto.SynonymType* method), 30
- `__ge__` () (*pronto.Term* method), 18
- `__ge__` () (*pronto.TermData* method), 34
- `__ge__` () (*pronto.Xref* method), 32
- `__getitem__` () (*pronto.Ontology* method), 11
- `__gt__` () (*pronto.Definition* method), 25
- `__gt__` () (*pronto.Entity* method), 13
- `__gt__` () (*pronto.LiteralPropertyValue* method), 31
- `__gt__` () (*pronto.Metadata* method), 24
- `__gt__` () (*pronto.Ontology* method), 11
- `__gt__` () (*pronto.Relationship* method), 15
- `__gt__` () (*pronto.RelationshipData* method), 33
- `__gt__` () (*pronto.ResourcePropertyValue* method), 31
- `__gt__` () (*pronto.Subset* method), 30
- `__gt__` () (*pronto.Synonym* method), 18
- `__gt__` () (*pronto.SynonymData* method), 33
- `__gt__` () (*pronto.SynonymType* method), 30
- `__gt__` () (*pronto.Term* method), 18
- `__gt__` () (*pronto.TermData* method), 34
- `__gt__` () (*pronto.Xref* method), 32
- `__hash__` () (*pronto.Definition* method), 25
- `__hash__` () (*pronto.Entity* method), 13
- `__hash__` () (*pronto.LiteralPropertyValue* method), 31
- `__hash__` () (*pronto.Metadata* method), 24
- `__hash__` () (*pronto.Relationship* method), 15
- `__hash__` () (*pronto.RelationshipData* method), 33
- `__hash__` () (*pronto.ResourcePropertyValue* method), 31
- `__hash__` () (*pronto.Subset* method), 30
- `__hash__` () (*pronto.Synonym* method), 18
- `__hash__` () (*pronto.SynonymData* method), 33
- `__hash__` () (*pronto.SynonymType* method), 30
- `__hash__` () (*pronto.Term* method), 18
- `__hash__` () (*pronto.TermData* method), 34
- `__hash__` () (*pronto.Xref* method), 32
- `__init__` () (*pronto.Ontology* method), 10
- `__iter__` () (*pronto.Ontology* method), 11
- `__le__` () (*pronto.Definition* method), 25
- `__le__` () (*pronto.Entity* method), 13
- `__le__` () (*pronto.LiteralPropertyValue* method), 31
- `__le__` () (*pronto.Metadata* method), 24
- `__le__` () (*pronto.Ontology* method), 11
- `__le__` () (*pronto.Relationship* method), 15
- `__le__` () (*pronto.RelationshipData* method), 33
- `__le__` () (*pronto.ResourcePropertyValue* method), 31
- `__le__` () (*pronto.Subset* method), 30
- `__le__` () (*pronto.Synonym* method), 18
- `__le__` () (*pronto.SynonymData* method), 33
- `__le__` () (*pronto.SynonymType* method), 31
- `__le__` () (*pronto.Term* method), 18
- `__le__` () (*pronto.TermData* method), 34
- `__le__` () (*pronto.Xref* method), 32
- `__len__` () (*pronto.Ontology* method), 11
- `__lt__` () (*pronto.Definition* method), 25
- `__lt__` () (*pronto.Entity* method), 13
- `__lt__` () (*pronto.LiteralPropertyValue* method), 31

`__lt__()` (*pronto.Metadata method*), 24
`__lt__()` (*pronto.Ontology method*), 11
`__lt__()` (*pronto.Relationship method*), 15
`__lt__()` (*pronto.RelationshipData method*), 33
`__lt__()` (*pronto.ResourcePropertyValue method*), 31
`__lt__()` (*pronto.Subset method*), 30
`__lt__()` (*pronto.Synonym method*), 18
`__lt__()` (*pronto.SynonymData method*), 33
`__lt__()` (*pronto.SynonymType method*), 30
`__lt__()` (*pronto.Term method*), 18
`__lt__()` (*pronto.TermData method*), 34
`__lt__()` (*pronto.Xref method*), 32
`__repr__()` (*pronto.Definition method*), 26
`__repr__()` (*pronto.Entity method*), 13
`__repr__()` (*pronto.LiteralPropertyValue method*), 31
`__repr__()` (*pronto.Metadata method*), 24
`__repr__()` (*pronto.Ontology method*), 11
`__repr__()` (*pronto.Relationship method*), 15
`__repr__()` (*pronto.RelationshipData method*), 33
`__repr__()` (*pronto.ResourcePropertyValue method*), 31
`__repr__()` (*pronto.Subset method*), 30
`__repr__()` (*pronto.Synonym method*), 18
`__repr__()` (*pronto.SynonymData method*), 33
`__repr__()` (*pronto.SynonymType method*), 31
`__repr__()` (*pronto.Term method*), 18
`__repr__()` (*pronto.TermData method*), 34
`__repr__()` (*pronto.Xref method*), 32

A

`add_synonym()` (*pronto.Entity method*), 15
`add_synonym()` (*pronto.Relationship method*), 16
`add_synonym()` (*pronto.Term method*), 19
`alternate_ids()` (*pronto.Entity property*), 13
`alternate_ids()` (*pronto.Relationship property*), 16
`alternate_ids()` (*pronto.Term property*), 19
`annotations` (*pronto.Metadata attribute*), 24
`annotations()` (*pronto.Entity property*), 13
`annotations()` (*pronto.Relationship property*), 16
`annotations()` (*pronto.Term property*), 19
`anonymous()` (*pronto.Entity property*), 13
`anonymous()` (*pronto.Relationship property*), 16
`anonymous()` (*pronto.Term property*), 19
`antisymmetric()` (*pronto.Relationship property*), 17
`asymmetric()` (*pronto.Relationship property*), 18
`auto_generated_by` (*pronto.Metadata attribute*), 23

B

`builtin()` (*pronto.Entity property*), 13
`builtin()` (*pronto.Relationship property*), 16
`builtin()` (*pronto.Term property*), 19

C

`capitalize()` (*pronto.Definition method*), 26

`casefold()` (*pronto.Definition method*), 26
`center()` (*pronto.Definition method*), 26
`comment()` (*pronto.Entity property*), 14
`comment()` (*pronto.Relationship property*), 16
`comment()` (*pronto.Term property*), 19
`count()` (*pronto.Definition method*), 26
`create_relationship()` (*pronto.Ontology method*), 12
`create_term()` (*pronto.Ontology method*), 12
`created_by()` (*pronto.Entity property*), 14
`created_by()` (*pronto.Relationship property*), 16
`created_by()` (*pronto.Term property*), 19
`creation_date()` (*pronto.Entity property*), 14
`creation_date()` (*pronto.Relationship property*), 17
`creation_date()` (*pronto.Term property*), 20

D

`data_version` (*pronto.Metadata attribute*), 23
`date` (*pronto.Metadata attribute*), 23
`default_namespace` (*pronto.Metadata attribute*), 23
`Definition` (*class in pronto*), 24
`definition()` (*pronto.Entity property*), 14
`definition()` (*pronto.Relationship property*), 17
`definition()` (*pronto.Term property*), 20
`description` (*pronto.Subset attribute*), 30
`disjoint_from()` (*pronto.Term property*), 22
`dump()` (*pronto.Ontology method*), 12
`dumps()` (*pronto.Ontology method*), 12

E

`encode()` (*pronto.Definition method*), 26
`endswith()` (*pronto.Definition method*), 26
`Entity` (*class in pronto*), 13
`expandtabs()` (*pronto.Definition method*), 26

F

`find()` (*pronto.Definition method*), 26
`format()` (*pronto.Definition method*), 26
`format_map()` (*pronto.Definition method*), 26
`format_version` (*pronto.Metadata attribute*), 23
`from_obo_library()` (*pronto.Ontology class method*), 10

G

`get()` (*pronto.Ontology method*), 11
`get_relationship()` (*pronto.Ontology method*), 12
`get_term()` (*pronto.Ontology method*), 12

I

`id()` (*pronto.Entity property*), 14
`id()` (*pronto.Relationship property*), 17
`id()` (*pronto.Term property*), 20
`idspace` (*pronto.Metadata attribute*), 24

imports (*pronto.Metadata attribute*), 23
 imports (*pronto.Ontology attribute*), 10
 index() (*pronto.Definition method*), 26
 intersection_of() (*pronto.Term property*), 22
 is_leaf() (*pronto.Term method*), 22
 isalnum() (*pronto.Definition method*), 26
 isalpha() (*pronto.Definition method*), 27
 isascii() (*pronto.Definition method*), 27
 isdecimal() (*pronto.Definition method*), 27
 isdigit() (*pronto.Definition method*), 27
 isidentifier() (*pronto.Definition method*), 27
 islower() (*pronto.Definition method*), 27
 isnumeric() (*pronto.Definition method*), 27
 isprintable() (*pronto.Definition method*), 27
 isspace() (*pronto.Definition method*), 27
 istitle() (*pronto.Definition method*), 27
 isupper() (*pronto.Definition method*), 27
 items() (*pronto.Ontology method*), 11

J

join() (*pronto.Definition method*), 28

K

keys() (*pronto.Ontology method*), 11

L

LiteralPropertyValue (*class in pronto*), 31
 ljust() (*pronto.Definition method*), 28
 lower() (*pronto.Definition method*), 28
 lstrip() (*pronto.Definition method*), 28

M

maketrans() (*pronto.Definition static method*), 28
 Metadata (*class in pronto*), 23
 metadata (*pronto.Ontology attribute*), 9

N

name (*pronto.Subset attribute*), 30
 name() (*pronto.Entity property*), 14
 name() (*pronto.Relationship property*), 17
 name() (*pronto.Term property*), 20
 namespace() (*pronto.Entity property*), 14
 namespace() (*pronto.Relationship property*), 17
 namespace() (*pronto.Term property*), 20
 namespace_id_rule (*pronto.Metadata attribute*), 23
 NotImplementedWarning, 34

O

objects() (*pronto.Term method*), 20
 obsolete() (*pronto.Entity property*), 14
 obsolete() (*pronto.Relationship property*), 17
 obsolete() (*pronto.Term property*), 20
 Ontology (*class in pronto*), 9

ontology (*pronto.Metadata attribute*), 23
 owl_axioms (*pronto.Metadata attribute*), 23

P

partition() (*pronto.Definition method*), 28
 pronto (*module*), 9
 ProntoWarning, 34

R

Relationship (*class in pronto*), 15
 RelationshipData (*class in pronto*), 33
 relationships() (*pronto.Ontology method*), 12
 remarks (*pronto.Metadata attribute*), 24
 replace() (*pronto.Definition method*), 28
 ResourcePropertyValue (*class in pronto*), 31
 rfind() (*pronto.Definition method*), 28
 rindex() (*pronto.Definition method*), 28
 rjust() (*pronto.Definition method*), 28
 rpartition() (*pronto.Definition method*), 28
 rsplit() (*pronto.Definition method*), 29
 rstrip() (*pronto.Definition method*), 29

S

saved_by (*pronto.Metadata attribute*), 23
 split() (*pronto.Definition method*), 29
 splitlines() (*pronto.Definition method*), 29
 startswith() (*pronto.Definition method*), 29
 strip() (*pronto.Definition method*), 29
 subclasses() (*pronto.Term method*), 21
 Subset (*class in pronto*), 30
 subsetdefs (*pronto.Metadata attribute*), 23
 subsets() (*pronto.Entity property*), 14
 subsets() (*pronto.Relationship property*), 17
 subsets() (*pronto.Term property*), 20
 superclasses() (*pronto.Term method*), 21
 swapcase() (*pronto.Definition method*), 29
 Synonym (*class in pronto*), 18
 SynonymData (*class in pronto*), 33
 synonyms() (*pronto.Entity property*), 14
 synonyms() (*pronto.Relationship property*), 17
 synonyms() (*pronto.Term property*), 20
 SynonymType (*class in pronto*), 30
 synonymtypedefs (*pronto.Metadata attribute*), 24
 SyntaxWarning, 34

T

Term (*class in pronto*), 18
 TermData (*class in pronto*), 34
 terms() (*pronto.Ontology method*), 12
 timeout (*pronto.Ontology attribute*), 9
 title() (*pronto.Definition method*), 29
 translate() (*pronto.Definition method*), 29

U

`unreserved` (*pronto.Metadata attribute*), [24](#)

`UnstableWarning`, [34](#)

`upper()` (*pronto.Definition method*), [29](#)

V

`values()` (*pronto.Ontology method*), [11](#)

X

`Xref` (*class in pronto*), [32](#)

`xrefs()` (*pronto.Entity property*), [15](#)

`xrefs()` (*pronto.Relationship property*), [17](#)

`xrefs()` (*pronto.Term property*), [20](#)

Z

`zfill()` (*pronto.Definition method*), [30](#)