zope.schema Documentation

Release 4.9.4.dev0

Zope Foundation Contributors

Oct 18, 2018

Contents

Contents:

CHAPTER 1

Zope 3 Schemas

1.1 Introduction

This package is intended to be independently reusable in any Python project. It is maintained by the Zope Toolkit project.

Schemas extend the notion of interfaces to detailed descriptions of Attributes (but not methods). Every schema is an interface and specifies the public fields of an object. A *field* roughly corresponds to an attribute of a python object. But a Field provides space for at least a title and a description. It can also constrain its value and provide a validation method. Besides you can optionally specify characteristics such as its value being read-only or not required.

Zope 3 schemas were born when Jim Fulton and Martijn Faassen thought about Formulator for Zope 3 and `PropertySets` while at the Zope 3 sprint at the Zope BBQ in Berlin. They realized that if you strip all view logic from forms then you have something similar to interfaces. And thus schemas were born.

1.2 Simple Usage

Let's have a look at a simple example. First we write an interface as usual, but instead of describing the attributes of the interface with `Attribute` instances, we now use schema fields:

```
>>> import zope.interface
>>> import zope.schema

>>> class IBookmark(zope.interface.Interface):
...     title = zope.schema.TextLine(
...         title=u'Title',
...         description=u'The title of the bookmark',
...         required=True)
...
...     url = zope.schema.URI(
...         title=u'Bookmark URL',
...         description=u'URL of the Bookmark',
```

(continues on next page)

(continued from previous page)

```
...         required=True)
...
```

Now we create a class that implements this interface and create an instance of it:

```
>>> @zope.interface.implementer(IBookmark)
... class Bookmark(object):
...
...     title = None
...     url = None

>>> bm = Bookmark()
```

We would now like to only add validated values to the class. This can be done by first validating and then setting the value on the object. The first step is to define some data:

```
>>> title = u'Zope 3 Website'
>>> url = 'http://dev.zope.org/Zope3'
```

Now we, get the fields from the interface:

```
>>> title_field = IBookmark.get('title')
>>> url_field = IBookmark.get('url')
```

Next we have to bind these fields to the context, so that instance-specific information can be used for validation:

```
>>> title_bound = title_field.bind(bm)
>>> url_bound = url_field.bind(bm)
```

Now that the fields are bound, we can finally validate the data:

```
>>> title_bound.validate(title)
>>> url_bound.validate(url)
```

If the validation is successful, `None` is returned. If a validation error occurs a `ValidationError` will be raised; for example:

```
>>> from zope.schema._compat import non_native_string
>>> url_bound.validate(non_native_string('http://zope.org/foo'))
Traceback (most recent call last):
...
WrongType: ...

>>> url_bound.validate('foo.bar')
Traceback (most recent call last):
...
InvalidURI: foo.bar
```

Now that the data has been successfully validated, we can set it on the object:

```
>>> title_bound.set(bm, title)
>>> url_bound.set(bm, url)
```

That's it. You still might think this is a lot of work to validate and set a value for an object. Note, however, that it is very easy to write helper functions that automate these tasks. If correctly designed, you will never have to worry explicitly about validation again, since the system takes care of it automatically.

1.3 What is a schema, how does it compare to an interface?

A schema is an extended interface which defines fields. You can validate that the attributes of an object conform to their fields defined on the schema. With plain interfaces you can only validate that methods conform to their interface specification.

So interfaces and schemas refer to different aspects of an object (respectively its code and state).

A schema starts out like an interface but defines certain fields to which an object's attributes must conform. Let's look at a stripped down example from the programmer's tutorial:

```
>>> import re

>>> class IContact(zope.interface.Interface):
...     """Provides access to basic contact information."""
...
...     first = zope.schema.TextLine(title=u"First name")
...
...     last = zope.schema.TextLine(title=u"Last name")
...
...     email = zope.schema.TextLine(title=u"Electronic mail address")
...
...     address = zope.schema.Text(title=u"Postal address")
...
...     postalCode = zope.schema.TextLine(
...         title=u"Postal code",
...         constraint=re.compile("\d{5,5}(-\d{4,4})?$").match)
```

`TextLine` is a field and expresses that an attribute is a single line of Unicode text. `Text` expresses an arbitrary Unicode ("text") object. The most interesting part is the last attribute specification. It constrains the `postalCode` attribute to only have values that are US postal codes.

Now we want a class that adheres to the `IContact` schema:

```
>>> @zope.interface.implementer(IContact)
... class Contact(object):
...
...     def __init__(self, first, last, email, address, pc):
...         self.first = first
...         self.last = last
...         self.email = email
...         self.address = address
...         self.postalCode = pc
```

Now you can see if an instance of `Contact` actually implements the schema:

```
>>> someone = Contact(u'Tim', u'Roberts', u'tim@roberts', u'',
...                   u'12032-3492')

>>> for field in zope.schema.getFields(IContact).values():
...     bound = field.bind(someone)
...     bound.validate(bound.get(someone))
```

1.4 Data Modeling Concepts

The `zope.schema` package provides a core set of field types, including single- and multi-line text fields, binary data fields, integers, floating-point numbers, and date/time values.

Selection issues; field type can specify:

- "Raw" data value

  Simple values not constrained by a selection list.

- Value from enumeration (options provided by schema)

  This models a single selection from a list of possible values specified by the schema. The selection list is expected to be the same for all values of the type. Changes to the list are driven by schema evolution.

  This is done by mixing-in the `IEnumerated` interface into the field type, and the Enumerated mix-in for the implementation (or emulating it in a concrete class).

- Value from selection list (options provided by an object)

  This models a single selection from a list of possible values specified by a source outside the schema. The selection list depends entirely on the source of the list, and may vary over time and from object to object. Changes to the list are not related to the schema, but changing how the list is determined is based on schema evolution.

  There is not currently a spelling of this, but it could be facilitated using alternate mix-ins similar to IEnumerated and Enumerated.

- Whether or not the field is read-only

  If a field value is read-only, it cannot be changed once the object is created.

- Whether or not the field is required

  If a field is designated as required, assigned field values must always be non-missing. See the next section for a description of missing values.

- A value designated as `missing`

  Missing values, when assigned to an object, indicate that there is 'no data' for that field. Missing values are analogous to null values in relational databases. For example, a boolean value can be True, False, or missing, in which case its value is unknown.

  While Python's None is the most likely value to signify 'missing', some fields may use different values. For example, it is common for text fields to use the empty string ('') to signify that a value is missing. Numeric fields may use 0 or -1 instead of None as their missing value.

  A field that is 'required' signifies that missing values are invalid and should not be assigned.

- A default value

  Default field values are assigned to objects when they are first created. A default factory can be specified to dynamically compute default values.

1.5 Fields and Widgets

Widgets are components that display field values and, in the case of writable fields, allow the user to edit those values.

Widgets:

- Display current field values, either in a read-only format, or in a format that lets the user change the field value.

- Update their corresponding field values based on values provided by users.
- Manage the relationships between their representation of a field value and the object's field value. For example, a widget responsible for editing a number will likely represent that number internally as a string. For this reason, widgets must be able to convert between the two value formats. In the case of the number-editing widget, string values typed by the user need to be converted to numbers such as int or float.
- Support the ability to assign a missing value to a field. For example, a widget may present a `None` option for selection that, when selected, indicates that the object should be updated with the field's `missing` value.

1.6 References

- Use case list, http://dev.zope.org/Zope3/Zope3SchemasUseCases
- Documented interfaces, zope/schema/interfaces.py
- Jim Fulton's Programmers Tutorial; in CVS: Docs/ZopeComponentArchitecture/PythonProgrammerTutorial/Chapter2

CHAPTER 2

Fields

This document highlights unusual and subtle aspects of various fields and field classes, and is not intended to be a general introduction to schema fields. Please see README.txt for a more general introduction.

While many field types, such as Int, TextLine, Text, and Bool are relatively straightforward, a few have some subtlety. We will explore the general class of collections and discuss how to create a custom creation field; discuss Choice fields, vocabularies, and their use with collections; and close with a look at the standard zope.app approach to using these fields to find views ("widgets").

2.1 Scalars

Scalar fields represent simple. immutable Python types.

2.1.1 Bytes

zope.schema.Bytes fields contain binary data, represented as a sequence of bytes (`str` in Python2, `bytes` in Python3).

Conversion from Unicode:

```
>>> from zope.schema import Bytes
>>> obj = Bytes(constraint=lambda v: b'x' in v)
>>> result = obj.fromUnicode(u" foo x.y.z bat")
>>> isinstance(result, bytes)
True
>>> str(result.decode("ascii"))
' foo x.y.z bat'
>>> obj.fromUnicode(u" foo y.z bat")
Traceback (most recent call last):
...
ConstraintNotSatisfied:  foo y.z bat
```

2.1.2 ASCII

zope.schema.ASCII fields are a restricted form of *zope.schema.Bytes*: they can contain only 7-bit bytes.

Validation accepts empty strings:

```
>>> from zope.schema import ASCII
>>> ascii = ASCII()
>>> empty = ''
>>> ascii._validate(empty)
```

and all kinds of alphanumeric strings:

```
>>> alphanumeric = "Bob\'s my 23rd uncle"
>>> ascii._validate(alphanumeric)
```

but fails with 8-bit (encoded) strings:

```
>>> umlauts = "Köhlerstraße"
>>> ascii._validate(umlauts)
Traceback (most recent call last):
...
InvalidValue
```

2.1.3 BytesLine

zope.schema.BytesLine fields are a restricted form of *zope.schema.Bytes*: they cannot contain newlines.

2.1.4 ASCIILine

zope.schema.BytesLine fields are a restricted form of *zope.schema.ASCII*: they cannot contain newlines.

2.1.5 Float

zope.schema.Float fields contain binary data, represented as a a Python `float`.

Conversion from Unicode:

```
>>> from zope.schema import Float
>>> f = Float()
>>> f.fromUnicode("1.25")
1.25
>>> f.fromUnicode("1.25.6")
Traceback (most recent call last):
...
InvalidFloatLiteral: invalid literal for float(): 1.25.6
```

2.1.6 Int

zope.schema.Int fields contain binary data, represented as a a Python `int`.

Conversion from Unicode:

```
>>> from zope.schema import Int
>>> f = Int()
>>> f.fromUnicode("1")
1
>>> f.fromUnicode("1.25.6")
Traceback (most recent call last):
...
InvalidIntLiteral: invalid literal for int() with base 10: 1.25.6
```

2.1.7 Decimal

zope.schema.Decimal fields contain binary data, represented as a a Python decimal.Decimal.

Conversion from Unicode:

```
>>> from zope.schema import Decimal
>>> f = Decimal()
>>> import decimal
>>> isinstance(f.fromUnicode("1.25"), decimal.Decimal)
True
>>> float(f.fromUnicode("1.25"))
1.25
>>> f.fromUnicode("1.25.6")
Traceback (most recent call last):
...
InvalidDecimalLiteral: invalid literal for Decimal(): 1.25.6
```

2.1.8 Datetime

zope.schema.Datetime fields contain binary data, represented as a a Python datetime.datetime.

2.1.9 Date

zope.schema.Date fields contain binary data, represented as a a Python datetime.date.

2.1.10 TimeDelta

zope.schema.TimeDelta fields contain binary data, represented as a a Python datetime.timedelta.

2.1.11 Time

zope.schema.Time fields contain binary data, represented as a a Python datetime.time.

2.1.12 Choice

zope.schema.Choice fields are constrained to values drawn from a specified set, which can be static or dynamic.

Conversion from Unicode enforces the constraint:

```
>>> from zope.schema.interfaces import IFromUnicode
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> from zope.schema import Choice
>>> t = Choice(
...     vocabulary=SimpleVocabulary.fromValues([u'foo',u'bar']))
>>> IFromUnicode.providedBy(t)
True
>>> t.fromUnicode(u"baz")
Traceback (most recent call last):
...
ConstraintNotSatisfied: baz
>>> result = t.fromUnicode(u"foo")
>>> isinstance(result, bytes)
False
>>> print(result)
foo
```

By default, ValueErrors are thrown if duplicate values or tokens are passed in. If you are using this vocabulary as part of a form that is generated from non-pristine data, this may not be the desired behavior. If you want to swallow these exceptions, pass in swallow_duplicates=True when initializing the vocabulary. See the test cases for an example.

2.1.13 URI

zope.schema.URI fields contain native Python strings (`str`), matching the "scheme:data" pattern.

Validation ensures that the pattern is matched:

```
>>> from zope.schema import URI
>>> uri = URI(__name__='test')
>>> uri.validate("http://www.python.org/foo/bar")
>>> uri.validate("DAV:")
>>> uri.validate("www.python.org/foo/bar")
Traceback (most recent call last):
...
InvalidURI: www.python.org/foo/bar
```

Conversion from Unicode:

```
>>> uri = URI(__name__='test')
>>> uri.fromUnicode("http://www.python.org/foo/bar")
'http://www.python.org/foo/bar'
>>> uri.fromUnicode("          http://www.python.org/foo/bar")
'http://www.python.org/foo/bar'
>>> uri.fromUnicode("      \n    http://www.python.org/foo/bar\n")
'http://www.python.org/foo/bar'
>>> uri.fromUnicode("http://www.python.org/ foo/bar")
Traceback (most recent call last):
...
InvalidURI: http://www.python.org/ foo/bar
```

2.1.14 DottedName

zope.schema.DottedName fields contain native Python strings (`str`), containing zero or more "dots" separating elements of the name. The minimum and maximum number of dots can be passed to the constructor:

```
>>> from zope.schema import DottedName
>>> DottedName(min_dots=-1)
Traceback (most recent call last):
...
ValueError: min_dots cannot be less than zero

>>> DottedName(max_dots=-1)
Traceback (most recent call last):
...
ValueError: max_dots cannot be less than min_dots

>>> DottedName(max_dots=1, min_dots=2)
Traceback (most recent call last):
...
ValueError: max_dots cannot be less than min_dots

>>> dotted_name = DottedName(max_dots=1, min_dots=1)

>>> from zope.interface.verify import verifyObject
>>> from zope.schema.interfaces import IDottedName
>>> verifyObject(IDottedName, dotted_name)
True

>>> dotted_name = DottedName(max_dots=1)
>>> dotted_name.min_dots
0

>>> dotted_name = DottedName(min_dots=1)
>>> dotted_name.max_dots
>>> dotted_name.min_dots
1
```

Validation ensures that the pattern is matched:

```
>>> dotted_name = DottedName(__name__='test')
>>> dotted_name.validate("a.b.c")
>>> dotted_name.validate("a")
>>> dotted_name.validate("   a")
Traceback (most recent call last):
...
InvalidDottedName:    a

>>> dotted_name = DottedName(__name__='test', min_dots=1)
>>> dotted_name.validate('a.b')
>>> dotted_name.validate('a.b.c.d')
>>> dotted_name.validate('a')
Traceback (most recent call last):
...
InvalidDottedName: ('too few dots; 1 required', 'a')

>>> dotted_name = DottedName(__name__='test', max_dots=0)
>>> dotted_name.validate('a')
>>> dotted_name.validate('a.b')
Traceback (most recent call last):
...
InvalidDottedName: ('too many dots; no more than 0 allowed', 'a.b')
```

(continues on next page)

(continued from previous page)

```
>>> dotted_name = DottedName(__name__='test', max_dots=2)
>>> dotted_name.validate('a')
>>> dotted_name.validate('a.b')
>>> dotted_name.validate('a.b.c')
>>> dotted_name.validate('a.b.c.d')
Traceback (most recent call last):
...
InvalidDottedName: ('too many dots; no more than 2 allowed', 'a.b.c.d')

>>> dotted_name = DottedName(__name__='test', max_dots=1, min_dots=1)
>>> dotted_name.validate('a.b')
>>> dotted_name.validate('a')
Traceback (most recent call last):
...
InvalidDottedName: ('too few dots; 1 required', 'a')
>>> dotted_name.validate('a.b.c')
Traceback (most recent call last):
...
InvalidDottedName: ('too many dots; no more than 1 allowed', 'a.b.c')
```

Id

zope.schema.Id fields contain native Python strings (`str`), matching either the URI pattern or a "dotted name".

Validation ensures that the pattern is matched:

```
>>> from zope.schema import Id
>>> id = Id(__name__='test')
>>> id.validate("http://www.python.org/foo/bar")
>>> id.validate("zope.app.content")
>>> id.validate("zope.app.content/a")
Traceback (most recent call last):
...
InvalidId: zope.app.content/a
>>> id.validate("http://zope.app.content x y")
Traceback (most recent call last):
...
InvalidId: http://zope.app.content x y
```

Conversion from Unicode:

```
>>> id = Id(__name__='test')
>>> id.fromUnicode("http://www.python.org/foo/bar")
'http://www.python.org/foo/bar'
>>> id.fromUnicode(u" http://www.python.org/foo/bar ")
'http://www.python.org/foo/bar'
>>> id.fromUnicode("http://www.python.org/ foo/bar")
Traceback (most recent call last):
...
InvalidId: http://www.python.org/ foo/bar
>>> id.fromUnicode("      \n x.y.z \n")
'x.y.z'
```

2.2 Collections

Normal fields typically describe the API of the attribute – does it behave as a Python Int, or a Float, or a Bool – and various constraints to the model, such as a maximum or minimum value. Collection fields have additional requirements because they contain other types, which may also be described and constrained.

For instance, imagine a list that contains non-negative floats and enforces uniqueness. In a schema, this might be written as follows:

```
>>> from zope.interface import Interface
>>> from zope.schema import List, Float
>>> class IInventoryItem(Interface):
...     pricePoints = List(
...         title=u"Price Points",
...         unique=True,
...         value_type=Float(title=u"Price", min=0.0)
...     )
```

This indicates several things.

- pricePoints is an attribute of objects that implement IInventoryItem.
- The contents of pricePoints can be accessed and manipulated via a Python list API.
- Each member of pricePoints must be a non-negative float.
- Members cannot be duplicated within pricePoints: each must be must be unique.
- The attribute and its contents have descriptive titles. Typically these would be message ids.

This declaration creates a field that implements a number of interfaces, among them these:

```
>>> from zope.schema.interfaces import IList, ISequence, ICollection
>>> IList.providedBy(IInventoryItem['pricePoints'])
True
>>> ISequence.providedBy(IInventoryItem['pricePoints'])
True
>>> ICollection.providedBy(IInventoryItem['pricePoints'])
True
```

2.3 Creating a custom collection field

Ideally, custom collection fields have interfaces that inherit appropriately from either zope.schema.interfaces.ISequence or zope.schema.interfaces.IUnorderedCollection. Most collection fields should be able to subclass *zope.schema._field.AbstractCollection* to get the necessary behavior. Notice the behavior of the Set field in zope.schema: this would also be necessary to implement a Bag.

2.4 Choices and Vocabularies

Choice fields are the schema way of spelling enumerated fields and more. By providing a dynamically generated list of options, the choices available to a choice field can be contextually calculated.

Simple choices can directly specify the values they accept:

```
>>> from zope.schema import Choice
>>> f = Choice((640, 1028, 1600))
>>> f.validate(640)
>>> f.validate(960)
Traceback (most recent call last):
...
ConstraintNotSatisfied: 960
>>> f.validate('bing')
Traceback (most recent call last):
...
ConstraintNotSatisfied: bing
```

More complex choices will want to use *vocabularies*, possibly created from a contextual *vocabulary factory*; this factory can either be directly provided at construction time or *named* and looked up in a registry at binding or validation time. Vocabularies have a simple interface, as defined in *zope.schema.interfaces.IBaseVocabulary*. A vocabulary must minimally be able to determine whether it contains a value, to create a term object for a value, and to return a query interface (or None) to find items in itself. Term objects are an abstraction that wraps a vocabulary value.

Many applications that deal with accepting user input and validating it against a choice may need a fuller vocabulary interface that provides "tokens" on its terms: ASCII values that have a one-to-one relationship to the values when the vocabulary is asked to "getTermByToken". If a vocabulary is small, it can also support the *zope.schema.interfaces.IIterableVocabulary* interface.

zope.schema.vocabulary.SimpleVocabulary is a vocabulary implementation that may do all you need for many simple tasks. The vocabulary interface is simple enough that writing a custom vocabulary is not too difficult itself.

See *zope.schema.vocabulary.TreeVocabulary* for another `IBaseVocabulary` supporting vocabulary that provides a nested, tree-like structure.

2.4.1 Vocabulary Factories

Sometimes the values for a choice really are dynamic. For example, they might depend on the context object being validated. In that case, we can provide an object that provides *zope.schema.interfaces.IContextSourceBinder* as the `source` parameter. When the Choice needs a vocabulary, it will call the `IContextSourceBinder`, passing in its context. This could be as simple as a function:

```
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> from zope.schema.interfaces import IContextSourceBinder
>>> from zope.interface import directlyProvides
>>> def myDynamicVocabulary(context):
...     v = range(context)
...     return SimpleVocabulary.fromValues(v)
>>> directlyProvides(myDynamicVocabulary, IContextSourceBinder)

>>> f = Choice(source=myDynamicVocabulary)
```

Note that the source is only invoked for fields that have been bound to a context:

```
>>> f.validate(1)
Traceback (most recent call last):
...
InvalidVocabularyError: Invalid vocabulary <function myDynamicVocabulary at␣
↪0x1101f7730>
>>> f = f.bind(3)
>>> f.validate(1)
```

(continues on next page)

(continued from previous page)

```
>>> f.validate(2)
>>> f.validate(3)
Traceback (most recent call last):
...
ConstraintNotSatisfied: 3
```

2.4.2 Named (Registered) Vocabularies

We can also provide a vocabulary name that will be resolved later against a registry of vocabulary factories (objects that implement *zope.schema.interfaces.IVocabularyFactory*). On the surface, this looks very similar to providing a `source` argument: they are both callable objects that take a context and return a vocabulary. The advantage of a named factory is a level of indirection, allowing the same name to be easily used in many different fields, even from packages that aren't aware of each other. For example, an application framework may define choices that use a 'permissions' vocabulary, and individual applications may define their own meaning for that name.

A simple version of this is provided in this package using a global vocabulary registry:

```
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> from zope.schema.vocabulary import getVocabularyRegistry
>>> from zope.schema.interfaces import IVocabularyFactory
>>> from zope.interface import implementer
>>> @implementer(IVocabularyFactory)
... class PermissionsVocabulary(object):
...
...     def __call__(self, context):
...         if context is None: raise AttributeError
...         return SimpleVocabulary.fromValues(context.possible_permissions)
>>> getVocabularyRegistry().register('permissions', PermissionsVocabulary())

>>> class Context(object):
...     possible_permissions = ('read', 'write')
```

Unlike `IContextSourceBinder`, the factory is invoked even for unbound fields; depending on the factory, this may or may not do anything useful (our factory produces errors):

```
>>> f = Choice(vocabulary='permissions')
>>> f.validate('read')
Traceback (most recent call last):
...
AttributeError
>>> context = Context()
>>> f = f.bind(context)
>>> f.validate("read")
>>> f.validate("write")
>>> f.validate("delete")
Traceback (most recent call last):
...
ConstraintNotSatisfied: ('delete', '')
```

The zope.vocabularyregistry package provides a registry that keeps factories as named utilities in the Zope component architecture. This is especially useful when combined with the concept of multiple component site managers, as that provides another layer of indirection.

2.5 Choices and Collections

Choices are a field type and can be used as a `value_type` for collections. Just as a collection of an "Int" value_type constrains members to integers, so a choice-based value type constrains members to choices within the Choice's vocabulary. Typically in the Zope application server widgets are found not only for the collection and the choice field but also for the vocabulary on which the choice is based.

2.6 Using Choice and Collection Fields within a Widget Framework

While fields support several use cases, including code documentation and data description and even casting, a significant use case influencing their design is to support form generation – generating widgets for a field. Choice and collection fields are expected to be used within widget frameworks. The zope.app approach typically (but configurably) uses multiple dispatches to find widgets on the basis of various aspects of the fields.

Widgets for all fields are found by looking up a browser view of the field providing an input or display widget view. Typically there is only a single "widget" registered for Choice fields. When it is looked up, it performs another dispatch – another lookup – for a widget registered for both the field and the vocabulary. This widget typically has enough information to render without a third dispatch.

Collection fields may fire several dispatches. The first is the usual lookup by field. A single "widget" should be registered for ICollection, which does a second lookup by field and value_type constraint, if any, or, theoretically, if value_type is None, renders some absolutely generic collection widget that allows input of any value imaginable: a check-in of such a widget would be unexpected. This second lookup may find a widget that knows how to render, and stop. However, the value_type may be a choice, which will usually fire a third dispatch: a search for a browser widget for the collection field, the value_type field, and the vocabulary. Further lookups may even be configured on the basis of uniqueness and other constraints.

This level of indirection may be unnecessary for some applications, and can be disabled with simple ZCML changes within `zope.app`.

CHAPTER 3

Sources

3.1 Concepts

Sources are designed with three concepts:

- The source itself - an iterable

  This can return any kind of object it wants. It doesn't have to care for browser representation, encoding, ...

- A way to map a value from the iterable to something that can be used for form *values* - this is called a token. A token is commonly a (unique) 7bit representation of the value.
- A way to map a value to something that can be displayed to the user - this is called a title

The last two elements are dispatched using a so called `term`. The ITitledTokenizedTerm interface contains a triple of (value, token, title).

Additionally there are some lookup functions to perform the mapping between values and terms and tokens and terms.

Sources that require context use a special factory: a context source binder that is called with the context and instanciates the source when it is actually used.

3.2 Sources in Fields

A choice field can be constructed with a source or source name. When a source is used, it will be used as the source for valid values.

Create a source for all odd numbers.

```
>>> from zope import interface
>>> from zope.schema.interfaces import ISource, IContextSourceBinder
>>> @interface.implementer(ISource)
... class MySource(object):
...     divisor = 2
```

(continues on next page)

(continued from previous page)

```
...     def __contains__(self, value):
...         return bool(value % self.divisor)
>>> my_source = MySource()
>>> 1 in my_source
True
>>> 2 in my_source
False

>>> from zope.schema import Choice
>>> choice = Choice(__name__='number', source=my_source)
>>> bound = choice.bind(object())
>>> bound.vocabulary
<...MySource...>
```

If a IContextSourceBinder is passed as the `source` argument to Choice, it's `bind` method will be called with the context as its only argument. The result must implement ISource and will be used as the source.

```
>>> _my_binder_called = []
>>> def my_binder(context):
...     _my_binder_called.append(context)
...     source = MySource()
...     source.divisor = context.divisor
...     return source
>>> interface.directlyProvides(my_binder, IContextSourceBinder)

>>> class Context(object):
...     divisor = 3

>>> choice = Choice(__name__='number', source=my_binder)
>>> bound = choice.bind(Context())
>>> len(_my_binder_called)
1
>>> bound.vocabulary
<...MySource...>
>>> bound.vocabulary.divisor
3
```

When using IContextSourceBinder together with default value, it's impossible to validate it on field initialization. Let's check if initalization doesn't fail in that case.

```
>>> choice = Choice(__name__='number', source=my_binder, default=2)

>>> del _my_binder_called[:]
>>> bound = choice.bind(Context())
>>> len(_my_binder_called)
1

>>> bound.validate(bound.default)
>>> bound.validate(3)
Traceback (most recent call last):
...
ConstraintNotSatisfied: 3
```

It's developer's responsibility to provide a default value that fits the constraints when using context-based sources.

CHAPTER 4

Schema Validation

There are two helper methods to verify schemas and interfaces:

`zope.schema.`**`getValidationErrors`**(*schema*, *value*)
Validate that *value* conforms to the schema interface *schema*.

This includes checking for any schema validation errors (using *getSchemaValidationErrors*). If that succeeds, then we proceed to check for any declared invariants.

Note that this does not include a check to see if the *value* actually provides the given *schema*.

Returns A sequence of (name, `zope.interface.Invalid`) tuples, where *name* is None if the error was from an invariant. If the sequence is empty, there were no errors.

`zope.schema.`**`getSchemaValidationErrors`**(*schema*, *value*)
Validate that *value* conforms to the schema interface *schema*.

All *zope.schema.interfaces.IField* members of the *schema* are validated after being bound to *value*. (Note that we do not check for arbitrary `zope.interface.Attribute` members being present.)

Returns A sequence of (name, *ValidationError*) tuples. A non-empty sequence indicates validation failed.

Invariants are documented by zope.interface.

Create an interface to validate against:

```
>>> import zope.interface
>>> import zope.schema
>>> _a_greater_b_called = []
>>> class ITwoInts(zope.interface.Interface):
...     a = zope.schema.Int(max=10)
...     b = zope.schema.Int(min=5)
...
...     @zope.interface.invariant
...     def a_greater_b(obj):
...         _a_greater_b_called.append(obj)
...         if obj.a <= obj.b:
```

(continues on next page)

(continued from previous page)

```
...             raise zope.interface.Invalid("%s<=%s" % (obj.a, obj.b))
...
```

Create a silly model:

```
>>> class TwoInts(object):
...     pass
```

Create an instance of TwoInts but do not set attributes. We get two errors:

```
>>> ti = TwoInts()
>>> r = zope.schema.getValidationErrors(ITwoInts, ti)
>>> r.sort()
>>> len(r)
2
>>> r[0][0]
'a'
>>> r[0][1].__class__.__name__
'SchemaNotFullyImplemented'
>>> r[0][1].args[0].args
("'TwoInts' object has no attribute 'a'",)
>>> r[1][0]
'b'
>>> r[1][1].__class__.__name__
'SchemaNotFullyImplemented'
>>> r[1][1].args[0].args
("'TwoInts' object has no attribute 'b'",)
```

The `getSchemaValidationErrors` function returns the same result:

```
>>> r = zope.schema.getSchemaValidationErrors(ITwoInts, ti)
>>> r.sort()
>>> len(r)
2
>>> r[0][0]
'a'
>>> r[0][1].__class__.__name__
'SchemaNotFullyImplemented'
>>> r[0][1].args[0].args
("'TwoInts' object has no attribute 'a'",)
>>> r[1][0]
'b'
>>> r[1][1].__class__.__name__
'SchemaNotFullyImplemented'
>>> r[1][1].args[0].args
("'TwoInts' object has no attribute 'b'",)
```

Note that see no error from the invariant because the invariants are not validated if there are other schema errors.

When we set a valid value for `a` we still get the same error for `b`:

```
>>> ti.a = 11
>>> errors = zope.schema.getValidationErrors(ITwoInts, ti)
>>> errors.sort()
>>> len(errors)
2
>>> errors[0][0]
```

(continues on next page)

(continued from previous page)

```
'a'
>>> print(errors[0][1].doc())
Value is too big
>>> errors[0][1].__class__.__name__
'TooBig'
>>> errors[0][1].args
(11, 10)
>>> errors[1][0]
'b'
>>> errors[1][1].__class__.__name__
'SchemaNotFullyImplemented'
>>> errors[1][1].args[0].args
("'TwoInts' object has no attribute 'b'",)
```

After setting a valid value for `a` there is only the error for the missing `b` left:

```
>>> ti.a = 8
>>> r = zope.schema.getValidationErrors(ITwoInts, ti)
>>> r
[('b', SchemaNotFullyImplemented(...AttributeError...))]
>>> r[0][1].args[0].args
("'TwoInts' object has no attribute 'b'",)
```

After setting valid value for `b` the schema is valid so the invariants are checked. As `b>a` the invariant fails:

```
>>> ti.b = 10
>>> errors = zope.schema.getValidationErrors(ITwoInts, ti)
>>> len(errors)
1
>>> errors[0][0] is None
True
>>> errors[0][1].__class__.__name__
'Invalid'
>>> len(_a_greater_b_called)
1
```

When using `getSchemaValidationErrors` we do not get an error any more:

```
>>> zope.schema.getSchemaValidationErrors(ITwoInts, ti)
[]
```

Set `b=5` so everything is fine:

```
>>> ti.b = 5
>>> del _a_greater_b_called[:]
>>> zope.schema.getValidationErrors(ITwoInts, ti)
[]
>>> len(_a_greater_b_called)
1
```

4.1 Compare ValidationError

There was an issue with compare validation error with something else then an exceptions. Let's test if we can compare ValidationErrors with different things

```
>>> from zope.schema._bootstrapinterfaces import ValidationError
>>> v1 = ValidationError('one')
>>> v2 = ValidationError('one')
>>> v3 = ValidationError('another one')
```

A ValidationError with the same arguments compares:

```
>>> v1 == v2
True
```

but not with an error with different arguments:

```
>>> v1 == v3
False
```

We can also compare validation errors with other things then errors. This was running into an AttributeError in previous versions of zope.schema. e.g. AttributeError: 'NoneType' object has no attribute 'args'

```
>>> v1 == None
False
>>> v1 == object()
False
>>> v1 == False
False
>>> v1 == True
False
>>> v1 == 0
False
>>> v1 == 1
False
>>> v1 == int
False
```

If we compare a ValidationError with another validation error based class, we will get the following result:

```
>>> from zope.schema._bootstrapinterfaces import RequiredMissing
>>> r1 = RequiredMissing('one')
>>> v1 == r1
True
```

CHAPTER 5

API

This document describes the low-level API of the interfaces and classes provided by this package. The narrative documentation is a better guide to the intended usage.

5.1 Interfaces

interface zope.schema.interfaces.**IField**

Extends: zope.schema._bootstrapinterfaces.IValidatable

Basic Schema Field Interface.

Fields are used for Interface specifications. They at least provide a title, description and a default value. You can also specify if they are required and/or readonly.

The Field Interface is also used for validation and specifying constraints.

We want to make it possible for a IField to not only work on its value but also on the object this value is bound to. This enables a Field implementation to perform validation against an object which also marks a certain place.

Note that many fields need information about the object containing a field. For example, when validating a value to be set as an object attribute, it may be necessary for the field to introspect the object's state. This means that the field needs to have access to the object when performing validation:

```
bound = field.bind(object)
bound.validate(value)
```

bind(*object*)

Return a copy of this field which is bound to context.

The copy of the Field will have the 'context' attribute set to 'object'. This way a Field can implement more complex checks involving the object's location/environment.

Many fields don't need to be bound. Only fields that condition validation or properties on an object containing the field need to be bound.

title
Title

A short summary or label

Implementation *zope.schema.TextLine*

Read Only False

Required False

Default Value u''

Allowed Type `unicode`

description
Description

A description of the field

Implementation *zope.schema.Text*

Read Only False

Required False

Default Value u''

Allowed Type `unicode`

required
Required

Tells whether a field requires its value to exist.

Implementation *zope.schema.Bool*

Read Only False

Required True

Default Value True

Allowed Type `bool`

readonly
Read Only

If true, the field's value cannot be changed.

Implementation *zope.schema.Bool*

Read Only False

Required False

Default Value False

Allowed Type `bool`

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Field*

Read Only False

Required True

Default Value None

missing_value
Missing Value

If input for this Field is missing, and that's ok, then this is the value to use

Implementation *zope.schema.Field*

Read Only False

Required True

Default Value None

order
Field Order

The order attribute can be used to determine the order in which fields in a schema were defined. If one field is created after another (in the same thread), its order will be greater.

(Fields in separate threads could have the same order.)

Implementation *zope.schema.Int*

Read Only True

Required True

Default Value None

Allowed Type `int`, `long`

constraint(*value*)
Check a customized constraint on the value.

You can implement this method with your Field to require a certain constraint. This relaxes the need to inherit/subclass a Field you to add a simple constraint. Returns true if the given value is within the Field's constraint.

validate(*value*)
Validate that the given value is a valid field value.

Returns nothing but raises an error if the value is invalid. It checks everything specific to a Field and also checks with the additional constraint.

get(*object*)
Get the value of the field for the given object.

query(*object*, *default=None*)
Query the value of the field for the given object.

Return the default if the value hasn't been set.

set(*object*, *value*)
Set the value of the field for the object

Raises a type error if the field is a read-only field.

interface `zope.schema.interfaces.IChoice`
Extends: *zope.schema.interfaces.IField*

Field whose value is contained in a predefined set

Only one, values or vocabulary, may be specified for a given choice.

vocabulary
Vocabulary or source providing values

The ISource, IContextSourceBinder or IBaseVocabulary object that provides values for this field.

Implementation *zope.schema.Field*

Read Only False

Required False

Default Value None

vocabularyName
Vocabulary name

Vocabulary name to lookup in the vocabulary registry

Implementation *zope.schema.TextLine*

Read Only False

Required False

Default Value None

Allowed Type `unicode`

interface `zope.schema.interfaces.`**IContextAwareDefaultFactory**
A default factory that requires a context.

The context is the field context. If the field is not bound, context may be `None`.

__call__(*context*)
Returns a default value for the field.

interface `zope.schema.interfaces.`**IOrderable**
Extends: *zope.schema.interfaces.IField*

Field requiring its value to be orderable.

The set of value needs support a complete ordering; the implementation mechanism is not constrained. Either `__cmp__()` or 'rich comparison' methods may be used.

interface `zope.schema.interfaces.`**ILen**
Extends: *zope.schema.interfaces.IField*

A Field requiring its value to have a length.

The value needs to have a conventional __len__ method.

interface `zope.schema.interfaces.`**IMinMax**
Extends: *zope.schema.interfaces.IOrderable*

Field requiring its value to be between min and max.

This implies that the value needs to support the IOrderable interface.

min
Start of the range

Implementation *zope.schema.Field*

Read Only False

Required False

Default Value None

max
End of the range (including the value itself)

Implementation *zope.schema.Field*

Read Only False

Required False

Default Value None

interface zope.schema.interfaces.**IMinMaxLen**
Extends: *zope.schema.interfaces.ILen*

Field requiring the length of its value to be within a range

min_length
Minimum length

Value after whitespace processing cannot have less than *min_length* characters (if a string type) or elements (if another sequence type). If *min_length* is None, there is no minimum.

Implementation *zope.schema.Int*

Read Only False

Required False

Default Value 0

Allowed Type int, long

max_length
Maximum length

Value after whitespace processing cannot have greater or equal than *max_length* characters (if a string type) or elements (if another sequence type). If *max_length* is None, there is no maximum.

Implementation *zope.schema.Int*

Read Only False

Required False

Default Value None

Allowed Type int, long

interface zope.schema.interfaces.**IInterfaceField**
Extends: *zope.schema.interfaces.IField*

Fields with a value that is an interface (implementing zope.interface.Interface).

interface zope.schema.interfaces.**IBool**
Extends: *zope.schema.interfaces.IField*

Boolean Field.

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Bool*

Read Only False

Required True

Default Value None

Allowed Type `bool`

interface `zope.schema.interfaces.`**IObject**
Extends: *zope.schema.interfaces.IField*

Field containing an Object value.

Changed in version 4.6.0: Add the *validate_invariants* attribute.

schema
The Interface that defines the Fields comprising the Object.

Implementation *zope.schema.Object*

Read Only False

Required True

Default Value None

Must Provide `zope.interface.interfaces.IInterface`

validate_invariants
Validate Invariants

A boolean that says whether `schema.validateInvariants` is called from `self.validate()`. The default is true.

Implementation *zope.schema.Bool*

Read Only False

Required True

Default Value True

Allowed Type `bool`

5.1.1 Conversions

interface `zope.schema.interfaces.`**IFromBytes**
Parse a byte string to a value.

If the string needs to be decoded, decoding is done using UTF-8.

New in version 4.8.0.

fromBytes(*value*)
Convert a byte string to a value.

interface `zope.schema.interfaces.`**IFromUnicode**
Parse a unicode string to a value

We will often adapt fields to this interface to support views and other applications that need to convert raw data as unicode values.

fromUnicode(*value*)
Convert a unicode string to a value.

5.1.2 Strings

interface zope.schema.interfaces.**IBytes**
Extends: *zope.schema.interfaces.IMinMaxLen*, *zope.schema.interfaces.IIterable*, *zope.schema.interfaces.IField*

Field containing a byte string (like the python str).

The value might be constrained to be with length limits.

interface zope.schema.interfaces.**IBytesLine**
Extends: *zope.schema.interfaces.IBytes*

Field containing a byte string without newlines.

interface zope.schema.interfaces.**IText**
Extends: *zope.schema.interfaces.IMinMaxLen*, *zope.schema.interfaces.IIterable*, *zope.schema.interfaces.IField*

Field containing a unicode string.

interface zope.schema.interfaces.**ITextLine**
Extends: *zope.schema.interfaces.IText*

Field containing a unicode string without newlines.

interface zope.schema.interfaces.**IASCII**
Extends: *zope.schema.interfaces.INativeString*

Field containing a 7-bit ASCII string. No characters > DEL (chr(127)) are allowed

The value might be constrained to be with length limits.

interface zope.schema.interfaces.**IASCIILine**
Extends: *zope.schema.interfaces.IASCII*

Field containing a 7-bit ASCII string without newlines.

interface zope.schema.interfaces.**INativeString**
Extends: *zope.schema.interfaces.IBytes*

A field that always contains the native `str` type.

Changed in version 4.9.0: This is now a distinct type instead of an alias for either *IText* or *IBytes*, depending on the platform.

interface zope.schema.interfaces.**INativeStringLine**
Extends: *zope.schema.interfaces.IBytesLine*

A field that always contains the native `str` type, without any newlines.

Changed in version 4.9.0: This is now a distinct type instead of an alias for either *ITextLine* or *IBytesLine*, depending on the platform.

interface zope.schema.interfaces.**IPassword**
Extends: *zope.schema.interfaces.ITextLine*

Field containing a unicode string without newlines that is a password.

interface zope.schema.interfaces.**IURI**
Extends: *zope.schema.interfaces.INativeStringLine*

A field containing an absolute URI

interface zope.schema.interfaces.**IId**
Extends: *zope.schema.interfaces.INativeStringLine*

A field containing a unique identifier

A unique identifier is either an absolute URI or a dotted name. If it's a dotted name, it should have a module/package name as a prefix.

interface zope.schema.interfaces.**IPythonIdentifier**
Extends: *zope.schema.interfaces.INativeStringLine*

A single Python identifier, such as a variable name.

New in version 4.9.0.

interface zope.schema.interfaces.**IDottedName**
Extends: *zope.schema.interfaces.INativeStringLine*

Dotted name field.

Values of DottedName fields must be Python-style dotted names.

min_dots
Minimum number of dots

Implementation *zope.schema.Int*

Read Only False

Required True

Default Value 0

Allowed Type int, long

max_dots
Maximum number of dots (should not be less than min_dots)

Implementation *zope.schema.Int*

Read Only False

Required False

Default Value None

Allowed Type int, long

5.1.3 Numbers

interface zope.schema.interfaces.**INumber**
Extends: *zope.schema.interfaces.IMinMax*, *zope.schema.interfaces.IField*

Field containing a generic number: numbers.Number.

See also:

zope.schema.Number

New in version 4.6.0.

min
Start of the range

Implementation *zope.schema.Number*

Read Only False

Required False

Default Value None

Allowed Type numbers.Number

max
End of the range (including the value itself)

Implementation *zope.schema.Number*

Read Only False

Required False

Default Value None

Allowed Type numbers.Number

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Number*

Read Only False

Required True

Default Value None

Allowed Type numbers.Number

interface zope.schema.interfaces.**IComplex**
Extends: *zope.schema.interfaces.INumber*

Field containing a complex number: numbers.Complex.

See also:

zope.schema.Real

New in version 4.6.0.

min
Start of the range

Implementation *zope.schema.Complex*

Read Only False

Required False

Default Value None

Allowed Type numbers.Complex

max
End of the range (including the value itself)

Implementation *zope.schema.Complex*

Read Only False

Required False

Default Value None

Allowed Type numbers.Complex

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Complex*

Read Only False

Required True

Default Value None

Allowed Type numbers.Complex

interface zope.schema.interfaces.**IReal**
Extends: *zope.schema.interfaces.IComplex*

Field containing a real number: numbers.IReal.

See also:

zope.schema.Real

New in version 4.6.0.

min
Start of the range

Implementation *zope.schema.Real*

Read Only False

Required False

Default Value None

Allowed Type numbers.Real

max
End of the range (including the value itself)

Implementation *zope.schema.Real*

Read Only False

Required False

Default Value None

Allowed Type numbers.Real

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Real*

Read Only False

Required True

Default Value None

Allowed Type numbers.Real

interface zope.schema.interfaces.**IRational**
Extends: *zope.schema.interfaces.IReal*

Field containing a rational number: `numbers.IRational`.

See also:

zope.schema.Rational

New in version 4.6.0.

min
Start of the range

Implementation *zope.schema.Rational*

Read Only False

Required False

Default Value None

Allowed Type `numbers.Rational`

max
End of the range (including the value itself)

Implementation *zope.schema.Rational*

Read Only False

Required False

Default Value None

Allowed Type `numbers.Rational`

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Rational*

Read Only False

Required True

Default Value None

Allowed Type `numbers.Rational`

interface zope.schema.interfaces.**IIntegral**
Extends: *zope.schema.interfaces.IRational*

Field containing an integral number: class:`numbers.Integral`.

See also:

zope.schema.Integral

New in version 4.6.0.

min
Start of the range

Implementation *zope.schema.Integral*

Read Only False

Required False

Default Value None

Allowed Type numbers.Integral

max
End of the range (including the value itself)

Implementation *zope.schema.Integral*

Read Only False

Required False

Default Value None

Allowed Type numbers.Integral

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Integral*

Read Only False

Required True

Default Value None

Allowed Type numbers.Integral

interface zope.schema.interfaces.**IInt**
Extends: *zope.schema.interfaces.IIntegral*

Field containing exactly the native class int (or, on Python 2, long).

See also:

zope.schema.Int

min
Start of the range

Implementation *zope.schema.Int*

Read Only False

Required False

Default Value None

Allowed Type int, long

max
End of the range (including the value itself)

Implementation *zope.schema.Int*

Read Only False

Required False

Default Value None

Allowed Type int, long

default
Default Value

The field default value may be None or a legal field value

Implementation *zope.schema.Int*

Read Only False

Required True

Default Value None

Allowed Type `int`, `long`

interface zope.schema.interfaces.**IFloat**
Extends: *zope.schema.interfaces.IReal*

Field containing exactly the native class `float`.

IReal is a more general interface, allowing all of floats, ints, and fractions.

See also:

zope.schema.Float

interface zope.schema.interfaces.**IDecimal**
Extends: *zope.schema.interfaces.INumber*

Field containing a `decimal.Decimal`

5.1.4 Date/Time

interface zope.schema.interfaces.**IDatetime**
Extends: *zope.schema.interfaces.IMinMax*, *zope.schema.interfaces.IField*

Field containing a datetime.

interface zope.schema.interfaces.**IDate**
Extends: *zope.schema.interfaces.IMinMax*, *zope.schema.interfaces.IField*

Field containing a date.

interface zope.schema.interfaces.**ITimedelta**
Extends: *zope.schema.interfaces.IMinMax*, *zope.schema.interfaces.IField*

Field containing a timedelta.

interface zope.schema.interfaces.**ITime**
Extends: *zope.schema.interfaces.IMinMax*, *zope.schema.interfaces.IField*

Field containing a time.

5.1.5 Collections

interface zope.schema.interfaces.**IIterable**
Extends: *zope.schema.interfaces.IField*

Fields with a value that can be iterated over.

The value needs to support iteration; the implementation mechanism is not constrained. (Either `__iter__()` or `__getitem__()` may be used.)

interface zope.schema.interfaces.**IContainer**
Extends: *zope.schema.interfaces.IField*

Fields whose value allows an `x in value` check.

The value needs to support the `in` operator, but is not constrained in how it does so (whether it defines `__contains__()` or `__getitem__()` is immaterial).

interface zope.schema.interfaces.**ICollection**
Extends: *zope.schema.interfaces.IMinMaxLen*, *zope.schema.interfaces.IIterable*, *zope.schema.interfaces.IContainer*

Abstract interface containing a collection value.

The Value must be iterable and may have a min_length/max_length.

value_type
Value Type

Field value items must conform to the given type, expressed via a Field.

Implementation *zope.schema.Object*

Read Only False

Required True

Default Value None

Must Provide *zope.schema.interfaces.IField*

unique
Unique Members

Specifies whether the members of the collection must be unique.

Implementation *zope.schema.Bool*

Read Only False

Required True

Default Value False

Allowed Type `bool`

interface zope.schema.interfaces.**ISequence**
Extends: *zope.schema.interfaces.ICollection*

Abstract interface specifying that the value is ordered

interface zope.schema.interfaces.**IMutableSequence**
Extends: *zope.schema.interfaces.ISequence*

Abstract interface specifying that the value is ordered and mutable.

New in version 4.6.0.

interface zope.schema.interfaces.**IUnorderedCollection**
Extends: *zope.schema.interfaces.ICollection*

Abstract interface specifying that the value cannot be ordered

interface zope.schema.interfaces.**IAbstractSet**
Extends: *zope.schema.interfaces.IUnorderedCollection*

An unordered collection of unique values.

unique
This ICollection interface attribute must be True

Implementation *zope.schema.Bool*

Read Only False

Required True

Default Value None

Allowed Type `bool`

interface zope.schema.interfaces.**IAbstractBag**
Extends: *zope.schema.interfaces.IUnorderedCollection*

An unordered collection of values, with no limitations on whether members are unique

unique
This ICollection interface attribute must be False

Implementation *zope.schema.Bool*

Read Only False

Required True

Default Value None

Allowed Type `bool`

interface zope.schema.interfaces.**ITuple**
Extends: *zope.schema.interfaces.ISequence*

Field containing a value that implements the API of a conventional Python tuple.

interface zope.schema.interfaces.**IList**
Extends: *zope.schema.interfaces.IMutableSequence*

Field containing a value that implements the API of a conventional Python list.

interface zope.schema.interfaces.**ISet**
Extends: *zope.schema.interfaces.IAbstractSet*

Field containing a value that implements the API of a Python2.4+ set.

interface zope.schema.interfaces.**IFrozenSet**
Extends: *zope.schema.interfaces.IAbstractSet*

Field containing a value that implements the API of a conventional Python 2.4+ frozenset.

Mappings

interface zope.schema.interfaces.**IMapping**
Extends: *zope.schema.interfaces.IMinMaxLen*, *zope.schema.interfaces.IIterable*, *zope.schema.interfaces.IContainer*

Field containing an instance of `collections.Mapping`.

The *key_type* and *value_type* fields allow specification of restrictions for keys and values contained in the dict.

key_type
Field keys must conform to the given type, expressed via a Field.

Implementation *zope.schema.Object*

Read Only False

Required True

Default Value None

Must Provide *zope.schema.interfaces.IField*

value_type
Field values must conform to the given type, expressed via a Field.

Implementation *zope.schema.Object*

Read Only False

Required True

Default Value None

Must Provide *zope.schema.interfaces.IField*

interface zope.schema.interfaces.**IMutableMapping**
Extends: *zope.schema.interfaces.IMapping*

Field containing an instance of `collections.MutableMapping`.

interface zope.schema.interfaces.**IDict**
Extends: *zope.schema.interfaces.IMutableMapping*

Field containing a conventional dict.

5.1.6 Events

interface zope.schema.interfaces.**IBeforeObjectAssignedEvent**
An object is going to be assigned to an attribute on another object.

Subscribers to this event can change the object on this event to change what object is going to be assigned. This is useful, e.g. for wrapping or replacing objects before they get assigned to conform to application policy.

object
The object that is going to be assigned.

name
The name of the attribute under which the object will be assigned.

context
The context object where the object will be assigned to.

interface zope.schema.interfaces.**IFieldEvent**

field
The field that has been changed

Implementation *zope.schema.Object*

Read Only False

Required True

Default Value None

Must Provide *zope.schema.interfaces.IField*

object
The object containing the field

interface zope.schema.interfaces.**IFieldUpdatedEvent**
Extends: *zope.schema.interfaces.IFieldEvent*

A field has been modified

Subscribers will get the old and the new value together with the field

old_value
The value of the field before modification

new_value
The value of the field after modification

5.1.7 Vocabularies

interface zope.schema.interfaces.**ITerm**
Object representing a single value in a vocabulary.

value
The value used to represent vocabulary term in a field.

interface zope.schema.interfaces.**ITokenizedTerm**
Extends: *zope.schema.interfaces.ITerm*

Object representing a single value in a tokenized vocabulary.

token
Token which can be used to represent the value on a stream.

The value of this attribute must be a non-empty 7-bit native string (i.e., the `str` type on both Python 2 and 3). Control characters, including newline, are not allowed.

interface zope.schema.interfaces.**ITitledTokenizedTerm**
Extends: *zope.schema.interfaces.ITokenizedTerm*

A tokenized term that includes a title.

title
Title

Implementation *zope.schema.TextLine*

Read Only False

Required True

Default Value None

Allowed Type `unicode`

interface zope.schema.interfaces.**ISource**
A set of values from which to choose

Sources represent sets of values. They are used to specify the source for choice fields.

Sources can be large (even infinite), in which case, they need to be queried to find out what their values are.

__contains__(*value*)
Return whether the value is available in this source

interface zope.schema.interfaces.**ISourceQueriables**
A collection of objects for querying sources

getQueriables()
Return an iterable of objects that can be queried

The returned obects should be two-tuples with:

- A unicode id

  The id must uniquely identify the queriable object within the set of queriable objects. Furthermore, in subsequent calls, the same id should be used for a given queriable object.

- A queriable object

  This is an object for which there is a view provided for searching for items.

interface zope.schema.interfaces.**IContextSourceBinder**

__call__(*context*)
Return a context-bound instance that implements ISource.

interface zope.schema.interfaces.**IBaseVocabulary**
Extends: *zope.schema.interfaces.ISource*

Representation of a vocabulary.

At this most basic level, a vocabulary only need to support a test for containment. This can be implemented either by __contains__() or by sequence __getitem__() (the later only being useful for vocabularies which are intrinsically ordered).

getTerm(*value*)
Return the ITerm object for the term 'value'.

If 'value' is not a valid term, this method raises LookupError.

interface zope.schema.interfaces.**IIterableVocabulary**
Vocabulary which supports iteration over allowed values.

The objects iteration provides must conform to the ITerm interface.

__iter__()
Return an iterator which provides the terms from the vocabulary.

__len__()
Return the number of valid terms, or sys.maxint.

interface zope.schema.interfaces.**IIterableSource**
Extends: *zope.schema.interfaces.ISource*

Source which supports iteration over allowed values.

The objects iteration provides must be values from the source.

__iter__()
Return an iterator which provides the values from the source.

__len__()
Return the number of valid values, or sys.maxint.

interface zope.schema.interfaces.**IVocabulary**
Extends: *zope.schema.interfaces.IIterableVocabulary*, *zope.schema.interfaces.IBaseVocabulary*

Vocabulary which is iterable.

interface zope.schema.interfaces.**IVocabularyTokenized**
Extends: *zope.schema.interfaces.IVocabulary*

Vocabulary that provides support for tokenized representation.

Terms returned from getTerm() and provided by iteration must conform to ITokenizedTerm.

getTermByToken(*token*)
Return an ITokenizedTerm for the passed-in token.

If `token` is not represented in the vocabulary, `LookupError` is raised.

interface zope.schema.interfaces.**ITreeVocabulary**
Extends: *zope.schema.interfaces.IVocabularyTokenized*, zope.interface.common.mapping.IEnumerableMapping

A tokenized vocabulary with a tree-like structure.

The tree is implemented as dictionary, with keys being ITokenizedTerm terms and the values being similar dictionaries. Leaf values are empty dictionaries.

interface zope.schema.interfaces.**IVocabularyRegistry**
Registry that provides *IBaseVocabulary* objects for specific fields.

The fields of this package use the vocabulary registry that is returned from *getVocabularyRegistry()*. This is a hook function; by default it returns an instance of *VocabularyRegistry*, but the function *setVocabularyRegistry()* can be used to change this.

In particular, the package zope.vocabularyregistry can be used to install a vocabulary registry that uses the `zope.component` architecture.

get(*context*, *name*)
Return the vocabulary named *name* for the content object *context*.

When the vocabulary cannot be found, `LookupError` is raised.

interface zope.schema.interfaces.**IVocabularyFactory**
An object that can create *IBaseVocabulary*.

Objects that implement this interface can be registered with the default *VocabularyRegistry* provided by this package.

Alternatively, zope.vocabularyregistry can be used to install a *IVocabularyRegistry* that looks for named utilities using `zope.component.getUtility()` which provide this interface.

__call__(*context*)
The *context* provides a location that the vocabulary can make use of.

5.1.8 Exceptions

exception zope.schema._bootstrapinterfaces.**ValidationError**
Bases: `zope.interface.exceptions.Invalid`

Raised if the Validation process fails.

field = None
The field that raised the error, if known.

value = None
The value that failed validation.

exception zope.schema.**ValidationError**
The preferred alias for *zope.schema._bootstrapinterfaces.ValidationError*.

exception zope.schema.interfaces.**StopValidation**
Bases: exceptions.Exception

Raised if the validation is completed early.

Note that this exception should be always caught, since it is just a way for the validator to save time.

exception zope.schema.interfaces.**RequiredMissing**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Required input is missing.

exception zope.schema.interfaces.**WrongType**(*value*, *expected_type*, *name*)
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Object is of wrong type.

Changed in version 4.7.0: Added named arguments to the constructor and the *expected_type* field.

expected_type = None
The type or tuple of types that was expected.

New in version 4.7.0.

exception zope.schema.interfaces.**ConstraintNotSatisfied**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Constraint not satisfied

exception zope.schema.interfaces.**NotAContainer**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Not a container

exception zope.schema.interfaces.**NotAnIterator**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Not an iterator

exception zope.schema.interfaces.**NotAnInterface**(*value*, *name*)
Bases: zope.schema._bootstrapinterfaces.WrongType, zope.schema._bootstrapinterfaces.SchemaNotProvided

Object is not an interface.

This is a *WrongType* exception for backwards compatibility with existing `except` clauses, but it is raised when `IInterface.providedBy` is not true, so it's also a *SchemaNotProvided*. The `expected_type` field is filled in as `IInterface`; this is not actually a `type`, and `isinstance(thing, IInterface)` is always false.

New in version 4.7.0.

expected_type = <InterfaceClass zope.interface.interfaces.IInterface>

Bounds

exception zope.schema.interfaces.**OutOfBounds**(*value*, *bound*)
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

A value was out of the allowed bounds.

This is the common superclass for *OrderableOutOfBounds* and *LenOutOfBounds*, which in turn are the superclasses for *TooBig* and *TooSmall*, and *TooLong* and *TooShort*, respectively.

New in version 4.7.0.

TOO_LARGE = <zope.schema._bootstrapinterfaces.TOO_LARGE object>
A constant for *violation_direction*.

TOO_SMALL = <zope.schema._bootstrapinterfaces.TOO_SMALL object>
A constant for *violation_direction*.

violation_direction = None
Whether the value was too large or not large enough. One of the values defined by the constants *TOO_LARGE* or *TOO_SMALL*

bound = None
The value that was exceeded

exception zope.schema.interfaces.**OrderableOutOfBounds**(*value*, *bound*)
Bases: zope.schema._bootstrapinterfaces.OutOfBounds

A value was too big or too small in comparison to another value.

New in version 4.7.0.

exception zope.schema.interfaces.**LenOutOfBounds**(*value*, *bound*)
Bases: zope.schema._bootstrapinterfaces.OutOfBounds

The length of the value was out of bounds.

New in version 4.7.0.

exception zope.schema.interfaces.**TooSmall**(*value*, *bound*)
Bases: zope.schema._bootstrapinterfaces.OrderableOutOfBounds

Value is too small

exception zope.schema.interfaces.**TooBig**(*value*, *bound*)
Bases: zope.schema._bootstrapinterfaces.OrderableOutOfBounds

Value is too big

exception zope.schema.interfaces.**TooLong**(*value*, *bound*)
Bases: zope.schema._bootstrapinterfaces.LenOutOfBounds

Value is too long

exception zope.schema.interfaces.**TooShort**(*value*, *bound*)
Bases: zope.schema._bootstrapinterfaces.LenOutOfBounds

Value is too short

exception zope.schema.interfaces.**InvalidValue**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Invalid value

exception zope.schema.interfaces.**WrongContainedType**(*errors*, *name*)
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Wrong contained type

Changed in version 4.7.0: Added named arguments to the constructor, and the *errors* property.

errors = ()
A collection of exceptions raised when validating the *value*.

New in version 4.7.0.

exception zope.schema.interfaces.**NotUnique**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

One or more entries of sequence are not unique.

exception zope.schema.interfaces.**SchemaNotFullyImplemented**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Schema not fully implemented

exception zope.schema.interfaces.**SchemaNotProvided**(*schema*, *value*)
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

Schema not provided

Changed in version 4.7.0: Added named arguments to the constructor and the *schema* property.

schema = None
The interface that the *value* was supposed to provide, but does not.

exception zope.schema.interfaces.**InvalidURI**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

The specified URI is not valid.

exception zope.schema.interfaces.**InvalidId**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

The specified id is not valid.

exception zope.schema.interfaces.**InvalidDottedName**
Bases: *zope.schema._bootstrapinterfaces.ValidationError*

The specified dotted name is not valid.

exception zope.schema.interfaces.**Unbound**
Bases: exceptions.Exception

The field is not bound.

5.2 Schema APIs

zope.schema.**getFields**(*schema*)
Return a dictionary containing all the Fields in a schema.

zope.schema.**getFieldsInOrder**(*schema*, *_field_key=<function <lambda>>*)
Return a list of (name, value) tuples in native schema order.

zope.schema.**getFieldNames**(*schema*)
Return a list of all the Field names in a schema.

zope.schema.**getFieldNamesInOrder**(*schema*)
Return a list of all the Field names in a schema in schema order.

zope.schema.**getValidationErrors**(*schema*, *value*)
Validate that *value* conforms to the schema interface *schema*.

This includes checking for any schema validation errors (using *getSchemaValidationErrors*). If that succeeds, then we proceed to check for any declared invariants.

Note that this does not include a check to see if the *value* actually provides the given *schema*.

Returns A sequence of (name, zope.interface.Invalid) tuples, where *name* is None if the error was from an invariant. If the sequence is empty, there were no errors.

zope.schema.**getSchemaValidationErrors**(*schema*, *value*)
Validate that *value* conforms to the schema interface *schema*.

All *zope.schema.interfaces.IField* members of the *schema* are validated after being bound to *value*. (Note that we do not check for arbitrary zope.interface.Attribute members being present.)

Returns A sequence of (name, *ValidationError*) tuples. A non-empty sequence indicates validation failed.

5.3 Field Implementations

class zope.schema.**Field**(*title=u''*, *description=u''*, *__name__=''*, *required=True*, *readonly=False*, *constraint=None*, *default=None*, *defaultFactory=None*, *missing_value=<Not Given>*)
Bases: zope.interface.interface.Attribute

Pass in field values as keyword parameters.

Generally, you want to pass either a title and description, or a doc string. If you pass no doc string, it will be computed from the title and description. If you pass a doc string that follows the Python coding style (title line separated from the body by a blank line), the title and description will be computed from the doc string. Unfortunately, the doc string must be passed as a positional argument.

Here are some examples:

```
>>> from zope.schema._bootstrapfields import Field
>>> f = Field()
>>> f.__doc__, str(f.title), str(f.description)
('', '', '')
```

```
>>> f = Field(title=u'sample')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample', 'sample', '')
```

```
>>> f = Field(title=u'sample', description=u'blah blah\nblah')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample\n\nblah blah\nblah', 'sample', 'blah blah\nblah')
```

getExtraDocLines()
Return a list of ReST formatted lines that will be added to the docstring returned by *getDoc()*.

By default, this will include information about the various properties of this object, such as required and readonly status, required type, and so on.

This implementation uses a field list for this.

Subclasses may override or extend.

New in version 4.6.0.

getDoc()
Returns the documentation for the object.

class zope.schema.**Collection**(*value_type=<Not Given>*, *unique=<Not Given>*, ***kw*)
Bases: zope.schema._bootstrapfields.MinMaxLen, zope.schema._bootstrapfields.Iterable

A generic collection implementing *zope.schema.interfaces.ICollection*.

Subclasses can define the attribute `value_type` to be a field such as an *Object* that will be checked for each member of the collection. This can then be omitted from the constructor call.

They can also define the attribute `_type` to be a concrete class (or tuple of classes) that the collection itself will be checked to be an instance of. This cannot be set in the constructor.

Changed in version 4.6.0: Add the ability for subclasses to specify `value_type` and `unique`, and allow eliding them from the constructor.

bind(*context*)
See zope.schema._bootstrapinterfaces.IField.

zope.schema._field.**AbstractCollection**
An alternate name for *Collection*.

Deprecated since version 4.6.0: Use *Collection* instead.

alias of `zope.schema._field.Collection`

class zope.schema.**Bool**(*title=u''*, *description=u''*, *__name__=''*, *required=True*, *readonly=False*, *constraint=None*, *default=None*, *defaultFactory=None*, *missing_value=<Not Given>*)
A field representing a Bool.

Changed in version 4.8.0: Implement *zope.schema.interfaces.IFromBytes*

Pass in field values as keyword parameters.

Generally, you want to pass either a title and description, or a doc string. If you pass no doc string, it will be computed from the title and description. If you pass a doc string that follows the Python coding style (title line separated from the body by a blank line), the title and description will be computed from the doc string. Unfortunately, the doc string must be passed as a positional argument.

Here are some examples:

```
>>> from zope.schema._bootstrapfields import Field
>>> f = Field()
>>> f.__doc__, str(f.title), str(f.description)
('', '', '')
```

```
>>> f = Field(title=u'sample')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample', 'sample', '')
```

```
>>> f = Field(title=u'sample', description=u'blah blah\nblah')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample\n\nblah blah\nblah', 'sample', 'blah blah\nblah')
```

fromUnicode(*value*)

```
>>> from zope.schema._bootstrapfields import Bool
>>> from zope.schema.interfaces import IFromUnicode
>>> b = Bool()
>>> IFromUnicode.providedBy(b)
True
>>> b.fromUnicode('True')
True
>>> b.fromUnicode('')
False
```

(continues on next page)

(continued from previous page)

```
>>> b.fromUnicode('true')
True
>>> b.fromUnicode('false') or b.fromUnicode('False')
False
>>> b.fromUnicode(u'\u2603')
False
```

fromBytes(*value*)

```
>>> from zope.schema._bootstrapfields import Bool
>>> from zope.schema.interfaces import IFromBytes
>>> b = Bool()
>>> IFromBytes.providedBy(b)
True
>>> b.fromBytes(b'True')
True
>>> b.fromBytes(b'')
False
>>> b.fromBytes(b'true')
True
>>> b.fromBytes(b'false') or b.fromBytes(b'False')
False
>>> b.fromBytes(u'\u2603'.encode('utf-8'))
False
```

class zope.schema.**Choice**(*values=None*, *vocabulary=None*, *source=None*, ***kw*)

Choice fields can have a value found in a constant or dynamic set of values given by the field definition.

Initialize object.

bind(*context*)

See zope.schema._bootstrapinterfaces.IField.

fromUnicode(*value*)

See IFromUnicode.

class zope.schema.**Container**(*title=u''*, *description=u''*, *__name__=''*, *required=True*, *readonly=False*, *constraint=None*, *default=None*, *defaultFactory=None*, *missing_value=<Not Given>*)

Pass in field values as keyword parameters.

Generally, you want to pass either a title and description, or a doc string. If you pass no doc string, it will be computed from the title and description. If you pass a doc string that follows the Python coding style (title line separated from the body by a blank line), the title and description will be computed from the doc string. Unfortunately, the doc string must be passed as a positional argument.

Here are some examples:

```
>>> from zope.schema._bootstrapfields import Field
>>> f = Field()
>>> f.__doc__, str(f.title), str(f.description)
('', '', '')
```

```
>>> f = Field(title=u'sample')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample', 'sample', '')
```

```
>>> f = Field(title=u'sample', description=u'blah blah\nblah')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample\n\nblah blah\nblah', 'sample', 'blah blah\nblah')
```

class zope.schema.**Date**(*min=None*, *max=None*, *default=None*, ***kw*)
Field containing a date.

class zope.schema.**Datetime**(**args*, ***kw*)
Field containing a datetime.

class zope.schema.**Dict**(*key_type=None*, *value_type=None*, ***kw*)
Bases: zope.schema._field.MutableMapping

A field representing a Dict.

class zope.schema.**FrozenSet**(**args*, ***kwargs*)

class zope.schema.**Id**(*min_length=0*, *max_length=None*, ***kw*)
Id field

Values of id fields must be either uris or dotted names.

Changed in version 4.8.0: Implement *zope.schema.interfaces.IFromBytes*

class zope.schema.**InterfaceField**(*title=u''*, *description=u''*, *__name__=''*, *required=True*, *readonly=False*, *constraint=None*, *default=None*, *defaultFactory=None*, *missing_value=<Not Given>*)
Fields with a value that is an interface (implementing zope.interface.Interface).

Pass in field values as keyword parameters.

Generally, you want to pass either a title and description, or a doc string. If you pass no doc string, it will be computed from the title and description. If you pass a doc string that follows the Python coding style (title line separated from the body by a blank line), the title and description will be computed from the doc string. Unfortunately, the doc string must be passed as a positional argument.

Here are some examples:

```
>>> from zope.schema._bootstrapfields import Field
>>> f = Field()
>>> f.__doc__, str(f.title), str(f.description)
('', '', '')
```

```
>>> f = Field(title=u'sample')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample', 'sample', '')
```

```
>>> f = Field(title=u'sample', description=u'blah blah\nblah')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample\n\nblah blah\nblah', 'sample', 'blah blah\nblah')
```

class zope.schema.**Iterable**(*title=u''*, *description=u''*, *__name__=''*, *required=True*, *readonly=False*, *constraint=None*, *default=None*, *defaultFactory=None*, *missing_value=<Not Given>*)
Pass in field values as keyword parameters.

Generally, you want to pass either a title and description, or a doc string. If you pass no doc string, it will be computed from the title and description. If you pass a doc string that follows the Python coding style (title line separated from the body by a blank line), the title and description will be computed from the doc string. Unfortunately, the doc string must be passed as a positional argument.

Here are some examples:

```
>>> from zope.schema._bootstrapfields import Field
>>> f = Field()
>>> f.__doc__, str(f.title), str(f.description)
('', '', '')
```

```
>>> f = Field(title=u'sample')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample', 'sample', '')
```

```
>>> f = Field(title=u'sample', description=u'blah blah\nblah')
>>> str(f.__doc__), str(f.title), str(f.description)
('sample\n\nblah blah\nblah', 'sample', 'blah blah\nblah')
```

class zope.schema.**List**(*value_type=<Not Given>*, *unique=<Not Given>*, ***kw*)
Bases: zope.schema._field.MutableSequence

A field representing a List.

class zope.schema.**Mapping**(*key_type=None*, *value_type=None*, ***kw*)
A field representing a mapping.

New in version 4.6.0.

bind(*object*)
See zope.schema._bootstrapinterfaces.IField.

class zope.schema.**MutableMapping**(*key_type=None*, *value_type=None*, ***kw*)
Bases: zope.schema._field.Mapping

A field representing a mutable mapping.

New in version 4.6.0.

class zope.schema.**MutableSequence**(*value_type=<Not Given>*, *unique=<Not Given>*, ***kw*)
Bases: zope.schema._field.Sequence

A field representing a mutable sequence.

New in version 4.6.0.

class zope.schema.**MinMaxLen**(*min_length=0*, *max_length=None*, ***kw*)
Bases: object

Expresses constraints on the length of a field.

MinMaxLen is a mixin used in combination with Field.

class zope.schema.**Object**(*schema=<Not Given>*, ***, *validate_invariants=True*, ***kwargs*)
Implementation of *zope.schema.interfaces.IObject*.

Create an IObject field. The keyword arguments are as for *Field*.

Changed in version 4.6.0: Add the keyword argument *validate_invariants*. When true (the default), the schema's validateInvariants method will be invoked to check the @invariant properties of the schema.

Changed in version 4.6.0: The *schema* argument can be ommitted in a subclass that specifies a schema attribute.

getExtraDocLines()
Return a list of ReST formatted lines that will be added to the docstring returned by getDoc().

By default, this will include information about the various properties of this object, such as required and readonly status, required type, and so on.

This implementation uses a field list for this.

Subclasses may override or extend.

New in version 4.6.0.

class zope.schema.**Orderable**(*min=None*, *max=None*, *default=None*, ***kw*)
Bases: object

Values of ordered fields can be sorted.

They can be restricted to a range of values.

Orderable is a mixin used in combination with Field.

class zope.schema.**Set**(**args*, ***kwargs*)
Bases: zope.schema._field._AbstractSet

A field representing a set.

class zope.schema.**Sequence**(*value_type=<Not Given>*, *unique=<Not Given>*, ***kw*)
Bases: zope.schema._field.Collection

A field representing an ordered sequence.

New in version 4.6.0.

class zope.schema.**Time**(*min=None*, *max=None*, *default=None*, ***kw*)
Field containing a time.

class zope.schema.**Timedelta**(*min=None*, *max=None*, *default=None*, ***kw*)
Field containing a timedelta.

class zope.schema.**Tuple**(*value_type=<Not Given>*, *unique=<Not Given>*, ***kw*)
Bases: zope.schema._field.Sequence

A field representing a Tuple.

class zope.schema.**URI**(*min_length=0*, *max_length=None*, ***kw*)
URI schema field.

URIs can be validated from both unicode values and bytes values, producing a native text string in both cases:

```
>>> from zope.schema import URI
>>> field = URI()
>>> field.fromUnicode(u'   https://example.com  ')
'https://example.com'
>>> field.fromBytes(b'   https://example.com ')
'https://example.com'
```

Changed in version 4.8.0: Implement *zope.schema.interfaces.IFromBytes*

5.3.1 Strings

class zope.schema.**ASCII**(*min_length=0*, *max_length=None*, ***kw*)
Field containing a 7-bit ASCII string. No characters > DEL (chr(127)) are allowed

The value might be constrained to be with length limits.

class zope.schema.**ASCIILine**(*min_length=0*, *max_length=None*, ***kw*)
Field containing a 7-bit ASCII string without newlines.

class zope.schema.**Bytes**(*min_length=0*, *max_length=None*, ***kw*)
Field containing a byte string (like the python str).

The value might be constrained to be with length limits.

fromUnicode(*value*)
See IFromUnicode.

class zope.schema.**BytesLine**(*min_length=0*, *max_length=None*, ***kw*)
A *Bytes* field with no newlines.

class zope.schema.**SourceText**(**args*, ***kw*)
Field for source text of object.

class zope.schema.**Text**(**args*, ***kw*)
A field containing text used for human discourse.

fromUnicode(*str*)

```
>>> from zope.schema.interfaces import WrongType
>>> from zope.schema.interfaces import ConstraintNotSatisfied
>>> from zope.schema import Text
>>> from zope.schema._compat import text_type
>>> t = Text(constraint=lambda v: 'x' in v)
>>> t.fromUnicode(b"foo x spam")
Traceback (most recent call last):
...
zope.schema._bootstrapinterfaces.WrongType: ('foo x spam', <type 'unicode'>, '
↪')
>>> result = t.fromUnicode(u"foo x spam")
>>> isinstance(result, bytes)
False
>>> str(result)
'foo x spam'
>>> t.fromUnicode(u"foo spam")
Traceback (most recent call last):
...
zope.schema._bootstrapinterfaces.ConstraintNotSatisfied: (u'foo spam', '')
```

class zope.schema.**TextLine**(**args*, ***kw*)
A text field with no newlines.

class zope.schema.**NativeString**(*min_length=0*, *max_length=None*, ***kw*)
A native string is always the type `str`.

In addition to *INativeString*, this implements *IFromUnicode* and *IFromBytes*.

Changed in version 4.9.0: This is now a distinct type instead of an alias for either *Text* or *Bytes*, depending on the platform.

class zope.schema.**NativeStringLine**(*min_length=0*, *max_length=None*, ***kw*)
A native string is always the type `str`; this field excludes newlines.

In addition to *INativeStringLine*, this implements *IFromUnicode* and *IFromBytes*.

Changed in version 4.9.0: This is now a distinct type instead of an alias for either *TextLine* or *BytesLine*, depending on the platform.

class zope.schema.**Password**(**args*, ***kw*)
A text field containing a text used as a password.

set (*context*, *value*)
Update the password.

We use a special marker value that a widget can use to tell us that the password didn't change. This is needed to support edit forms that don't display the existing password and want to work together with encryption.

class zope.schema.**DottedName**(**args*, ***kw*)
Dotted name field.

Values of DottedName fields must be Python-style dotted names.

Dotted names can be validated from both unicode values and bytes values, producing a native text string in both cases:

```
>>> from zope.schema import DottedName
>>> field = DottedName()
>>> field.fromUnicode(u'zope.schema')
'zope.schema'
>>> field.fromBytes(b'zope.schema')
'zope.schema'
>>> field.fromUnicode(u'zope._schema')
'zope._schema'
```

Changed in version 4.8.0: Implement *zope.schema.interfaces.IFromBytes*

Changed in version 4.9.0: Allow leading underscores in each component.

class zope.schema.**PythonIdentifier**(*min_length=0*, *max_length=None*, ***kw*)
This field describes a python identifier, i.e. a variable name.

Empty strings are allowed.

Identifiers can be validated from both unicode values and bytes values, producing a native text string in both cases:

```
>>> from zope.schema import PythonIdentifier
>>> field = PythonIdentifier()
>>> field.fromUnicode(u'zope')
'zope'
>>> field.fromBytes(b'_zope')
'_zope'
>>> field.fromUnicode(u'    ')
''
```

New in version 4.9.0.

5.3.2 Numbers

class zope.schema.**Number**(*min=None*, *max=None*, *default=None*, ***kw*)
Bases: zope.schema._bootstrapfields.Orderable, zope.schema._bootstrapfields.Field

A field representing a numbers.Number and implementing *zope.schema.interfaces.INumber*.

The fromUnicode() method will attempt to use the smallest or strictest possible type to represent incoming strings:

```
>>> from zope.schema._bootstrapfields import Number
>>> f = Number()
>>> f.fromUnicode(u"1")
1
>>> f.fromUnicode(u"125.6")
125.6
>>> f.fromUnicode(u"1+0j")
(1+0j)
>>> f.fromUnicode(u"1/2")
Fraction(1, 2)
>>> f.fromUnicode(str(2**31234) + '.' + str(2**256))
Decimal('234...936')
>>> f.fromUnicode(u"not a number")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Decimal: 'not a number'
```

Similarly, `fromBytes()` will do the same for incoming byte strings:

```
>>> from zope.schema._bootstrapfields import Number
>>> f = Number()
>>> f.fromBytes(b"1")
1
>>> f.fromBytes(b"125.6")
125.6
>>> f.fromBytes(b"1+0j")
(1+0j)
>>> f.fromBytes(b"1/2")
Fraction(1, 2)
>>> f.fromBytes((str(2**31234) + '.' + str(2**256)).encode('ascii'))
Decimal('234...936')
>>> f.fromBytes(b"not a number")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Decimal: 'not a number'
```

New in version 4.6.0.

Changed in version 4.8.0: Implement *zope.schema.interfaces.IFromBytes*

class zope.schema.**Complex**(*min=None*, *max=None*, *default=None*, ***kw*)

Bases: zope.schema._bootstrapfields.Number

A field representing a numbers.Complex and implementing *zope.schema.interfaces.IComplex*.

The `fromUnicode()` method is like that for *Number*, but doesn't allow Decimals:

```
>>> from zope.schema._bootstrapfields import Complex
>>> f = Complex()
>>> f.fromUnicode(u"1")
1
>>> f.fromUnicode(u"125.6")
125.6
>>> f.fromUnicode(u"1+0j")
(1+0j)
>>> f.fromUnicode(u"1/2")
Fraction(1, 2)
>>> f.fromUnicode(str(2**31234) + '.' + str(2**256))
inf
```

(continues on next page)

(continued from previous page)

```
>>> f.fromUnicode(u"not a number")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Decimal: 'not a number'
```

Similarly for `fromBytes()`:

```
>>> from zope.schema._bootstrapfields import Complex
>>> f = Complex()
>>> f.fromBytes(b"1")
1
>>> f.fromBytes(b"125.6")
125.6
>>> f.fromBytes(b"1+0j")
(1+0j)
>>> f.fromBytes(b"1/2")
Fraction(1, 2)
>>> f.fromBytes((str(2**31234) + '.' + str(2**256)).encode('ascii'))
inf
>>> f.fromBytes(b"not a number")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Decimal: 'not a number'
```

New in version 4.6.0.

class zope.schema.**Real**(*min=None*, *max=None*, *default=None*, ***kw*)

Bases: zope.schema._bootstrapfields.Complex

A field representing a `numbers.Real` and implementing *zope.schema.interfaces.IReal*.

The `fromUnicode()` method is like that for *Complex*, but doesn't allow Decimals or complex numbers:

```
>>> from zope.schema._bootstrapfields import Real
>>> f = Real()
>>> f.fromUnicode("1")
1
>>> f.fromUnicode("125.6")
125.6
>>> f.fromUnicode("1+0j")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Fraction: '1+0j'
>>> f.fromUnicode("1/2")
Fraction(1, 2)
>>> f.fromUnicode(str(2**31234) + '.' + str(2**256))
inf
>>> f.fromUnicode("not a number")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Decimal: 'not a number'
```

New in version 4.6.0.

class zope.schema.**Rational**(*min=None*, *max=None*, *default=None*, ***kw*)

Bases: zope.schema._bootstrapfields.Real

A field representing a `numbers.Rational` and implementing *zope.schema.interfaces.IRational*.

The `fromUnicode()` method is like that for *Real*, but does not allow arbitrary floating point numbers:

```
>>> from zope.schema._bootstrapfields import Rational
>>> f = Rational()
>>> f.fromUnicode("1")
1
>>> f.fromUnicode("1/2")
Fraction(1, 2)
>>> f.fromUnicode("125.6")
Fraction(628, 5)
>>> f.fromUnicode("1+0j")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Fraction: '1+0j'
>>> f.fromUnicode(str(2**31234) + '.' + str(2**256))
Fraction(777..., 330...)
>>> f.fromUnicode("not a number")
Traceback (most recent call last):
...
InvalidNumberLiteral: Invalid literal for Decimal: 'not a number'
```

New in version 4.6.0.

class zope.schema.**Integral**(*min=None*, *max=None*, *default=None*, ***kw*)

Bases: `zope.schema._bootstrapfields.Rational`

A field representing a `numbers.Integral` and implementing *zope.schema.interfaces.IIntegral*.

The `fromUnicode()` method only allows integral values:

```
>>> from zope.schema._bootstrapfields import Integral
>>> f = Integral()
>>> f.fromUnicode("125")
125
>>> f.fromUnicode("125.6")
Traceback (most recent call last):
...
InvalidIntLiteral: invalid literal for int(): 125.6
```

Similarly for `fromBytes()`:

```
>>> from zope.schema._bootstrapfields import Integral
>>> f = Integral()
>>> f.fromBytes(b"125")
125
>>> f.fromBytes(b"125.6")
Traceback (most recent call last):
...
InvalidIntLiteral: invalid literal for int(): 125.6
```

New in version 4.6.0.

class zope.schema.**Float**(*min=None*, *max=None*, *default=None*, ***kw*)

Bases: `zope.schema._bootstrapfields.Real`

A field representing a native `float` and implementing *zope.schema.interfaces.IFloat*.

The class *zope.schema.Real* is a more general version, accepting floats, integers, and fractions.

The `fromUnicode()` method only accepts values that can be parsed by the `float` constructor:

```
>>> from zope.schema._field import Float
>>> f = Float()
>>> f.fromUnicode("1")
1.0
>>> f.fromUnicode("125.6")
125.6
>>> f.fromUnicode("1+0j")
Traceback (most recent call last):
...
InvalidFloatLiteral: Invalid literal for float(): 1+0j
>>> f.fromUnicode("1/2")
Traceback (most recent call last):
...
InvalidFloatLiteral: invalid literal for float(): 1/2
>>> f.fromUnicode(str(2**31234) + '.' + str(2**256))
inf
>>> f.fromUnicode("not a number")
Traceback (most recent call last):
...
InvalidFloatLiteral: could not convert string to float: not a number
```

Likewise for `fromBytes()`:

```
>>> from zope.schema._field import Float
>>> f = Float()
>>> f.fromBytes(b"1")
1.0
>>> f.fromBytes(b"125.6")
125.6
>>> f.fromBytes(b"1+0j")
Traceback (most recent call last):
...
InvalidFloatLiteral: Invalid literal for float(): 1+0j
>>> f.fromBytes(b"1/2")
Traceback (most recent call last):
...
InvalidFloatLiteral: invalid literal for float(): 1/2
>>> f.fromBytes((str(2**31234) + '.' + str(2**256)).encode('ascii'))
inf
>>> f.fromBytes(b"not a number")
Traceback (most recent call last):
...
InvalidFloatLiteral: could not convert string to float: not a number
```

class zope.schema.**Int**(*min=None*, *max=None*, *default=None*, ***kw*)

Bases: zope.schema._bootstrapfields.Integral

A field representing a native integer type. and implementing *zope.schema.interfaces.IInt*.

class zope.schema.**Decimal**(*min=None*, *max=None*, *default=None*, ***kw*)

Bases: zope.schema._bootstrapfields.Number

A field representing a native decimal.Decimal and implementing *zope.schema.interfaces.IDecimal*.

The `fromUnicode()` method only accepts values that can be parsed by the `Decimal` constructor:

```
>>> from zope.schema._field import Decimal
>>> f = Decimal()
>>> f.fromUnicode("1")
Decimal('1')
>>> f.fromUnicode("125.6")
Decimal('125.6')
>>> f.fromUnicode("1+0j")
Traceback (most recent call last):
...
InvalidDecimalLiteral: Invalid literal for Decimal(): 1+0j
>>> f.fromUnicode("1/2")
Traceback (most recent call last):
...
InvalidDecimalLiteral: Invalid literal for Decimal(): 1/2
>>> f.fromUnicode(str(2**31234) + '.' + str(2**256))
Decimal('2349...936')
>>> f.fromUnicode("not a number")
Traceback (most recent call last):
...
InvalidDecimalLiteral: could not convert string to float: not a number
```

Likewise for `fromBytes()`:

```
>>> from zope.schema._field import Decimal
>>> f = Decimal()
>>> f.fromBytes(b"1")
Decimal('1')
>>> f.fromBytes(b"125.6")
Decimal('125.6')
>>> f.fromBytes(b"1+0j")
Traceback (most recent call last):
...
InvalidDecimalLiteral: Invalid literal for Decimal(): 1+0j
>>> f.fromBytes(b"1/2")
Traceback (most recent call last):
...
InvalidDecimalLiteral: Invalid literal for Decimal(): 1/2
>>> f.fromBytes((str(2**31234) + '.' + str(2**256)).encode("ascii"))
Decimal('2349...936')
>>> f.fromBytes(b"not a number")
Traceback (most recent call last):
...
InvalidDecimalLiteral: could not convert string to float: not a number
```

5.4 Vocabularies

Vocabulary support for schema.

class zope.schema.vocabulary.**SimpleTerm**(*value*, *token=None*, *title=None*)

Bases: object

Simple tokenized term used by SimpleVocabulary.

Changed in version 4.6.0: Implement equality and hashing based on the value, token and title.

Create a term for *value* and *token*. If *token* is omitted, str(value) is used for the token, escaping any non-ASCII characters.

If *title* is provided, term implements *zope.schema.interfaces.ITitledTokenizedTerm*.

class zope.schema.vocabulary.**SimpleVocabulary**(*terms*, **interfaces*, ***kwargs*)
Bases: object

Vocabulary that works from a sequence of terms.

Changed in version 4.6.0: Implement equality and hashing based on the terms list and interfaces implemented by this object.

Initialize the vocabulary given a list of terms.

The vocabulary keeps a reference to the list of terms passed in; it should never be modified while the vocabulary is used.

One or more interfaces may also be provided so that alternate widgets may be bound without subclassing.

By default, ValueErrors are thrown if duplicate values or tokens are passed in. If you want to swallow these exceptions, pass in `swallow_duplicates=True`. In this case, the values will override themselves.

classmethod fromItems(*items*, **interfaces*)
Construct a vocabulary from a list of (token, value) pairs or (token, value, title) triples. The list does not have to be homogeneous.

The order of the items is preserved as the order of the terms in the vocabulary. Terms are created by calling the class method `createTerm`()` with the pair or triple.

One or more interfaces may also be provided so that alternate widgets may be bound without subclassing.

Changed in version 4.6.0: Allow passing in triples to set item titles.

classmethod fromValues(*values*, **interfaces*)
Construct a vocabulary from a simple list.

Values of the list become both the tokens and values of the terms in the vocabulary. The order of the values is preserved as the order of the terms in the vocabulary. Tokens are created by calling the class method *createTerm()* with the value as the only parameter.

One or more interfaces may also be provided so that alternate widgets may be bound without subclassing.

classmethod createTerm(**args*)
Create a single term from data.

Subclasses may override this with a class method that creates a term of the appropriate type from the arguments.

getTerm(*value*)
See zope.schema.interfaces.IBaseVocabulary

getTermByToken(*token*)
See zope.schema.interfaces.IVocabularyTokenized

class zope.schema.vocabulary.**TreeVocabulary**(*terms*, **interfaces*)
Bases: object

Vocabulary that relies on a tree (i.e nested) structure.

Initialize the vocabulary given a recursive dict (i.e a tree) with ITokenizedTerm objects for keys and self-similar dicts representing the branches for values.

Refer to the method fromDict for more details.

Concerning the ITokenizedTerm keys, the 'value' and 'token' attributes of each key (including nested ones) must be unique.

One or more interfaces may also be provided so that alternate widgets may be bound without subclassing.

terms_factory
alias of collections.OrderedDict

get(*key*, *default=None*)
Get a value for a key

The default is returned if there is no value for the key.

keys()
Return the keys of the mapping object.

values()
Return the values of the mapping object.

items()
Return the items of the mapping object.

classmethod fromDict(*dict_*, **interfaces*)
Constructs a vocabulary from a dictionary-like object (like dict or OrderedDict), that has tuples for keys.

The tuples should have either 2 or 3 values, i.e: (token, value, title) or (token, value). Only tuples that have three values will create a *zope.schema.interfaces.ITitledTokenizedTerm*.

For example, a dict with 2-valued tuples:

```
dict_ = {
  ('exampleregions', 'Regions used in ATVocabExample'): {
      ('aut', 'Austria'): {
          ('tyr', 'Tyrol'): {
              ('auss', 'Ausserfern'): {},
          }
      },
      ('ger', 'Germany'): {
          ('bav', 'Bavaria'):{}
      },
  }
}
```

One or more interfaces may also be provided so that alternate widgets may be bound without subclassing.

Changed in version 4.6.0: Only create ITitledTokenizedTerm when a title is actually provided.

getTerm(*value*)
See zope.schema.interfaces.IBaseVocabulary

getTermByToken(*token*)
See zope.schema.interfaces.IVocabularyTokenized

getTermPath(*value*)
Returns a list of strings representing the path from the root node to the node with the given value in the tree.

Returns an empty string if no node has that value.

exception zope.schema.vocabulary.**VocabularyRegistryError**(*name*)
Bases: exceptions.LookupError

A specialized subclass of LookupError raised for unknown (unregistered) vocabularies.

See also:

VocabularyRegistry

class zope.schema.vocabulary.**VocabularyRegistry**
Bases: object

Default implementation of *zope.schema.interfaces.IVocabularyRegistry*.

An instance of this class is used by default by *getVocabularyRegistry()*, which in turn is used by *Choice* fields.

Named vocabularies must be manually registered with this object using *register()*. This associates a vocabulary name with a *zope.schema.interfaces.IVocabularyFactory*.

An alternative to this is to use the zope.component registry via zope.vocabularyregistry.

get(*context*, *name*)
See zope.schema.interfaces.IVocabularyRegistry

register(*name*, *factory*)
Register a *factory* for the vocabulary with the given *name*.

zope.schema.vocabulary.**getVocabularyRegistry**()
Return the vocabulary registry.

If the registry has not been created yet, an instance of VocabularyRegistry will be installed and used.

zope.schema.vocabulary.**setVocabularyRegistry**(*registry*)
Set the vocabulary registry.

5.5 Accessors

5.5.1 Field accessors

Accessors are used to model methods used to access data defined by fields. Accessors are fields that work by decorating existing fields.

To define accessors in an interface, use the accessors function:

```
class IMyInterface(Interface):

   getFoo, setFoo = accessors(Text(title=u'Foo', ...))

   getBar = accessors(TextLine(title=u'Foo', readonly=True, ...)
```

Normally a read accessor and a write accessor are defined. Only a read accessor is defined for read-only fields.

Read accessors function as access method specifications and as field specifications. Write accessors are solely method specifications.

class zope.schema.accessors.**FieldReadAccessor**(*field*)
Bases: zope.interface.interface.Method

Field read accessor

class zope.schema.accessors.**FieldWriteAccessor**(*field*)
Bases: zope.interface.interface.Method

zope.schema.accessors.**accessors**(*field*)

CHAPTER 6

Changes

6.1 4.9.4 (unreleased)

- Nothing changed yet.

6.2 4.9.3 (2018-10-12)

- Fixed a ReST error in getDoc() results when having "subfields" with titles.

6.3 4.9.2 (2018-10-11)

- Make sure that the title for `IObject.validate_invariants` is a unicode string.

6.4 4.9.1 (2018-10-05)

- Fix `SimpleTerm` token for non-ASCII bytes values.

6.5 4.9.0 (2018-09-24)

- Make `NativeString` and `NativeStringLine` distinct types that implement the newly-distinct interfaces `INativeString` and `INativeStringLine`. Previously these were just aliases for either `Text` (on Python 3) or `Bytes` (on Python 2).
- Fix `Field.getDoc()` when `value_type` or `key_type` is present. Previously it could produce ReST that generated Sphinx warnings. See issue 76.
- Make `DottedName` accept leading underscores for each segment.

- Add `PythonIdentifier`, which accepts one segment of a dotted name, e.g., a python variable or class.

6.6 4.8.0 (2018-09-19)

- Add the interface `IFromBytes`, which is implemented by the numeric and bytes fields, as well as `URI`, `DottedName`, and `Id`.
- Fix passing `None` as the description to a field constructor. See issue 69.

6.7 4.7.0 (2018-09-11)

- Make `WrongType` have an `expected_type` field.
- Add `NotAnInterface`, an exception derived from `WrongType` and `SchemaNotProvided` and raised by the constructor of `Object` and when validation fails for `InterfaceField`.
- Give `SchemaNotProvided` a `schema` field.
- Give `WrongContainedType` an `errors` list.
- Give `TooShort`, `TooLong`, `TooBig` and `TooSmall` a `bound` field and the common superclasses `LenOutOfBounds`, `OrderableOutOfBounds`, respectively, both of which inherit from `OutOfBounds`.

6.8 4.6.2 (2018-09-10)

- Fix checking a field's constraint to set the `field` and `value` properties if the constraint raises a `ValidationError`. See issue 66.

6.9 4.6.1 (2018-09-10)

- Fix the `Field` constructor to again allow `MessageID` values for the `description`. This was a regression introduced with the fix for issue 60. See issue 63.

6.10 4.6.0 (2018-09-07)

- Add support for Python 3.7.
- `Object` instances call their schema's `validateInvariants` method by default to collect errors from functions decorated with `@invariant` when validating. This can be disabled by passing `validate_invariants=False` to the `Object` constructor. See issue 10.
- `ValidationError` can be sorted on Python 3.
- `DottedName` and `Id` consistently handle non-ASCII unicode values on Python 2 and 3 by raising `InvalidDottedName` and `InvalidId` in `fromUnicode` respectively. Previously, a `UnicodeEncodeError` would be raised on Python 2 while Python 3 would raise the descriptive exception.

- `Field` instances are hashable on Python 3, and use a defined hashing algorithm that matches what equality does on all versions of Python. Previously, on Python 2, fields were hashed based on their identity. This violated the rule that equal objects should have equal hashes, and now they do. Since having equal hashes does not imply that the objects are equal, this is not expected to be a compatibility problem. See issue 36.
- `Field` instances are only equal when their `.interface` is equal. In practice, this means that two otherwise identical fields of separate schemas are not equal, do not hash the same, and can both be members of the same `dict` or `set`. Prior to this release, when hashing was identity based but only worked on Python 2, that was the typical behaviour. (Field objects that are *not* members of a schema continue to compare and hash equal if they have the same attributes and interfaces.) See issue 40.
- Orderable fields, including `Int`, `Float`, `Decimal`, `Timedelta`, `Date` and `Time`, can now have a `missing_value` without needing to specify concrete `min` and `max` values (they must still specify a `default` value). See issue 9.
- `Choice`, `SimpleVocabulary` and `SimpleTerm` all gracefully handle using Unicode token values with non-ASCII characters by encoding them with the `backslashreplace` error handler. See issue 15 and PR 6.
- All instances of `ValidationError` have a `field` and `value` attribute that is set to the field that raised the exception and the value that failed validation.
- `Float`, `Int` and `Decimal` fields raise `ValidationError` subclasses for literals that cannot be parsed. These subclasses also subclass `ValueError` for backwards compatibility.
- Add a new exception `SchemaNotCorrectlyImplemented`, a subclass of `WrongContainedType` that is raised by the `Object` field. It has a dictionary (`schema_errors`) mapping invalid schema attributes to their corresponding exception, and a list (`invariant_errors`) containing the exceptions raised by validating invariants. See issue 16.
- Add new fields `Mapping` and `MutableMapping`, corresponding to the collections ABCs of the same name; `Dict` now extends and specializes `MutableMapping` to only accept instances of `dict`.
- Add new fields `Sequence` and `MutableSequence`, corresponding to the collections ABCs of the same name; `Tuple` now extends `Sequence` and `List` now extends `MutableSequence`.
- Add new field `Collection`, implementing `ICollection`. This is the base class of `Sequence`. Previously this was known as `AbstractCollection` and was not public. It can be subclassed to add `value_type`, `_type` and `unique` attributes at the class level, enabling a simpler constructor call. See issue 23.
- Make `Object` respect a `schema` attribute defined by a subclass, enabling a simpler constructor call. See issue 23.
- Add fields and interfaces representing Python's numeric tower. In descending order of generality these are `Number`, `Complex`, `Real`, `Rational` and `Integral`. The `Int` class extends `Integral`, the `Float` class extends `Real`, and the `Decimal` class extends `Number`. See issue 49.
- Make `Iterable` and `Container` properly implement `IIterable` and `IContainer`, respectively.
- Make `SimpleVocabulary.fromItems` accept triples to allow specifying the title of terms. See issue 18.
- Make `TreeVocabulary.fromDict` only create `ITitledTokenizedTerms` when a title is actually provided.
- Make `Choice` fields reliably raise a `ValidationError` when a named vocabulary cannot be found; for backwards compatibility this is also a `ValueError`. Previously this only worked when the default `VocabularyRegistry` was in use, not when it was replaced with zope.vocabularyregistry. See issue 55.
- Make `SimpleVocabulary` and `SimpleTerm` have value-based equality and hashing methods.
- All fields of the schema of an `Object` field are bound to the top-level value being validated before attempting validation of their particular attribute. Previously only `IChoice` fields were bound. See issue 17.

- Share the internal logic of `Object` field validation and `zope.schema.getValidationErrors`. See issue 57.
- Make `Field.getDoc()` return more information about the properties of the field, such as its required and readonly status. Subclasses can add more information using the new method `Field.getExtraDocLines()`. This is used to generate Sphinx documentation when using repoze.sphinx.autointerface. See issue 60.

6.11 4.5.0 (2017-07-10)

- Drop support for Python 2.6, 3.2, and 3.3.
- Add support for Python 3.5 and 3.6.
- Drop support for 'setup.py test'. Use zope.testrunner instead.

6.12 4.4.2 (2014-09-04)

- Fix description of min max field: max value is included, not excluded.

6.13 4.4.1 (2014-03-19)

- Add support for Python 3.4.

6.14 4.4.0 (2014-01-22)

- Add an event on field properties to notify that a field has been updated. This event enables definition of subscribers based on an event, a context and a field. The event contains also the old value and the new value. (also see package `zope.schemaevent` that define a field event handler)

6.15 4.3.3 (2014-01-06)

- PEP 8 cleanup.
- Don't raise RequiredMissing if a field's defaultFactory returns the field's missing_value.
- Update `boostrap.py` to version 2.2.
- Add the ability to swallow ValueErrors when rendering a SimpleVocabulary, allowing for cases where vocabulary items may be duplicated (e.g., due to user input).
- Include the field name in `ConstraintNotSatisfied`.

6.16 4.3.2 (2013-02-24)

- Fix Python 2.6 support. (Forgot to run tox with all environments before last release.)

6.17 4.3.1 (2013-02-24)

- Make sure that we do not fail during bytes decoding of term token when generated from a bytes value by ignoring all errors. (Another option would have been to hexlify the value, but that would break way too many tests.)

6.18 4.3.0 (2013-02-24)

- Fix a bug where bytes values were turned into tokens inproperly in Python 3.
- Add `zope.schema.fieldproperty.createFieldProperties()` function which maps schema fields into `FieldProperty` instances.

6.19 4.2.2 (2012-11-21)

- Add support for Python 3.3.

6.20 4.2.1 (2012-11-09)

- Fix the default property of fields that have no defaultFactory attribute.

6.21 4.2.0 (2012-05-12)

- Automate build of Sphinx HTML docs and running doctest snippets via tox.
- Drop explicit support for Python 3.1.
- Introduce NativeString and NativeStringLine which are equal to Bytes and BytesLine on Python 2 and Text and TextLine on Python 3.
- Change IURI from a Bytes string to a "native" string. This is a backwards incompatibility which only affects Python 3.
- Bring unit test coverage to 100%.
- Move doctests from the package and wired up as normal Sphinx documentation.
- Add explicit support for PyPy.
- Add support for continuous integration using `tox` and `jenkins`.
- Drop the external `six` dependency in favor of a much-trimmed `zope.schema._compat` module.
- Ensure tests pass when run under `nose`.
- Add `setup.py dev` alias (runs `setup.py develop` plus installs `nose` and `coverage`).
- Add `setup.py docs` alias (installs `Sphinx` and dependencies).

6.22 4.1.1 (2012-03-23)

- Remove trailing slash in MANIFEST.in, it causes Winbot to crash.

6.23 4.1.0 (2012-03-23)

- Add TreeVocabulary for nested tree-like vocabularies.
- Fix broken Object field validation where the schema contains a Choice with ICountextSourceBinder source. In this case the vocabulary was not iterable because the field was not bound and the source binder didn't return the real vocabulary. Added simple test for IContextSourceBinder validation. But a test with an Object field with a schema using a Choice with IContextSourceBinder is still missing.

6.24 4.0.1 (2011-11-14)

- Fix bug in `fromUnicode` method of `DottedName` which would fail validation on being given unicode. Introduced in 4.0.0.

6.25 4.0.0 (2011-11-09)

- Fix deprecated unittest methods.
- Port to Python 3. This adds a dependency on six and removes support for Python 2.5.

6.26 3.8.1 (2011-09-23)

- Fix broken Object field validation. Previous version was using a volatile property on object field values which ends in a ForbiddenAttribute error on security proxied objects.

6.27 3.8.0 (2011-03-18)

- Implement a `defaultFactory` attribute for all fields. It is a callable that can be used to compute default values. The simplest case is:

```
Date(defaultFactory=datetime.date.today)
```

  If the factory needs a context to compute a sensible default value, then it must provide `IContextAwareDefaultFactory`, which can be used as follows:

```
@provider(IContextAwareDefaultFactory)
def today(context):
    return context.today()

Date(defaultFactory=today)
```

6.28 3.7.1 (2010-12-25)

- Rename the validation token, used in the validation of schema with Object Field to avoid infinite recursion: `__schema_being_validated` became `_v_schema_being_validated`, a volatile attribute, to avoid persistency and therefore, read/write conflicts.

- Don't allow "[]^`" in DottedName. https://bugs.launchpad.net/zope.schema/+bug/191236

6.29 3.7.0 (2010-09-12)

- Improve error messages when term tokens or values are duplicates.
- Fix the buildout so the tests run.

6.30 3.6.4 (2010-06-08)

- fix validation of schema with Object Field that specify Interface schema.

6.31 3.6.3 (2010-04-30)

- Prefer the standard libraries doctest module to the one from zope.testing.

6.32 3.6.2 (2010-04-30)

- Avoid maximum recursion when validating Object field that points to cycles
- Make the dependency on `zope.i18nmessageid` optional.

6.33 3.6.1 (2010-01-05)

- Allow "setup.py test" to run at least a subset of the tests runnable via `bin/test` (227 for `setup.py test` vs. 258. for `bin/test`)
- Make `zope.schema._bootstrapfields.ValidatedProperty` descriptor work under Jython.
- Make "setup.py test" tests pass on Jython.

6.34 3.6.0 (2009-12-22)

- Prefer zope.testing.doctest over doctestunit.
- Extend validation error to hold the field name.
- Add FieldProperty class that uses Field.get and Field.set methods instead of storing directly on the instance __dict__.

6.35 3.5.4 (2009-03-25)

- Don't fail trying to validate default value for Choice fields with IContextSourceBinder object given as a source. See https://bugs.launchpad.net/zope3/+bug/340416.
- Add an interface for `DottedName` field.

- Add `vocabularyName` attribute to the `IChoice` interface, change "vocabulary" attribute description to be more sensible, making it `zope.schema.Field` instead of plain `zope.interface.Attribute`.
- Make IBool interface of Bool more important than IFromUnicode so adapters registered for IBool take precendence over adapters registered for IFromUnicode.

6.36 3.5.3 (2009-03-10)

- Make Choice and Bool fields implement IFromUnicode interface, because they do provide the `fromUnicode` method.
- Change package's mailing list address to zope-dev at zope.org, as zope3-dev at zope.org is now retired.
- Fix package's documentation formatting. Change package's description.
- Add buildout part that builds Sphinx-generated documentation.
- Remove zpkg-related file.

6.37 3.5.2 (2009-02-04)

- Made validation tests compatible with Python 2.5 again (hopefully not breaking Python 2.4)
- Add an __all__ package attribute to expose documentation.

6.38 3.5.1 (2009-01-31)

- Stop using the old old set type.
- Make tests compatible and silent with Python 2.4.
- Fix __cmp__ method in ValidationError. Show some side effects based on the existing __cmp__ implementation. See validation.txt
- Make 'repr' of the ValidationError and its subclasses more sensible. This may require you to adapt your doctests for the new style, but now it makes much more sense for debugging for developers.

6.39 3.5.0a2 (2008-12-11)

- Move zope.testing to "test" extras_require, as it is not needed for zope.schema itself.
- Change the order of classes in SET_TYPES tuple, introduced in previous release to one that was in 3.4 (SetType, set), because third-party code could be dependent on that order. The one example is z3c.form's converter.

6.40 3.5.0a1 (2008-10-10)

- Add the doctests to the long description.
- Remove use of deprecated 'sets' module when running under Python 2.6.
- Remove spurious doctest failure when running under Python 2.6.

- Add support to bootstrap on Jython.
- Add helper methods for schema validation: `getValidationErrors` and `getSchemaValidationErrors`.
- zope.schema now works on Python2.5

6.41 3.4.0 (2007-09-28)

Add BeforeObjectAssignedEvent that is triggered before the object field sets a value.

6.42 3.3.0 (2007-03-15)

Corresponds to the version of the zope.schema package shipped as part of the Zope 3.3.0 release.

6.43 3.2.1 (2006-03-26)

Corresponds to the version of the zope.schema package shipped as part of the Zope 3.2.1 release.

Fix missing import of 'VocabularyRegistryError'. See http://www.zope.org/Collectors/Zope3-dev/544 .

6.44 3.2.0 (2006-01-05)

Corresponds to the version of the zope.schema package shipped as part of the Zope 3.2.0 release.

Add "iterable" sources to replace vocabularies, which are now deprecated and scheduled for removal in Zope 3.3.

6.45 3.1.0 (2005-10-03)

Corresponds to the version of the zope.schema package shipped as part of the Zope 3.1.0 release.

Allow 'Choice' fields to take either a 'vocabulary' or a 'source' argument (sources are a simpler implementation).

Add 'TimeDelta' and 'ASCIILine' field types.

6.46 3.0.0 (2004-11-07)

Corresponds to the version of the zope.schema package shipped as part of the Zope X3.0.0 release.

CHAPTER 7

Hacking on `zope.schema`

7.1 Getting the Code

The main repository for `zope.schema` is in the Zope Foundation Github repository:

https://github.com/zopefoundation/zope.schema

You can get a read-only checkout from there:

```
$ git clone https://github.com/zopefoundation/zope.schema.git
```

or fork it and get a writeable checkout of your fork:

```
$ git clone git@github.com/jrandom/zope.schema.git
```

The project also mirrors the trunk from the Github repository as a Bazaar branch on Launchpad:

https://code.launchpad.net/zope.schema

You can branch the trunk from there using Bazaar:

```
$ bzr branch lp:zope.schema
```

7.2 Working in a `virtualenv`

7.2.1 Installing

If you use the `virtualenv` package to create lightweight Python development environments, you can run the tests using nothing more than the `python` binary in a virtualenv. First, create a scratch environment:

```
$ /path/to/virtualenv --no-site-packages /tmp/hack-zope.schema
```

Next, get this package registered as a "development egg" in the environment:

```
$ /tmp/hack-zope.schema/bin/python setup.py develop
```

7.2.2 Running the tests

Run the tests using the build-in `setuptools` testrunner:

```
$ /tmp/hack-zope.schema/bin/python setup.py test
running test
........
----------------------------------------------------------------------
Ran 400 tests in 0.152s

OK
```

If you have the `nose` package installed in the virtualenv, you can use its testrunner too:

```
$ /tmp/hack-zope.schema/bin/easy_install nose
...
$ /tmp/hack-zope.schema/bin/python setup.py nosetests
running nosetests
.......
----------------------------------------------------------------------
Ran 400 tests in 0.152s

OK
```

or:

```
$ /tmp/hack-zope.schema/bin/nosetests
.......
----------------------------------------------------------------------
Ran 400 tests in 0.152s

OK
```

If you have the `coverage` pacakge installed in the virtualenv, you can see how well the tests cover the code:

```
$ /tmp/hack-zope.schema/bin/easy_install nose coverage
...
$ /tmp/hack-zope.schema/bin/python setup.py nosetests \
    --with coverage --cover-package=zope.schema
running nosetests
...
Name                                 Stmts   Miss  Cover   Missing
------------------------------------------------------------------
zope.schema                             43      0   100%
zope.schema._bootstrapfields           213      0   100%
zope.schema._bootstrapinterfaces        40      0   100%
zope.schema._compat                      4      0   100%
zope.schema._field                     425      0   100%
zope.schema._messageid                   2      0   100%
zope.schema._schema                     45      0   100%
zope.schema.accessors                   50      0   100%
zope.schema.fieldproperty               63      0   100%
zope.schema.interfaces                 156      0   100%
zope.schema.vocabulary                 166      0   100%
```

(continues on next page)

(continued from previous page)

```
----------------------------------------------------------------------
TOTAL                                1207      0   100%
----------------------------------------------------------------------
Ran 410 tests in 1.677s

OK
```

7.2.3 Building the documentation

`zope.schema` uses the nifty `Sphinx` documentation system for building its docs. Using the same virtualenv you set up to run the tests, you can build the docs:

```
$ /tmp/hack-zope.schema/bin/easy_install Sphinx
...
$ bin/sphinx-build -b html -d docs/_build/doctrees docs docs/_build/html
...
build succeeded.
```

You can also test the code snippets in the documentation:

```
$ bin/sphinx-build -b doctest -d docs/_build/doctrees docs docs/_build/doctest
...

Doctest summary
===============
  130 tests
    0 failures in tests
    0 failures in setup code
build succeeded.
Testing of doctests in the sources finished, look at the  \
    results in _build/doctest/output.txt.
```

7.3 Using `zc.buildout`

7.3.1 Setting up the buildout

`zope.schema` ships with its own `buildout.cfg` file and `bootstrap.py` for setting up a development buildout:

```
$ /path/to/python2.6 bootstrap.py
...
Generated script '.../bin/buildout'
$ bin/buildout
Develop: '/home/jrandom/projects/Zope/BTK/schema/.'
...
Generated script '.../bin/sphinx-quickstart'.
Generated script '.../bin/sphinx-build'.
```

7.3.2 Running the tests

Run the tests:

```
$ bin/test --all
Running zope.testing.testrunner.layer.UnitTests tests:
  Set up zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
  Ran 400 tests with 0 failures and 0 errors in 0.366 seconds.
Tearing down left over layers:
  Tear down zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
```

7.4 Using `tox`

7.4.1 Running Tests on Multiple Python Versions

tox is a Python-based test automation tool designed to run tests against multiple Python versions. It creates a `virtualenv` for each configured version, installs the current package and configured dependencies into each `virtualenv`, and then runs the configured commands.

`zope.schema` configures the following `tox` environments via its `tox.ini` file:

- The `py26`, `py27`, `py33`, `py34`, and `pypy` environments builds a `virtualenv` with `pypy`, installs `zope.schema` and dependencies, and runs the tests via `python setup.py test -q`.
- The `coverage` environment builds a `virtualenv` with `python2.6`, installs `zope.schema`, installs `nose` and `coverage`, and runs `nosetests` with statement coverage.
- The `docs` environment builds a virtualenv with `python2.6`, installs `zope.schema`, installs `Sphinx` and dependencies, and then builds the docs and exercises the doctest snippets.

This example requires that you have a working `python2.6` on your path, as well as installing `tox`:

```
$ tox -e py26
GLOB sdist-make: .../zope.interface/setup.py
py26 sdist-reinst: .../zope.interface/.tox/dist/zope.interface-4.0.2dev.zip
py26 runtests: commands[0]
..........
----------------------------------------------------------------------
Ran 400 tests in 0.152s

OK
_____________________________________ summary ______________________________________
py26: commands succeeded
congratulations :)
```

Running `tox` with no arguments runs all the configured environments, including building the docs and testing their snippets:

```
$ tox
GLOB sdist-make: .../zope.interface/setup.py
py26 sdist-reinst: .../zope.interface/.tox/dist/zope.interface-4.0.2dev.zip
py26 runtests: commands[0]
...
Doctest summary
===============
 140 tests
   0 failures in tests
   0 failures in setup code
   0 failures in cleanup code
```

(continues on next page)

(continued from previous page)

```
build succeeded.
_____________________________________ summary ______________________________________
py26: commands succeeded
py27: commands succeeded
py32: commands succeeded
pypy: commands succeeded
coverage: commands succeeded
docs: commands succeeded
congratulations :)
```

7.5 Contributing to zope.schema

7.5.1 Submitting a Bug Report

zope.schema tracks its bugs on Github:

https://github.com/zopefoundation/zope.schema/issues

Please submit bug reports and feature requests there.

7.5.2 Sharing Your Changes

Note: Please ensure that all tests are passing before you submit your code. If possible, your submission should include new tests for new features or bug fixes, although it is possible that you may have tested your new code by updating existing tests.

If have made a change you would like to share, the best route is to fork the Githb repository, check out your fork, make your changes on a branch in your fork, and push it. You can then submit a pull request from your branch:

https://github.com/zopefoundation/zope.schema/pulls

If you branched the code from Launchpad using Bazaar, you have another option: you can "push" your branch to Launchpad:

```
$ bzr push lp:~jrandom/zope.schema/cool_feature
```

After pushing your branch, you can link it to a bug report on Launchpad, or request that the maintainers merge your branch using the Launchpad "merge request" feature.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

z

Index

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O