
zope.schema Documentation

Release 4.5.1.dev0

Zope Foundation Contributors

Oct 09, 2017

Contents

1	Zope 3 Schemas	3
1.1	Introduction	3
1.2	Simple Usage	3
1.3	What is a schema, how does it compare to an interface?	5
1.4	Data Modeling Concepts	6
1.5	Fields and Widgets	6
1.6	References	7
2	Fields	9
2.1	Scalars	9
2.2	Collections	14
2.3	Creating a custom collection field	15
2.4	Choices and Vocabularies	15
2.5	Choices and Collections	16
2.6	Using Choice and Collection Fields within a Widget Framework	16
3	Sources	17
3.1	Concepts	17
3.2	Sources in Fields	17
4	Schema Validation	19
4.1	Compare ValidationError	21
5	API	23
5.1	Interfaces	23
5.2	Schema APIs	31
5.3	Fields	31
5.4	Accessors	36
6	Hacking on zope.schema	39
6.1	Getting the Code	39
6.2	Working in a virtualenv	39
6.3	Using zc.buildout	41
6.4	Using tox	42
6.5	Contributing to zope.schema	43
7	Indices and tables	45

Contents:

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.

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',
```

```
...         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.

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))
```

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.

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.

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`

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”).

Scalars

Scalar fields represent simple, immutable Python types.

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
```

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
```

BytesLine

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

ASCIILine

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

Float

zope.schema.Float fields contain binary data, represented as 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):
...
ValueError: invalid literal for float(): 1.25.6
```

Decimal

zope.schema.Decimal fields contain binary data, represented as 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):
...
ValueError: invalid literal for Decimal(): 1.25.6

```

Datetime

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

Date

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

TimeDelta

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

Time

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

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(['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.

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
```

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')

>>> 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'
```

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="Price Points",
...         unique=True,
...         value_type=Float(title="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 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
```

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.

Choices and Vocabularies

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

Simple choices do not have to explicitly use vocabularies:

```
>>> 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 registered vocabularies. 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.

The Zope application server typically needs a fuller 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 `IIterableVocabulary` interface.

If a vocabulary has been registered, then the choice merely needs to pass the vocabulary identifier to the “vocabulary” argument of the choice during instantiation.

A start to a vocabulary implementation that may do all you need for many simple tasks may be found in `zope.schema.vocabulary.SimpleVocabulary`. Because registered vocabularies are simply callables passed a context, many registered vocabularies can simply be functions that rely on `SimpleVocabulary`:

```
>>> from zope.schema.vocabulary import SimpleVocabulary
>>> def myDynamicVocabulary(context):
...     v = dynamic_context_calculation_that_returns_an_iterable(context)
...     return SimpleVocabulary.fromValues(v)
...
...
...

```

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.

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.

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`.

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 instantiates the source when it is actually used.

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
...     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 an `IContextSourceBinder` is passed as the `source` argument to `Choice`, its `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 initialization 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.

Schema Validation

There are two helper methods to verify schemas and interfaces:

`getValidationErrors()` first validates via the `zope.schema` field validators. If that succeeds the invariants are checked.

`getSchemaValidationErrors()` *only* validates via the `zope.schema` field validators. The invariants are *not* checked.

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:
...             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]
'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
```

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 than 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
```

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.

Interfaces

class `zope.schema.interfaces.IField`
Bases: `zope.interface.Interface`

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)
```

class `zope.schema.interfaces.IFromUnicode`
Bases: `zope.interface.Interface`

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.

class `zope.schema.interfaces.IChoice`
Bases: `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.

class `zope.schema.interfaces.IContextAwareDefaultFactory`

Bases: `zope.interface.Interface`

A default factory that requires a context.

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

class `zope.schema.interfaces.IOrderable`

Bases: `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.

class `zope.schema.interfaces.ILen`

Bases: `zope.schema.interfaces.IField`

A Field requiring its value to have a length.

The value needs to have a conventional `__len__` method.

class `zope.schema.interfaces.IMinMax`

Bases: `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.

class `zope.schema.interfaces.IMinMaxLen`

Bases: `zope.schema.interfaces.ILen`

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

class `zope.schema.interfaces.IInterfaceField`

Bases: `zope.schema.interfaces.IField`

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

class `zope.schema.interfaces.IBool`

Bases: `zope.schema.interfaces.IField`

Boolean Field.

class `zope.schema.interfaces.IObject`

Bases: `zope.schema.interfaces.IField`

Field containing an Object value.

class `zope.schema.interfaces.IDict`

Bases: `zope.schema.interfaces.IMinMaxLen`, `zope.schema.interfaces.IIterable`,
`zope.schema.interfaces.IContainer`

Field containing a conventional dict.

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

Strings

class `zope.schema.interfaces.IBytes`

Bases: `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.

class `zope.schema.interfaces.IBytesLine`

Bases: `zope.schema.interfaces.IBytes`

Field containing a byte string without newlines.

class `zope.schema.interfaces.IText`

Bases: `zope.schema.interfaces.IMinMaxLen`, `zope.schema.interfaces.IIterable`,
`zope.schema.interfaces.IField`

Field containing a unicode string.

class `zope.schema.interfaces.ITextLine`

Bases: `zope.schema.interfaces.IText`

Field containing a unicode string without newlines.

class `zope.schema.interfaces.IASCII`

Bases: `zope.schema.interfaces.IBytes`

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.interfaces.IASCIILine`

Bases: `zope.schema.interfaces.IASCII`

Field containing a 7-bit ASCII string without newlines.

class `zope.schema.interfaces.IPassword`

Bases: `zope.schema.interfaces.ITextLine`

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

class `zope.schema.interfaces.IURI`

Bases: `zope.schema.interfaces.IBytesLine`

A field containing an absolute URI

class `zope.schema.interfaces.IId`

Bases: `zope.schema.interfaces.IBytesLine`

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.

class `zope.schema.interfaces.IDottedName`

Bases: `zope.schema.interfaces.IBytesLine`

Dotted name field.

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

Numbers

- class** `zope.schema.interfaces.IInt`
Bases: `zope.schema.interfaces.IMinMax`, `zope.schema.interfaces.IField`
Field containing an Integer Value.
- class** `zope.schema.interfaces.IFloat`
Bases: `zope.schema.interfaces.IMinMax`, `zope.schema.interfaces.IField`
Field containing a Float.
- class** `zope.schema.interfaces.IDecimal`
Bases: `zope.schema.interfaces.IMinMax`, `zope.schema.interfaces.IField`
Field containing a Decimal.

Date/Time

- class** `zope.schema.interfaces.IDatetime`
Bases: `zope.schema.interfaces.IMinMax`, `zope.schema.interfaces.IField`
Field containing a datetime.
- class** `zope.schema.interfaces.IDate`
Bases: `zope.schema.interfaces.IMinMax`, `zope.schema.interfaces.IField`
Field containing a date.
- class** `zope.schema.interfaces.ITimedelta`
Bases: `zope.schema.interfaces.IMinMax`, `zope.schema.interfaces.IField`
Field containing a timedelta.
- class** `zope.schema.interfaces.ITime`
Bases: `zope.schema.interfaces.IMinMax`, `zope.schema.interfaces.IField`
Field containing a time.

Collections

- class** `zope.schema.interfaces.IIterable`
Bases: `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.)
- class** `zope.schema.interfaces.IContainer`
Bases: `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).
- class** `zope.schema.interfaces ICollection`
Bases: `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`.

class `zope.schema.interfaces.ISequence`

Bases: `zope.schema.interfaces ICollection`

Abstract interface specifying that the value is ordered

class `zope.schema.interfaces.IUnorderedCollection`

Bases: `zope.schema.interfaces ICollection`

Abstract interface specifying that the value cannot be ordered

class `zope.schema.interfaces.IAbstractSet`

Bases: `zope.schema.interfaces.IUnorderedCollection`

An unordered collection of unique values.

class `zope.schema.interfaces.IAbstractBag`

Bases: `zope.schema.interfaces.IUnorderedCollection`

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

class `zope.schema.interfaces.ITuple`

Bases: `zope.schema.interfaces.ISequence`

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

class `zope.schema.interfaces.IList`

Bases: `zope.schema.interfaces.ISequence`

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

class `zope.schema.interfaces.ISet`

Bases: `zope.schema.interfaces.IAbstractSet`

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

class `zope.schema.interfaces.IFrozenSet`

Bases: `zope.schema.interfaces.IAbstractSet`

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

Events

class `zope.schema.interfaces.IBeforeObjectAssignedEvent`

Bases: `zope.interface.Interface`

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.

class `zope.schema.interfaces.IFieldEvent`

Bases: `zope.interface.Interface`

class `zope.schema.interfaces.IFieldUpdatedEvent`

Bases: `zope.schema.interfaces.IFieldEvent`

A field has been modified

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

Vocabularies

class `zope.schema.interfaces.ITerm`
Bases: `zope.interface.Interface`

Object representing a single value in a vocabulary.

class `zope.schema.interfaces.ITokenizedTerm`
Bases: `zope.schema.interfaces.ITerm`

Object representing a single value in a tokenized vocabulary.

class `zope.schema.interfaces.ITitledTokenizedTerm`
Bases: `zope.schema.interfaces.ITokenizedTerm`

A tokenized term that includes a title.

class `zope.schema.interfaces.ISource`
Bases: `zope.interface.Interface`

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.

class `zope.schema.interfaces.ISourceQueriables`
Bases: `zope.interface.Interface`

A collection of objects for querying sources

class `zope.schema.interfaces.IContextSourceBinder`
Bases: `zope.interface.Interface`

class `zope.schema.interfaces.IBaseVocabulary`
Bases: `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).

class `zope.schema.interfaces.IIterableVocabulary`
Bases: `zope.interface.Interface`

Vocabulary which supports iteration over allowed values.

The objects iteration provides must conform to the `ITerm` interface.

class `zope.schema.interfaces.IIterableSource`
Bases: `zope.schema.interfaces.ISource`

Source which supports iteration over allowed values.

The objects iteration provides must be values from the source.

class `zope.schema.interfaces.IVocabulary`
Bases: `zope.schema.interfaces.IIterableVocabulary`, `zope.schema.interfaces.IBaseVocabulary`

Vocabulary which is iterable.

class `zope.schema.interfaces.IVocabularyTokenized`
Bases: `zope.schema.interfaces.IVocabulary`

Vocabulary that provides support for tokenized representation.

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

class `zope.schema.interfaces.ITreeVocabulary`

Bases: `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.

class `zope.schema.interfaces.IVocabularyRegistry`

Bases: `zope.interface.Interface`

Registry that provides `IBaseVocabulary` objects for specific fields.

class `zope.schema.interfaces.IVocabularyFactory`

Bases: `zope.interface.Interface`

Can create vocabularies.

Exceptions

exception `zope.schema._bootstrapinterfaces.ValidationError`

Bases: `zope.interface.exceptions.Invalid`

Raised if the Validation process fails.

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`

Bases: `zope.schema._bootstrapinterfaces.ValidationError`

Object is of wrong type.

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.TooSmall`
Bases: `zope.schema._bootstrapinterfaces.ValidationError`
Value is too small
- exception** `zope.schema.interfaces.TooBig`
Bases: `zope.schema._bootstrapinterfaces.ValidationError`
Value is too big
- exception** `zope.schema.interfaces.TooLong`
Bases: `zope.schema._bootstrapinterfaces.ValidationError`
Value is too long
- exception** `zope.schema.interfaces.TooShort`
Bases: `zope.schema._bootstrapinterfaces.ValidationError`
Value is too short
- exception** `zope.schema.interfaces.InvalidValue`
Bases: `zope.schema._bootstrapinterfaces.ValidationError`
Invalid value
- exception** `zope.schema.interfaces.WrongContainedType`
Bases: `zope.schema._bootstrapinterfaces.ValidationError`
Wrong contained type
- 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`
Bases: `zope.schema._bootstrapinterfaces.ValidationError`
Schema not provided
- 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.

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*, *object*)

Return a list of all validation errors.

`zope.schema.getSchemaValidationErrors` (*schema*, *object*)

Fields

```
class zope.schema.Field(title=u'', description=u'', __name__='', required=True, read-
                        only=False, constraint=None, default=None, defaultFactory=None, miss-
                        ing_value=<object object>)
```

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\n\nblah blah\nblah', 'sample', 'blah blah\nblah')
```

```
class zope.schema._field.AbstractCollection(value_type=None, unique=False, **kw)
```

Bases: `zope.schema._bootstrapfields.MinMaxLen`, `zope.schema._bootstrapfields.Iterable`

bind (*object*)

See `zope.schema._bootstrapinterfaces.IField`.

```
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.Bool` (*title=u'', description=u'', __name__='', required=True, readonly=False, constraint=None, default=None, defaultFactory=None, missing_value=<object object>*)
A field representing a Bool.

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 (*str*)

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

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 (*uc*)
See IFromUnicode.

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

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 (*object*)

See `zope.schema._bootstrapinterfaces.IField`.

fromUnicode (*str*)

See `IFromUnicode`.

class `zope.schema.Container` (*title=u'', description=u'', __name__='', required=True, read-only=False, constraint=None, default=None, defaultFactory=None, missing_value=<object object>*)

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.Decimal` (**args, **kw*)
Field containing a Decimal.

fromUnicode (*uc*)

See `IFromUnicode`.

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

bind (*object*)

See `zope.schema._bootstrapinterfaces.IField`.

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

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

class `zope.schema.Float` (**args, **kw*)
Field containing a Float.

fromUnicode (*uc*)
See IFromUnicode.

class `zope.schema.FrozenSet` (**kw)

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

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

fromUnicode (*value*)
See IFromUnicode.

class `zope.schema.Int` (**args, **kw*)
A field representing an Integer.

fromUnicode (*str*)

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

class `zope.schema.InterfaceField` (*title=u'', description=u'', __name__='', required=True, readonly=False, constraint=None, default=None, defaultFactory=None, missing_value=<object object>*)

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=<object object>*)

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=None, unique=False, **kw*)

Bases: `zope.schema._field.AbstractCollection`

A field representing a List.

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.

`zope.schema.NativeString`

alias of Bytes

`zope.schema.NativeStringLine`

alias of BytesLine

class `zope.schema.Object` (*schema, **kw*)

Field containing an Object value.

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.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.Set` (***kw*)

Bases: `zope.schema._field.AbstractCollection`

A field representing a set.

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 import Text
>>> t = Text(constraint=lambda v: 'x' in v)
>>> t.fromUnicode(b"foo x spam")
Traceback (most recent call last):
...
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):
...
ConstraintNotSatisfied: (u'foo spam', '')
```

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

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=None, unique=False, **kw*)
Bases: `zope.schema._field.AbstractCollection`
A field representing a Tuple.

class `zope.schema.URI` (*min_length=0, max_length=None, **kw*)
URI schema field
fromUnicode (*value*)
See IFromUnicode.

Accessors

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

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

Hacking on `zope.schema`

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:

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

Working in a `virtualenv`

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
```

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` package 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                               Stmt  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%
```

```

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

OK

```

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.

```

Using zc.buildout

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'.

```

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.
```

Using tox

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
build succeeded.
```

```
summary
py26: commands succeeded
py27: commands succeeded
py32: commands succeeded
pypy: commands succeeded
coverage: commands succeeded
docs: commands succeeded
congratulations :)
```

Contributing to zope . schema

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.

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 Github 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:

```
$ bazaar 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 7

Indices and tables

- `genindex`
- `modindex`
- `search`

Z

`zope.schema.accessors`, 36

A

AbstractCollection (class in zope.schema._field), 31
 accessors() (in module zope.schema.accessors), 37
 ASCII (class in zope.schema), 31
 ASCIIline (class in zope.schema), 32

B

bind() (zope.schema._field.AbstractCollection method), 31
 bind() (zope.schema.Choice method), 33
 bind() (zope.schema.Dict method), 33
 Bool (class in zope.schema), 32
 Bytes (class in zope.schema), 32
 BytesLine (class in zope.schema), 32

C

Choice (class in zope.schema), 32
 ConstraintNotSatisfied, 29
 Container (class in zope.schema), 33

D

Date (class in zope.schema), 33
 Datetime (class in zope.schema), 33
 Decimal (class in zope.schema), 33
 Dict (class in zope.schema), 33
 DottedName (class in zope.schema), 33

F

Field (class in zope.schema), 31
 FieldReadAccessor (class in zope.schema.accessors), 37
 Float (class in zope.schema), 33
 fromUnicode() (zope.schema.Bool method), 32
 fromUnicode() (zope.schema.Bytes method), 32
 fromUnicode() (zope.schema.Choice method), 33
 fromUnicode() (zope.schema.Decimal method), 33
 fromUnicode() (zope.schema.Float method), 33
 fromUnicode() (zope.schema.Id method), 34
 fromUnicode() (zope.schema.Int method), 34
 fromUnicode() (zope.schema.Text method), 36

fromUnicode() (zope.schema.URI method), 36
 FrozenSet (class in zope.schema), 34

G

getFieldNames() (in module zope.schema), 31
 getFieldNamesInOrder() (in module zope.schema), 31
 getFields() (in module zope.schema), 31
 getFieldsInOrder() (in module zope.schema), 31
 getSchemaValidationErrors() (in module zope.schema), 31
 getValidationErrors() (in module zope.schema), 31

I

IAbstractBag (class in zope.schema.interfaces), 27
 IAbstractSet (class in zope.schema.interfaces), 27
 IASCII (class in zope.schema.interfaces), 25
 IASCIIline (class in zope.schema.interfaces), 25
 IBaseVocabulary (class in zope.schema.interfaces), 28
 IBeforeObjectAssignedEvent (class in zope.schema.interfaces), 27
 IBool (class in zope.schema.interfaces), 24
 IBytes (class in zope.schema.interfaces), 25
 IBytesLine (class in zope.schema.interfaces), 25
 IChoice (class in zope.schema.interfaces), 23
 ICollection (class in zope.schema.interfaces), 26
 IContainer (class in zope.schema.interfaces), 26
 IContextAwareDefaultFactory (class in zope.schema.interfaces), 24
 IContextSourceBinder (class in zope.schema.interfaces), 28
 Id (class in zope.schema), 34
 IDate (class in zope.schema.interfaces), 26
 IDatetime (class in zope.schema.interfaces), 26
 IDecimal (class in zope.schema.interfaces), 26
 IDict (class in zope.schema.interfaces), 24
 IDottedName (class in zope.schema.interfaces), 25
 IField (class in zope.schema.interfaces), 23
 IFieldEvent (class in zope.schema.interfaces), 27
 IFieldUpdatedEvent (class in zope.schema.interfaces), 27

IFloat (class in zope.schema.interfaces), 26
IFromUnicode (class in zope.schema.interfaces), 23
IFrozenSet (class in zope.schema.interfaces), 27
IId (class in zope.schema.interfaces), 25
IInt (class in zope.schema.interfaces), 26
IInterfaceField (class in zope.schema.interfaces), 24
IIterable (class in zope.schema.interfaces), 26
IIterableSource (class in zope.schema.interfaces), 28
IIterableVocabulary (class in zope.schema.interfaces), 28
ILen (class in zope.schema.interfaces), 24
IList (class in zope.schema.interfaces), 27
IMinMax (class in zope.schema.interfaces), 24
IMinMaxLen (class in zope.schema.interfaces), 24
Int (class in zope.schema), 34
InterfaceField (class in zope.schema), 34
InvalidDottedName, 30
InvalidId, 30
InvalidURI, 30
InvalidValue, 30
IObject (class in zope.schema.interfaces), 24
IOrderable (class in zope.schema.interfaces), 24
IPassword (class in zope.schema.interfaces), 25
ISequence (class in zope.schema.interfaces), 27
ISet (class in zope.schema.interfaces), 27
ISource (class in zope.schema.interfaces), 28
ISourceQueriables (class in zope.schema.interfaces), 28
Iterable (class in zope.schema), 34
ITerm (class in zope.schema.interfaces), 28
IText (class in zope.schema.interfaces), 25
ITextLine (class in zope.schema.interfaces), 25
ITime (class in zope.schema.interfaces), 26
ITimedelta (class in zope.schema.interfaces), 26
ITitledTokenizedTerm (class in zope.schema.interfaces),
28
ITokenizedTerm (class in zope.schema.interfaces), 28
ITreeVocabulary (class in zope.schema.interfaces), 29
ITuple (class in zope.schema.interfaces), 27
IUnorderedCollection (class in zope.schema.interfaces),
27
IURI (class in zope.schema.interfaces), 25
IVocabulary (class in zope.schema.interfaces), 28
IVocabularyFactory (class in zope.schema.interfaces), 29
IVocabularyRegistry (class in zope.schema.interfaces), 29
IVocabularyTokenized (class in zope.schema.interfaces),
28

L

List (class in zope.schema), 35

M

MinMaxLen (class in zope.schema), 35

N

NativeString (in module zope.schema), 35

NativeStringLine (in module zope.schema), 35
NotAContainer, 29
NotAnIterator, 29
NotUnique, 30

O

Object (class in zope.schema), 35
Orderable (class in zope.schema), 35

P

Password (class in zope.schema), 35

R

RequiredMissing, 29

S

SchemaNotFullyImplemented, 30
SchemaNotProvided, 30
Set (class in zope.schema), 35
set() (zope.schema.Password method), 35
SourceText (class in zope.schema), 36
StopValidation, 29

T

Text (class in zope.schema), 36
TextLine (class in zope.schema), 36
Time (class in zope.schema), 36
Timedelta (class in zope.schema), 36
TooBig, 30
TooLong, 30
TooShort, 30
TooSmall, 29
Tuple (class in zope.schema), 36

U

Unbound, 30
URI (class in zope.schema), 36

V

ValidationError, 29

W

WrongContainedType, 30
WrongType, 29

Z

zope.schema.accessors (module), 36
zope.schema.ValidationError, 29