flatland

Release 0.9.1.dev5+gd08c8cd

Jul 26, 2019

Contents

Flatland maps between rich, structured Python application data and the string-oriented flat namespace of web forms, key/value stores, text files and user input. Flatland provides a schema-driven mapping toolkit with optional data validation.

Flatland is great for:

- Collecting, validating, re-displaying and processing HTML form data
- Dealing with rich structures (lists, dicts, lists of dicts, etc.) in web data
- Validating JSON, YAML, and other structured formats
- Associating arbitrary Python types with JSON, .ini, or sys.argv members that would otherwise deserialize as simple strings.
- Reusing a single data schema for HTML, JSON APIs, RPC, ...

The core of the Flatland toolkit is a flexible and extensible declarative schema system representing many data types and structures.

A validation system and library of schema-aware validators is also provided, with rich i18n capabilities for use in HTML, network APIs and other environments where user-facing messaging is required.

CHAPTER 1

Contents

1.1 Overview

1.1.1 Philosophy

flatland's design stems from a few basic tenets:

- All input is suspect
- Input can come from multiple sources and interfaces
- Bad input isn't exceptional: it is expected

With flatland, you describe exactly what elements your form may contain. Forms extract and process only their known elements out of the `(key, value)` input data. Unexpected or malicious data will not be processed.

The description of forms and their fields is data-centric rather than HTML or interface-centric. In a flatland form schema, a password input field is simply a string, not a "PasswordInput" or the like. The decision about how to represent that field is left up to another layer entirely. Maybe you do want an `<input type="password">` control, or maybe `<input type="hidden">` in some cases, or sometimes the data is coming in as JSON. flatland can act as another type of M in your M/VC, MC, MVC or MTV.

Humans are imperfect and filling out forms will always be error-prone. flatland recognizes this and provides features to make error detection and correction part of the regular workflow of a form. By default, validation routines will consider every element of a form and mark all problematic fields, allowing users to take action on all issues at once.

1.1.2 Introduction

Field schemas define all possible fields the form may contain. A schema may a single field, a collection of fields, or an even richer structure. Nested mappings and lists of fields are supported, as well as compound fields and even more exotic types.

```
from flatland import Form, String

class SignInForm(Form):
    username = String
    password = String
```

Field schemas are long-lived objects similar to class definitions. The instantiations of a flatland schema are called data elements, a tree structure of data-holding objects. The elements of a flatland form may be initiated blank, using default values, or with values taken from your objects.

```
form = SignInForm.from_flat(request.POST)
if form.validate():
    logging.info(u"sign-in: %s" % form['username'].value)
    redirect('/app/')
else:
    render('login.html', form=form)
```

Elements are rich objects that validate and normalize input data as well as hold field-level error and warning messages. Elements can be exported to a native Python structure, flattened back into Unicode (key, value) pairs or used as-is in output templates for form layout, redisplay and error reporting.

```
>>> as_regular_python_data = form.value
>>> isinstance(as_regular_python_data, dict)
True
>>> as_regular_python_data['username']
u'jek'
>>> form2 = SignInForm(as_regular_python_data)
>>> assert form['username'].value == form2['username'].value
```

1.2 Defining and Using Forms

1.2.1 Introduction

Basic Forms

class Schema(*value=Unspecified*, ***kw*)

Bases: *flatland.schema.containers.Dict*

A declarative collection of named elements.

Schemas behave like *Dict*, but are defined with Python class syntax:

```
>>> from flatland import Schema, String
>>> class HelloSchema(Schema):
...     hello = String
...     world = String
...
```

Elements are assigned names from the attribute declaration. If a named element schema is used, a renamed copy will be assigned to the Schema to match the declaration.

```
>>> class HelloSchema(Schema):
...     hello = String.named('hello')     # redundant
...     world = String.named('goodbye')   # will be renamed 'world'
```

(continues on next page)

(continued from previous page)

```
...
>>> helloworld = HelloSchema()
>>> sorted(helloworld.keys())
[u'hello', u'world']
```

Schemas may embed other container fields and other schemas:

```
>>> from flatland import List
>>> class BigSchema(Schema):
...     main_hello = HelloSchema
...     alt_hello = List.of(String.named('alt_name'),
...                         HelloSchema.named('alt_hello'))
...
```

This would create a Schema with one `HelloSchema` embedded as `main_hello`, and a list of zero or more dicts, each containing an `alt_name` and another `HelloSchema` named `alt_hello`.

Schemas may inherit from other Schemas or Dicts. Element attributes declared in a subclass will override those of a superclass. Multiple inheritance is supported.

The special behavior of `Schema` is limited to class construction time only. After construction, the `Schema` acts exactly like a *`Dict`*. In particular, fields declared as class attributes as above do **not** remain class attributes. They are removed from the class dictionary and placed in the *`field_schema`*:

```
>>> hasattr(HelloSchema, 'hello')
False
>>> sorted([field.name for field in HelloSchema.field_schema])
[u'hello', u'world']
```

The order of `field_schema` is undefined.

class SparseSchema(*value=Unspecified*, ***kw*)

Bases: *`flatland.schema.containers.SparseDict`*

A sparse variant of *`Schema`*.

Exactly as `Schema`, but based upon ~flatland.schema.containers.SparseDict‘.

class Form(*value=Unspecified*, ***kw*)

Bases: *`flatland.schema.containers.Dict`*

An alias for Schema, for older flatland version compatibility.

Elements

Elements describe the possible fields of a form; their names, structure, Python types and rules for validation. A typical schema consists of at least one container type and one or more *scalar* types:

```
from flatland import Dict, String
SearchSchema = Dict.named('search').of(String.named(u'keywords'))
```

Todo: FIXME UPDATE:

FieldSchemas are a bit like Python `class` definitions: they need be defined only once and don’t do much on their own. `FieldSchema.create_element()` produces *`Elements`*; closely related objects that hold and manipulate form data. Much like a Python `class`, a single `FieldSchema` may produce an unlimited number of `Element` instances.

```
>>> form = SearchSchema({u'keywords': u'foo bar baz'})
>>> form.value
{u'keywords': u'foo bar baz'}
```

Todo: FIXME UPDATE:

FieldSchema instances may be freely composed and shared among many containers.

```
>>> from flatland import List
>>> ComposedSchema = Dict.of(SearchSchema,
...                          List.named(u'many_searches').of(SearchSchema))
>>> form = ComposedSchema()
>>> sorted(form.value.keys())
[u'many_searches', u'search']
```

Todo: FIXME UPDATE:

Elements can be supplied to template environments and used to great effect there: elements contain all of the information needed to display or redisplay a HTML form field, including errors specific to a field.

The *u*, *x*, *xa* and `el()` members are especially useful in templates and have shortened names to help preserve your sanity when used in markup.

Element

class Element(*value=Unspecified*, ***kw*)

Base class for form fields.

A data node that stores a Python and a text value plus added state.

Instance Attributes

parent

An owning element, or None if element is topmost or not a member of a hierarchy.

valid

errors

A list of validation error messages.

warnings

A list of validation warning messages.

Members

name = None

The string name of the element.

optional = False

If True, *validate()* will return True when no value has been set.

validators are not called for optional, empty elements.

validators = ()
A sequence of validators, invoked by *validate()*.

See *Validation*.

default = None
The default value of this element.

default_factory = None
A callable to generate default element values. Passed an element.

default_factory will be used preferentially over *default*.

ugettext = None
If set, provides translation support to validation messages.

See *Message Internationalization*.

ungettext = None
If set, provides translation support to validation messages.

See *Message Internationalization*.

value = None
The element's native Python value.

Only validation routines should write this attribute directly: use *set()* to update the element's value.

raw = Unset
The element's raw, unadapted value from input.

u = u''
A string representation of the element's value.

As in *value*, writing directly to this attribute should be restricted to validation routines.

properties = {}
A mapping of arbitrary data associated with the element.

named(*name*)
Return a class with `name` = *name*

Parameters **name** – a string or None.

Returns a new class

using(***overrides*)
Return a class with attributes set from ***overrides*.

Parameters ****overrides** – new values for any attributes already present on the class. A `TypeError` is raised for unknown attributes.

Returns a new class

validated_by(**validators*)
Return a class with validators set to **validators*.

Parameters ***validators** – one or more validator functions, replacing any validators present on the class.

Returns a new class

including_validators(**validators*, ***kw*)
Return a class with additional **validators*.

Parameters

- ***validators** – one or more validator functions
- **position** – defaults to -1. By default, additional validators are placed after existing validators. Use 0 for before, or any other list index to splice in *validators* at that point.

Returns a new class

with_properties(**iterable*, ***properties*)
Return a class with ***properties* set.

Param optional positional parameter, an iterable of property name / value pairs

Parameters ****properties** – property names and values as keyword arguments

Returns a new class

classmethod from_flat(*pairs*, ***kw*)
Return a new element with its value initialized from *pairs*.

Parameters ****kw** – passed through to the `element_type`.

This is a convenience constructor for:

```
element = cls(**kw)
element.set_flat(pairs)
```

classmethod from_defaults(***kw*)
Return a new element with its value initialized from field defaults.

Parameters ****kw** – passed through to the `element_type`.

This is a convenience constructor for:

```
element = cls(**kw)
element.set_default()
```

all_valid
True if this element and all children are valid.

root
The top-most parent of the element.

parents
An iterator of all parent elements.

path
An iterator of all elements from root to the Element, inclusive.

children
An iterator of immediate child elements.

all_children
An iterator of all child elements, breadth-first.

fq_name()
Return the fully qualified path name of the element.

Returns *find()* compatible element path string from the *Element.root* (/) down to the element.

```
>>> from flatland import Dict, Integer
>>> Point = Dict.named(u'point').of(Integer.named(u'x'),
...                                 Integer.named(u'y'))
>>> p = Point(dict(x=10, y=20))
```

(continues on next page)

(continued from previous page)

```
>>> p.name
u'point'
>>> p.fq_name()
u'/'
>>> p['x'].name
u'x'
>>> p['x'].fq_name()
u'/x'
```

The index used in a path may not be the *name* of the element. For example, sequence members are referenced by their numeric index.

```
>>> from flatland import List, String
>>> Addresses = List.named('addresses').of(String.named('address'))
>>> form = Addresses([u'uptown', u'downtown'])
>>> form.name
u'addresses'
>>> form.fq_name()
u'/'
>>> form[0].name
u'address'
>>> form[0].fq_name()
u'/0'
```

find(*path*, *single=False*, *strict=True*)

Find child elements by string path.

Parameters

- **path** – a /-separated string specifying elements to select, such as 'child/grandchild/great grandchild'. Relative & absolute paths are supported, as well as container expansion. See *Path Lookups*.
- **single** – if true, return a scalar result rather than a list of elements. If no elements match *path*, `None` is returned. If multiple elements match, a `LookupError` is raised. If multiple elements are found and *strict* is false, an unspecified element from the result set is returned.
- **strict** – defaults to True. If *path* specifies children or sequence indexes that do not exist, a `LookupError` is raised.

Returns a list of *Element* instances, an *Element* if *single* is true, or raises `LookupError`.

```
>>> cities = form.find('/contact/addresses[:]/city')
>>> [el.value for el in cities]
[u'Kingsport', u'Dunwich']
>>> form.find('/contact/name', single=True)
<String u'name'; value=u'Obed Marsh'>
```

find_one(*path*)

Find a single element at *path*.

An alias for *find()*. Equivalent to `find(path, single=True, strict=True)`.

add_error(*message*)

Register an error message on this element, ignoring duplicates.

add_warning(*message*)

Register a warning message on this element, ignoring duplicates.

flattened_name(*sep=u'_'*)

Return the element's complete flattened name as a string.

Joins this element's *path* with *sep* and returns the fully qualified, flattened name. Encodes all `Container` and other structures into a single string.

Example:

```
>>> import flatland
>>> form = flatland.List('addresses',
...                      flatland.String('address'))
>>> element = form()
>>> element.set([u'uptown', u'downtown'])
>>> element[0].value
u'uptown'
>>> element['0'].flattened_name()
u'addresses_0_address'
```

flatten(*sep=u'_'*, *value=<operator.attrgetter object>*)

Export an element hierarchy as a flat sequence of key, value pairs.

Parameters

- **sep** – a string, will join together element names.
- **value** – a 1-arg callable called once for each element. Defaults to a callable that returns the *u* of each element.

Encodes the element hierarchy in a *sep*-separated name string, paired with any representation of the element you like. The default is the text value of the element, and the output of the default *flatten()* can be round-tripped with *set_flat()*.

Given a simple form with a string field and a nested dictionary:

```
>>> from flatland import Schema, Dict, String
>>> class Nested(Schema):
...     contact = Dict.of(String.named(u'name'),
...                       Dict.named(u'address').
...                            of(String.named(u'email')))
...
>>> element = Nested()
>>> element.flatten()
[(u'contact_name', u''), (u'contact_address_email', u'')]
```

The value of each pair can be customized with the *value* callable:

```
>>> element.flatten(value=operator.attrgetter('u'))
[(u'contact_name', u''), (u'contact_address_email', u'')]
>>> element.flatten(value=lambda el: el.value)
[(u'contact_name', None), (u'contact_address_email', None)]
```

Solo elements will return a sequence containing a single pair:

```
>>> element['name'].flatten()
[(u'contact_name', u'')]
```

set(*obj*)

Process *obj* and assign the native and text values.

Attempts to adapt the given *obj* and assigns this element's *value* and *u* attributes in tandem. Returns True if the adaptation was successful.

If adaptation succeeds, *value* will contain the adapted native value and *u* will contain a text serialized version of it. A native value of None will be represented as u" in *u*.

If adaptation fails, *value* will be `None` and *u* will contain `str(value)` (or unicode), or `u''` for None.

```
>>> from flatland import Integer
>>> el = Integer()
>>> el.u, el.value
(u'', None)
```

```
>>> el.set('123')
True
>>> el.u, el.value
(u'123', 123)
```

```
>>> el.set(456)
True
>>> el.u, el.value
(u'456', 456)
```

```
>>> el.set('abc')
False
>>> el.u, el.value
(u'abc', None)
```

```
>>> el.set(None)
True
>>> el.u, el.value
(u'', None)
```

set_flat (*pairs*, *sep=u'_'*)
Set element values from pairs, expanding the element tree as needed.

Given a sequence of name/value tuples or a dict, build out a structured tree of value elements.

set_default ()
set() the element to the schema default.

is_empty
True if the element has no value.

validate (*state=None*, *recurse=True*)
Assess the validity of this element and its children.

Parameters

- **state** – optional, will be passed unchanged to all validator callables.
- **recurse** – if False, do not validate children.

Returns True or False.

Iterates through this element and all of its children, invoking each validation on each. Each element will be visited twice: once heading down the tree, breadth-first, and again heading back up in reverse order.

Returns True if all validations pass, False if one or more fail.

default_value
A calculated "default" value.

If *default_factory* is present, it will be called with the element as a single positional argument. The result of the call will be returned.

Otherwise, returns *default*.

When comparing an element's *value* to its default value, use this property in the comparison.

x

Sugar, the XML-escaped value of *u*.

xa

Sugar, the XML-attribute-escaped quoted value of *u*.

Traversal

Flatland supplies a rich set of tools for working with structured data. For this section, we'll use the following schema as an example. It is simple yet has a bit of variety in its structure.

```
from flatland import Form, Dict, List, String, Integer

class Annotation(Form):
    """A spot on a 2D surface."""
    title = String
    flags = List.of(Integer)
    location = Dict.of(Integer.named('x'),
                       Integer.named('y'))

sample_data = {
  'title': 'Interesting Spot',
  'flags': [1, 3, 5],
  'location': {'x': 10, 'y': 20},
}

ann1 = Annotation(sample_data, name=u'ann1')
```

Going Raw

You may not even need to use any of these traversal strategies in your application. An element's *value* is a full & recursive "export" of its native Python value. Many times this is sufficient.

```
>>> ann1['title']     # ann1 is a flatland structure
<String u'title'; value=u'Interesting Spot'>
>>> isinstance(ann1.value, dict)  # but its .value is not
True
>>> ann1.value == sample_data
True
```

Python Syntax

Containers elements such as `Form`, *Dict*, and *List* implement the Python methods you'd expect for their type. In most cases you may use them as if they were `dict` and `list` instances- the difference being that they always contain *Element* instances.

For example, `Form` and `Dict` can be indexed and used like `dict`:

```
>>> ann1['title'].value
u'Interesting Spot'
>>> ann1['location']['x'].value
10
>>> sorted(ann1['location'].items())
[(u'x', <Integer u'x'; value=10>), (u'y', <Integer u'y'; value=20>)]
>>> u'title' in ann1
True
```

And `List` and similar types can be used like lists:

```
>>> ann1['flags']
[<Integer None; value=1>, <Integer None; value=3>, <Integer None; value=5>]
>>> ann1['flags'][0].value
1
>>> ann1['flags'].value
[1, 3, 5]
>>> Integer(3) in ann1['flags']
True
>>> 3 in ann1['flags']
True
```

The final example is of special note: the value in the expression is not an `Element`. Most containers will accept native Python values in these types of expressions and convert them into a temporary `Element` for the operation. The example below is equivalent to the example above.

```
>>> ann1['flags'].member_schema(3) in ann1['flags']
True
```

Traversal Properties

Elements of all types support a core set of properties that allow navigation to related elements: *root*, *parents*, *children*, and *all_children*.

```
>>> list(ann1['flags'].children)
[<Integer None; value=1>, <Integer None; value=3>, <Integer None; value=5>]
>>> list(ann1['title'].children)  # title is a String and has no children
[]
>>> sorted(el.name for el in ann1.all_children if el.name)
[u'flags', u'location', u'title', u'x', u'y']
>>> [el.name for el in ann1['location']['x'].parents]
[u'location', u'ann1']
```

Each of these properties (excepting `root`) returns an iterator of elements.

Path Lookups

Another option for operating on elements is the *find()* method. `find` selects elements by *path*, a string that represents one or more related elements. Looking up elements by path is a powerful technique to use when authoring flexible & reusable validators.

```
>>> ann1.find('title')   # find 'ann1's child named 'title'
[<String u'title'; value=u'Interesting Spot'>]
```

Paths are evaluated relative to the element:

```
>>> ann1['location'].find('x')
[<Integer u'x'; value=10>]
```

Referencing parents is possible with `..`:

```
>>> ann1['location']['x'].find('../../title')
[<String u'title'; value=u'Interesting Spot'>]
```

Absolute paths begin with a `/`.

```
>>> ann1['location']['x'].find('/title')
[<String u'title'; value=u'Interesting Spot'>]
```

Members of sequences can be selected like any other child (their index number is their name), or you can use Python-like slicing:

```
>>> ann1.find('/flags/0')
[<Integer None; value=1>]
>>> ann1.find('/flags[0]')
[<Integer None; value=1>]
```

Full Python slice notation is supported as well. With slices, paths can select more than one element.

```
>>> ann1.find('/flags[:]')
[<Integer None; value=1>, <Integer None; value=3>, <Integer None; value=5>]
>>> ann1.find('/flags[1:]')
[<Integer None; value=3>, <Integer None; value=5>]
```

Further path operations are permissible after slices. A richer schema is needed to illustrate this:

```
>>> Points = List.of(List.of(Dict.of(Integer.named('x'),
...                                  Integer.named('y'))))
>>> p = Points([[dict(x=1, y=1), dict(x=2, y=2)],
...             [dict(x=3, y=3)]])
>>> p.find('[:][:]/x')
[<Integer u'x'; value=1>, <Integer u'x'; value=2>, <Integer u'x'; value=3>]
```

The equivalent straight Python to select the same set of elements is quite a bit more wordy.

Path Syntax

`/` (leading slash) Selects the root of the element tree

`element` The name of a child element

`element/child` `/` separates path segments

`..` Traverse to the parent element

`element[0]` For a sequence container element, select the 0th child

`element[:]` Select all children of a container element (need not be a sequence)

`element[1:5]` Select a slice of a sequence container's children

Annotations & Properties

Flatland provides two options for annotating schemas and data.

Standard Python

Element schemas are normal Python classes and can be extended in all of the usual ways. For example, you can add an attribute when subclassing:

```
from flatland import String

class Textbox(String):
    tooltip = 'Undefined'
```

Once an attribute has been added to an element class, its value can be overridden by further subclassing or, more compactly, with the *using()* schema constructor:

```
class Password(Textbox):
    tooltip = 'Enter your password'


Password = Textbox.using(tooltip='Enter your password')
assert Password.tooltip == 'Enter your password'
```

Both are equivalent, and the custom `tooltip` will be inherited by any subclasses of `Password`. Likewise, instances of `Password` will have the attribute as well.

```
el = Password()
assert el.tooltip == 'Enter your password'
```

And because the *Element()* constructor allows overriding any schema attribute by keyword argument, individual element instances can be constructed with own values, masking the value provided by their class.

```
password_match = Textbox(tooltip='Enter your password again')

assert password_match.tooltip == 'Enter your password again'
```

Properties

Another option for annotation is the *properties* mapping of element classes and instances. Unlike class attributes, almost any object you like can be used as the key in the mapping.

The unique feature of *properties* is data inheritance:

```
from flatland import String

# Textboxes are Strings with tooltips
Textbox = String.with_properties(tooltip='Undefined')

# A Password is a Textbox with a custom tooltip message
Password = Textbox.with_properties(tooltip='Enter your password')

assert Textbox.properties['tooltip'] == 'Undefined'
assert Password.properties['tooltip'] == 'Enter your password'
```

Annotations made on a schema are visible to itself and any subclasses, but not to its parents.

```
# Add disabled to all Textboxes
Textbox.properties['disabled'] = False

# disabled is inherited from Textbox
assert Password.properties['disabled'] is False

# changes in a subclass do not affect the parent
del Password.properties['disabled']
assert 'disabled' in Textbox.properties
```

Annotating With Properties

To create a new schema that includes additional properties, construct it with *with_properties()*:

```
Textbox = String.with_properties(tooltip='Undefined')
```

Or if the schema has already been created, manipulate its *properties* mapping:

```
class Textbox(String):
   pass

Textbox.properties['tooltip'] = 'Undefined'
```

The *properties* mapping is implemented as a view over the Element schema inheritance hierarchy. If annotations are added to a superclass such as *String*, they are visible immediately to all Strings and subclasses.

Private Annotations

To create a schema with completely unrelated properties, not inheriting from its superclass at all, declare it with *using()*:

```
Alone = Textbox.using(properties={'something': 'else'})
assert 'tooltip' not in Alone.properties
```

Or when subclassing longhand, construct a `Properties` collection explicitly.

```
from flatland import Properties

class Alone(Textbox):
   properties = Properties(something='else')

assert 'tooltip' not in Alone.properties
```

An instance may also have a private collection of properties. This can be done either at or after construction:

```
solo1 = Textbox(properties={'something': 'else'})

solo2 = Textbox()
solo2.properties = {'something': 'else'}

Textbox.properties['background_color'] = 'red'
```

(continues on next page)

(continued from previous page)

```
assert 'background_color' not in solo1.properties
assert 'background_color' not in solo2.properties
```

Exceptions

exception AdaptationError
Bases: exceptions.Exception

A value could not be coerced into native format.

1.2.2 Element Types

Strings, Numbers and Booleans

Strings

class String(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Scalar*

A regular old text string.

strip = True
If true, strip leading and trailing whitespace during conversion.

adapt(*value*)
Return a Python representation.

Returns a text value or None

If *strip* is true, leading and trailing whitespace will be removed.

serialize(*value*)
Return a text representation.

Returns a Unicode value or u'' if *value* is None

If *strip* is true, leading and trailing whitespace will be removed.

is_empty
True if the String is missing or has no value.

Numbers

class Integer(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Number*

Element type for Python's int.

type_
alias of __builtin__.int

format = u'%i'
u'%i'

class Long(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Number*

Element type for Python's long.

type_
alias of `__builtin__.long`

format = u'%i'
`u'%i'`

class Float(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Number*

Element type for Python's float.

type_
alias of `__builtin__.float`

format = u'%f'
`u'%f'`

class Decimal(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Number*

Element type for Python's Decimal.

type_
alias of `decimal.Decimal`

format = u'%f'
`u'%f'`

Booleans

class Boolean(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Scalar*

Element type for Python's `bool`.

true = u'1'
The text serialization for `True`: `u'1'`.

true_synonyms = (u'on', u'true', u'True', u'1')
A sequence of acceptable string equivalents for True.

Defaults to `(u'on', u'true', u'True', u'1')`

false = u''
The text serialization for `False`: `u''`.

false_synonyms = (u'off', u'false', u'False', u'0', u'')
A sequence of acceptable string equivalents for False.

Defaults to `(u'off', u'false', u'False', u'0', u'')`

adapt(*value*)
Coerce *value* to `bool`.

Returns a `bool` or `None`

If *value* is text, returns `True` if the value is in *true_synonyms*, `False` if in *false_synonyms* and `None` otherwise.

For non-text values, equivalent to `bool(value)`.

serialize(*value*)
Convert `bool(value)` to a canonical text representation.

Returns either `self.true` or `self.false`.

Dates and Times

class DateTime(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Temporal*

Element type for Python datetime.datetime.

Serializes to and from YYYY-MM-DD HH:MM:SS format.

type_
alias of `datetime.datetime`

class Date(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Temporal*

Element type for Python datetime.date.

Serializes to and from YYYY-MM-DD format.

type_
alias of `datetime.date`

class Time(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Temporal*

Element type for Python datetime.time.

Serializes to and from HH:MM:SS format.

type_
alias of `datetime.time`

Dicts

Todo: intro

`set()` Policy

Todo: strict, duck, etc.

Validation

If *descent_validators* is defined, these validators will be run first, before member elements are validated.

If *validators* is defined, these validators will be run after member elements are validated.

Dict

class Dict(*value=Unspecified*, ***kw*)

Bases: *flatland.schema.containers.Mapping*, dict

A mapping Container with named members.

policy = 'subset'

Deprecated. One of 'strict', 'subset' or 'duck'. Default 'subset'.

Operates as *SetWithAllFields* and *SetWithKnownFields*, except raises an exception immediately upon *set()*.

To migrate to the new validators, set *policy* to None to disable the policy behavior.

of(**fields*)

Todo: doc of()

---

classmethod from_object(*obj*, *include=None*, *omit=None*, *rename=None*, ***kw*)

Return an element initialized with an object's attributes.

Parameters

- **obj** – any object
- **include** – optional, an iterable of attribute names to pull from *obj*, if present on the object. Only these attributes will be included.
- **omit** – optional, an iterable of attribute names to ignore on **obj**. All other attributes matching a named field on the mapping will be included.
- **rename** – optional, a mapping of attribute-to-field name transformations. Attributes specified in the mapping will be included regardless of *include* or *omit*.
- ****kw** – keyword arguments will be passed to the element's constructor.

include and *omit* are mutually exclusive.

This is a convenience constructor for *set_by_object()*:

```
element = cls(**kw)
element.set_by_object(obj, include, omit, rename)
```

set(*value*, *policy=None*)

Todo: doc set()

---

set_by_object(*obj*, *include=None*, *omit=None*, *rename=None*)

Set fields with an object's attributes.

Parameters

- **obj** – any object
- **include** – optional, an iterable of attribute names to pull from *obj*, if present on the object. Only these attributes will be included.

- **omit** – optional, an iterable of attribute names to ignore on **obj**. All other attributes matching a named field on the mapping will be included.
- **rename** – optional, a mapping of attribute-to-field name transformations. Attributes specified in the mapping will be included regardless of *include* or *omit*.

include and *omit* are mutually exclusive.

Sets fields on *self*, using as many attributes as possible from *obj*. Object attributes that do not correspond to field names are ignored.

Mapping instances have two corresponding methods useful for round-tripping values in and out of your domain objects.

update_object() performs the inverse of `set_object()`, and *slice()* is useful for constructing new objects.

```
>>> user = User('biff', 'secret')
>>> form = UserForm()
>>> form.set_by_object(user)
>>> form['login'].value
u'biff'
>>> form['password'] = u'new-password'
>>> form.update_object(user, omit=['verify_password'])
>>> user.password
u'new-password'
>>> user_keywords = form.slice(omit=['verify_password'], key=str)
>>> sorted(user_keywords.keys())
['login', 'password']
>>> new_user = User(**user_keywords)
```

update_object(*obj*, *include=None*, *omit=None*, *rename=None*, *key=<function identifier_transform>*)
Update an object's attributes using the element's values.

Produces a *slice()* using *include*, *omit*, *rename* and *key*, and sets the selected attributes on *obj* using `setattr`.

Returns nothing. *obj* is modified directly.

slice(*include=None*, *omit=None*, *rename=None*, *key=None*)
Return a `dict` containing a subset of the element's values.

SparseDict

class SparseDict(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.containers.Dict*

A Mapping which may contain a subset of the schema's allowed keys.

This differs from *Dict* in that new instances are not created with empty values for all possible keys. In addition, mutating operations are allowed so long as the operations operate within the schema. For example, you may *pop()* and `del` members of the mapping.

minimum_fields = None
The subset of fields to autovivify on instantiation.

May be `None` or `'required'`. If `None`, mappings will be created empty and mutation operations are unrestricted within the bounds of the `field_schema`. If `required`, fields with `optional` of `False`

will always be present after instantiation, and attempts to remove them from the mapping with `del` and friends will raise `TypeError`.

may_contain(*key*)
Return True if the element schema allows a field named **key**.

clear() → None. Remove all items from D.

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

set_default()
set() the element to the schema default.

Lists

Instances of *List* hold other elements and operate like Python lists. Lists are configured with a *member_schema*, such as an *Integer*. Each list member will be an instance of that type. The `List.of()` schema constructor will set `member_schema`:

```
>>> from flatland import List, Integer
>>> Numbers = List.of(Integer)
>>> Numbers.member_schema
<class 'flatland.schema.scalars.Integer'>
```

Python list methods and operators may be passed instances of *member_schema* or plain Python values. Using plain values is a shorthand for creating a `member_schema` instance and *set()ting* it with the value:

```
>>> ones = Numbers()
>>> ones.append(1)
>>> ones.value
[1]
>>> another_one = Integer()
>>> another_one.set(1)
True
>>> ones.append(another_one)
>>> ones.value
[1, 1]
```

List extends *Sequence* and adds positional naming to its elements. Elements are addressable via their list index in *find()* and their index in the list is reflected in their flattened name:

Example:

```
>>> from flatland import List, String
>>> Names = List.named('names').of(String.named('name'))
>>> names = Names([u'a', u'b'])
>>> names.value
[u'a', u'b']
>>> names.flatten()
[(u'names_0_name', u'a'), (u'names_1_name', u'b')]
>>> names[1].value
u'b'
```

(continues on next page)

(continued from previous page)

```
>>> names.find_one('1').value
u'b'
```

Validation

If *descent_validators* is defined, these validators will be run first, before member elements are validated.

If *validators* is defined, these validators will be run after member elements are validated.

List

class List(*value=Unspecified*, ***kw*)

Bases: *flatland.schema.containers.Sequence*

An ordered, homogeneous Sequence.

slot_type

alias of ListSlot

member_schema = ()

An *Element* class for member elements.

See also the *of()* schema configuration method.

maximum_set_flat_members = 1024

Maximum list members set in a set_flat() operation.

Once this maximum of child members has been added, subsequent data will be dropped. This ceiling prevents denial of service attacks when processing Lists with prune_empty set to False; without it remote attackers can trivially exhaust memory by specifying one low and one very high index.

append(*value*)

Append *value* to end.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before appending.

extend(*iterable*)

Append *iterable* values to the end.

If values of *iterable* are not instances of *member_schema*, they will be wrapped in a new element of that type before extending.

pop([*index*]) → item – remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

insert(*index*, *value*)

Insert *value* at *index*.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before inserting.

remove(*value*)

Remove member with value *value*.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before searching for a matching element to remove.

sort(*cmp=None*, *key=None*, *reverse=False*)
L.sort(cmp=None, key=None, reverse=False) – stable sort *IN PLACE*; cmp(x, y) -> -1, 0, 1

reverse()
L.reverse() – reverse *IN PLACE*

set_default()
set() the element to the schema default.

List's set_default supports two modes for *default* values:

- If default is an integer, the List will be filled with that many elements. Each element will then have *set_default()* called on it.
- Otherwise if default has a value, the list will be `set()` with it.

Arrays and MultiValues

Array

class Array(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.containers.Sequence*

A transparent homogeneous Container, for multivalued form elements.

Arrays hold a collection of values under a single name, allowing all values of a repeated `(key, value)` pair to be captured and used. Elements are sequence-like.

MultiValue

class MultiValue(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.containers.Array*, *flatland.schema.scalars.Scalar*

A transparent homogeneous Container, for multivalued form elements.

MultiValues combine aspects of *Scalar* and *Sequence* fields, allowing all values of a repeated `(key, value)` pair to be captured and used.

MultiValues take on the name of their child and have no identity of their own when flattened. Elements are mostly sequence-like and can be indexed and iterated. However the *u* or *value* are scalar-like, and return values from the first element in the sequence.

u
The .u of the first item in the sequence, or u''.

value
The .value of the first item in the sequence, or None.

Enumerations

Constrained Types

class Constrained(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Scalar*

A scalar type with a constrained set of legal values.

Wraps another scalar type and ensures that a value *set()* is within bounds defined by *valid_value()*. If *valid_value()* returns false, the element is not converted and will have a *value* of None.

Constrained is a semi-abstract class that requires an implementation of *valid_value()*, either by subclassing or overriding on a per-instance basis through the constructor.

An example of a wrapper of int values that only allows the values of 1, 2 or 3:

```
>>> from flatland import Constrained, Integer
>>> def is_valid(element, value):
...     return value in (1, 2, 3)
...
>>> schema = Constrained.using(child_type=Integer, valid_value=is_valid)
>>> element = schema()
>>> element.set(u'2')
True
>>> element.value
2
>>> element.set(u'5')
False
>>> element.value is None
True
```

Enum is a subclass which provides a convenient enumerated wrapper.

child_type
The type of constrained value, defaulting to String.

alias of *String*

static valid_value(*element*, *value*)
Returns True if *value* for *element* is within the constraints.

This method is abstract. Override in a subclass or pass a custom callable to the *Constrained* constructor.

adapt(*value*)
Given any object *obj*, try to coerce it into native format.

Returns the native format or raises AdaptationError on failure.

This abstract method is called by `set()`.

serialize(*value*)
Given any object *obj*, coerce it into a text representation.

Returns **Must** return a Unicode text object, always.

No special effort is made to coerce values not of native or a compatible type.

This semi-abstract method is called by `set()`. The base implementation returns `str(obj)` (or unicode).

Enum

class Enum(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Constrained*

A scalar type with a limited set of allowed values.

By default values are *strings*, but can be of any type you like by customizing *child_type*.

valued(**enum_values*)
Return a class with `valid_values` = *enum_values*

Parameters ***enum_values** – zero or more values for *valid_values*.

Returns a new class

valid_values = ()
Valid element values.

Attempting to `set()` a value not present in *valid_values* will cause an adaptation failure, and `value` will be `None`.

valid_value(*element*, *value*)
True if *value* is within *valid_values*.

Compound Fields

class DateYYYYMMDD(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.compound.Compound*, *flatland.schema.scalars.Date*

compose()
Return a text, native tuple built from children's state.

Returns a 2-tuple of text representation, native value. These correspond to the `serialize_element()` and `adapt_element()` methods of *Scalar* objects.

For example, a compound date field may return a '-' delimited string of year, month and day digits and a `datetime.date`.

explode(*value*)
Given a compound value, assign values to children.

Parameters **value** – a value to be adapted and exploded

For example, a compound date field may read attributes from a `datetime.date` value and `set()` them on child fields.

The decision to perform type checking on *value* is completely up to you and you may find you want different rules for different compound types.

class JoinedString(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.containers.Array*, *flatland.schema.scalars.String*

A sequence container that acts like a compounded string such as CSV.

Marshals child element values to and from a single string:

```
>>> from flatland import JoinedString
>>> el = JoinedString(['x', 'y', 'z'])
>>> el.value
u'x,y,z'
>>> el2 = JoinedString('foo,bar')
>>> el2[1].value
u'bar'
>>> el2.value
u'foo,bar'
```

Only the joined representation is considered when flattening or restoring with `set_flat()`. JoinedStrings run validation after their children.

separator = u','
The string used to join children's *u* representations. Will also be used to split incoming strings, unless *separator_regex* is also defined.

separator_regex = None
Optional, a regular expression, used preferentially to split an incoming separated value into components. Used in combination with *separator*, a permissive parsing policy can be combined with a normalized representation, e.g.:

```
>>> import re
>>> schema = JoinedString.using(separator=', ',
...                             separator_regex=re.compile('\s*,\s*'))
...
>>> schema('a  ,  b,c,d').value
u'a, b, c, d'
```

member_schema
alias of *flatland.schema.scalars.String*

set(*value*)
Assign the native and Unicode value.

Attempts to adapt the given *iterable* and assigns this element's *value* and *u* attributes in tandem. Returns True if the adaptation was successful. See *Element.set()*.

Set must be supplied a Python sequence or iterable:

```
>>> from flatland import Integer, List
>>> Numbers = List.of(Integer)
>>> nums = Numbers()
>>> nums.set([1, 2, 3, 4])
True
>>> nums.value
[1, 2, 3, 4]
```

value
A read-only *separator*-joined string of child values.

u
A read-only *separator*-joined string of child values.

1.2.3 Advanced Usage

References

class Ref(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Scalar*

adapt(*value*)
Given any object *obj*, try to coerce it into native format.

Returns the native format or raises AdaptationError on failure.

This abstract method is called by `set()`.

serialize(*value*)
Given any object *obj*, coerce it into a text representation.

Returns **Must** return a Unicode text object, always.

No special effort is made to coerce values not of native or a compatible type.

This semi-abstract method is called by `set()`. The base implementation returns `str(obj)` (or unicode).

u
The text representation of the reference target.

value
The native value representation of the reference target.

1.3 Validation

1.3.1 Basic Validation

All elements support validation. The default, built-in validation logic is simple: if the element is empty, it is invalid. Otherwise it is valid.

If that sounds too simple don't worry- you can customize validation to suit your needs.

Validating an Element

```
>>> from flatland import String
>>> form = String()
>>> form.is_empty
True
>>> form.valid
Unevaluated
>>> form.validate()
False
>>> form.valid
False
```

Validation sets the `valid` attribute of each element it inspects. `validate()` may be invoked more than once.

```
>>> form.set('Squiznart')
True
>>> form.is_empty
False
>>> form.validate()
True
>>> form.valid
True
```

Note that default validation does not set any error messages that might be displayed to an interactive user. Messages are easily added through custom validation.

Validating Entire Forms At Once

`validate()` is recursive by default. Called on a parent node, it will descend through all of its children, validating each. If the parent or any one of its children are invalid, `validate` returns false. Note that recursion does **not** stop if it finds an invalid child: all children are evaluated, and each will have its `valid` attribute updated.

Optional Elements

If an element is marked as `optional`, it is exempt from validation when empty. With the default validation strategy, this effectively means that element can never be invalid. With custom validation, optional fields become more useful.

```
>>> from flatland import Dict, Integer
>>> schema = Dict.of(Integer.named('x'),
...                  Integer.named('y'),
...                  Integer.named('z').using(optional=True))
>>> form = schema(dict(x=1))
>>> form.validate()
False
>>> form.valid
True
>>> form['x'].valid
True
>>> form['y'].valid
False
>>> form['z'].valid
True
```

Validation Signals

The *`flatland.signals.validator_validated`* signal is emitted each time a validator evaluates an element. The signal's sender is the validator (or the symbol `flatland.validation.NotEmpty` for the default validation strategy). The signal also sends the `element`, the `state`, and the `result` of the validation function.

During development, it can be convenient to connect the *`validator_validated`* signal to a logging function to aid in debugging.

```
from flatland.signals import validator_validated

@validator_validated.connect
def monitor_validation(sender, element, state, result):
    # print or logging.debug validations as they happen:
    print("validation: %s(%s) valid == %r" % (
      sender, element.flattened_name(), result))
```

```
>>> from flatland import String
>>> form = String(name='surname')
>>> form.validate()
validation: NotEmpty(surname) valid == False
False
```

1.3.2 Custom Validation

The default validation support is useful for some tasks, however in many cases you will want to provide your own validation rules tailored to your schema.

Flatland provides a low level interface for custom validation logic, based on a simple callable. Also provided is a higher level, class-based interface that provides conveniences for messaging, i18n and validator reuse. A library of commonly needed validators is included.

Custom Validation Basics

To use custom validation, assign a list of one or more validators to a field's `validators` attribute. Each validator will be evaluated in sequence until a validator returns false or the list of validators is exhausted. If the list is exhausted and all have returned true, the element is considered valid.

A validator is a callable of the form:

`validator`(*element*, *state*) → bool

`element` is the element being validated, and `state` is the value passed into `validate()`, which defaults to `None`.

A typical validator will examine the `value` of the element:

```
def no_shouting(element, state):
    """Disallow ALL CAPS TEXT."""
    if element.value.isupper():
        return False
    else:
        return True

# Try out the validator
from flatland import String
form = String(validators=[no_shouting])
form.set('OH HAI')
assert not form.validate()
assert not form.valid
```

Validation Phases

There are two phases when validating an element or container of elements. First, each element is visited once descending down the container, breadth-first. Then each is visited again ascending back up the container.

The simple, scalar types such as `String` and `Integer` process their `validators` on the **descent** phase. The containers, such as `Form` and `List` process `validators` on the **ascent** phase.

The upshot of the phased evaluation is that container validators fire after their children, allowing container validation logic that considers the validity and status of child elements.

```
>>> from flatland import Dict, String
>>> def tattle(element, state):
...     print(element.name)
...     return True
...
>>> schema = (Dict.named('outer').
...                of(String.named('inner').
...                          using(validators=[tattle])).
...                using(validators=[tattle]))
>>> form = schema()
>>> form.validate()
inner
outer
True
```

Short-Circuiting Descent Validation

Descent validation can be aborted early by returning `SkipAll` or `SkipAllFalse` from a validator. Children will not be validated or have their `valid` attribute assigned. This capability comes in handy in a web environment when designing rich UIs.

Containers will run any validators in their `descent_validators` list during the descent phase. Descent validation is the only phase that may be short-circuited.

```
>>> from flatland import Dict, SkipAll, String
>>> def skip_children(element, state):
...     return SkipAll
...
>>> def always_fail(element, state):
...     return False
...
>>> schema = Dict.of(String.named('child').using(validators=[always_fail])).\
...               using(descent_validators=[skip_children])
>>> form = schema()
>>> form.validate()
True
>>> form['child'].valid
Unevaluated
```

Messaging

A form that fails to submit without a clear reason is frustrating. Messages may be stashed in the `errors` and `warnings` lists on elements. In your UI or template code, these can be used to flag individual form elements that failed validation and the reason(s) why.

```
def no_shouting(element, state):
    """Disallow ALL CAPS TEXT."""
    if element.value.isupper():
        element.errors.append("NO SHOUTING!")
        return False
    else:
        return True
```

See also `add_error()`, a wrapper around `errors.append` that ensures that identical messages aren't added to an element more than once.

A powerful and i18n-capable interface to validation and messaging is available in the higher level *Validation* API.

Normalization

If you want to tweak the element's `value` or `u` string representation, validators are free to assign directly to those attributes. There is no special enforcement of assignment to these attributes, however the convention is to consider them immutable outside of normalizing validators.

Validation `state`

`validate()` accepts an optional `state` argument. `state` can be anything you like, such as a dictionary, an object, or a string. Whatever you choose, it will be supplied to each and every validator that's called.

`state` can be a convenient way of passing transient information to validators that require additional information to make their decision. For example, in a web environment, one may need to supply the client's IP address or the logged-in user for some validators to function.

A dictionary is a good place to start if you're considering passing information in `state`. None of the validators that ship with flatland access `state`, so no worries about type conflicts there.

```
class User(object):
    """A mock website user class."""

    def check_password(self, plaintext):
        """Mock comparing a password to one stored in a database."""
        return plaintext == 'secret'

def password_validator(element, state):
    """Check that a field matches the user's current password."""
    user = state['user']
    return user.check_password(element.value)

from flatland import String
form = String(validators=[password_validator])
form.set('WrongPassword')
state = dict(user=User())
assert not form.validate(state)
```

Examining Other Elements

`Element` provides a rich API for accessing a form's members, an element's parents, children, etc. Writing simple validators such as requiring two fields to match is easy, and complex validations are not much harder.

```
def passwords_must_match(element, state):
    """Both password fields must match for a password change to succeed."""
    if element.value == element.find('../password2', single=True).value:
        return True
    element.errors.append("Passwords must match.")
    return False

from flatland import Form, String
class ChangePassword(Form):
    password = String.using(validators=[passwords_must_match])
    password2 = String
    new_password = String

form = ChangePassword()
form.set({'password': 'foo', 'password2': 'f00', 'new_password': 'bar'})
assert not form.validate()
assert form['password'].errors
```

Short-Circuiting Validation

To stop validation of an element & skip any remaining members of `flatland.Element.validators`, return `flatland.Skip` from the validator:

```
from flatland import Skip

```

(continues on next page)

(continued from previous page)

```
def succeed_early(element, state):
    return Skip

def always_fails(element, state):
    return False

from flatland import String
form = String(validators=[succeed_early, always_fails])
assert form.validate()
```

Above, `always_fails` is never invoked.

To stop validation early with a failure, simply return False.

1.3.3 Validator API

The `Validator` class implements the validator callable interface and adds conveniences for messaging, internationalization, and customization.

To use it, subclass `Validator` and implement `validate()`.

```
from flatland.validation import Validator

class NoShouting(Validator):
    """Disallow ALL CAPS TEXT."""

    has_shouting = "NO SHOUTING in %(label)s, please."

    def validate(self, element, state):
        if element.value.isupper():
            self.note_error(element, state, 'has_shouting')
            return False
        return True

from flatland import String

schema = String.using(validators=[NoShouting()])
```

Above is a `Validator` version of the basic Customizing Validators example. In this version, the `flatland.validation.Validator.note_error()` method allows the messaging to be separated from the validation logic. `note_error` has some useful features, including templating and automatic I18N translation.

Customizing Validators

The base constructor of the `Validator` class has a twist that makes customizing existing Validators on the fly a breeze. The constructor can be passed keyword arguments matching any class attribute, and they will be overridden on the instance.

```
schema = String.using(validators=[NoShouting(has_shouting='shh.')])
```

Subclassing achieves the same effect.

```
class QuietPlease(NoShouting):
    has_shouting = 'shh.'
```

(continues on next page)

(continued from previous page)

```
schema = String.using(validators=[QuietPlease()])
```

The validators that ship with Flatland place all of their messaging and as much configurable behavior as possible in class attributes to support easy customization.

Message Templating

Messages prepared by `Validator.note_error()` and `Validator.note_warning()` may be templated using keywords in the sprintf-style Python string format syntax.

Possible keys are taken from multiple sources. In order of priority:

- Keyword arguments sent to the `note_error` and `note_warning` methods.
- Elements of `state`, if `state` is dict-like or supports `[index]` access.
- Attributes of `state`.
- Attributes of the `Validator` instance.
- Attributes of the `element`.

Message Pluralization

Flatland supports `ngettext`-style message pluralization. For this style, messages are specified as a 3-tuple of (`singular message`, `plural message`, `n-key`). `n_key` is any valid templating keyword, and its value `n` will be looked up using the same resolution rules. If the value `n` equals 1, the singular form will be used. Otherwise the plural.

```
from flatland.validation import Validator

class MinLength(Validator):

    min_length = 2

    too_short = (
      "%(label)s must be at least one character long.",
      "%(label)s must be at least %(min_length)s characters long.",
      "min_length")

    def validate(self, element, state):
        if len(element.value) < self.min_length:
            self.note_error(element, state, "too_short")
            return False
        return True
```

Conditional pluralization functions with or without I18N configured.

Message Internationalization

Messages can be translated using gettext-compatible functions. Translation works in conjunction with message templating features: the message itself is translated, and strings substituted into the message are also translated individually.

Translation uses `ugettext` and optionally `ungettext` functions that you provide. You may place these functions in the `state`, place them on the `element` or its schema, or place them in Python's builtins.

An element's ancestry will be searched for these functions. If you like, you may assign them solely to the top-most element or its schema and they will be used to translate all of its child elements.

If you opt to supply `ugettext` but not `ungettext`, Flatland's built-in pluralization will kick in if a pluralized message is found. Flatland will choose the correct form internally, and the result will be fed through `ugettext` for translation.

Dynamic Messages

Dynamic generated messages can also take advantage of the templating and internationalization features. There are two options for dynamic messages through `Validator.note_error()` and `Validator.note_warning()`:

1. Supply the message directly to `note_error` using `message="..."` instead of a message key.
2. Messages looked up by key may also be callables. The callable will be invoked with `element` and `state`, and should return either a message string or a 3-tuple as described in pluralization.

The Validator Class

class Validator(***kw*)
Base class for fancy validators.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

note_error(*element*, *state*, *key=None*, *message=None*, ***info*)
Record a validation error message on an element.

Parameters

- **element** – An *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.
- **key** – semi-optional, default None. The name of a message-holding attribute on this instance. Will be used to `message = getattr(self, key)`.
- **message** – semi-optional, default None. A validation message. Use to provide a specific message rather than look one up by *key*.
- ****info** – optional. Additional data to make available to validation message string formatting.

Returns False

Either *key* or *message* is required. The message will have formatting expanded by *expand_message()* and be appended to *element.errors*.

Always returns False. This enables a convenient shorthand when writing validators:

```
from flatland.validation import Validator

class MyValidator(Validator):
    my_message = 'Oh noes!'

    def validate(self, element, state):
        if not element.value:
            return self.note_error(element, state, 'my_message')
        else:
            return True
```

note_warning(*element*, *state*, *key=None*, *message=None*, ***info*)

Record a validation warning message on an element.

Parameters

- **element** – An *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.
- **key** – semi-optional, default None. The name of a message-holding attribute on this instance. Will be used to `message = getattr(self, key)`.
- **message** – semi-optional, default None. A validation message. Use to provide a specific message rather than look one up by *key*.
- ****info** – optional. Additional data to make available to validation message string formatting.

Returns False

Either *key* or *message* is required. The message will have formatting expanded by *expand_message()* and be appended to *element.warnings*.

Always returns False.

find_transformer(*type*, *element*, *state*, *message*)

Locate a message-transforming function, such as ugettext.

Returns None or a callable. The callable must return a message. The call signature of the callable is expected to match `ugettext` or `ungettext`:

- If *type* is 'ugettext', the callable should take a message as a positional argument.
- If *type* is 'ungettext', the callable should take three positional arguments: a message for the singular form, a message for the plural form, and an integer.

Subclasses may override this method to provide advanced message transformation and translation functionality, on a per-element or per-message granularity if desired.

The default implementation uses the following logic to locate a transformer:

1. If *state* has an attribute or item named *type*, return that.
2. If the *element* or any of its parents have an attribute named *type*, return that.
3. If the schema of *element* or the schema of any of its parents have an attribute named *type*, return that.
4. If *type* is in `builtins`, return that.

5. Otherwise return `None`.

expand_message(*element*, *state*, *message*, ***extra_format_args*)
Apply formatting to a validation message.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.
- **message** – a string, 3-tuple or callable. If a 3-tuple, must be of the form ('single form', 'plural form', n_key).

  If callable, will be called with 2 positional arguments (*element*, *state*) and must return a string or 3-tuple.
- ****extra_format_args** – optional. Additional data to make available to validation message string formatting.

Returns the formatted string

See *Message Templating*, *Message Pluralization* and *Message Internationalization* for full information on how messages are expanded.

1.3.4 Included Validators

Scalars

class Present(***kw*)
Bases: *flatland.validation.base.Validator*

Validates that a value is present.

Messages

missing
Emitted if the *u* string value of the element is empty, as in the case for an HTML form submitted with an input box left blank.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class IsTrue(***kw*)
Bases: *flatland.validation.base.Validator*

Validates that a value evaluates to true.

Messages

false
Emitted if `bool(element.value)` is not True.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class IsFalse(***kw*)
Bases: *flatland.validation.base.Validator*

Validates that a value evaluates to false.

Messages

true
Emitted if `bool(element.value)` is not False.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ValueIn(*valid_options=Unspecified*, ***kw*)
Bases: *flatland.validation.base.Validator*

Validates that the value is within a set of possible values.

Example:

```
import flatland
from flatland.validation import ValueIn

is_yesno = ValueIn(valid_options=['yes', 'no'])
schema = flatland.String('yn', validators=[is_yesno])
```

Attributes

valid_options
A list, set, or other container of valid element values.

Messages

fail
Emitted if the element's value is not within the *valid_options*.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class Converted(**kw*)
Bases: *flatland.validation.base.Validator*

Validates that an element was converted to a Python value.

Example:

```
import flatland
from flatland.validation import Converted

not_bogus = Converted(incorrect='Please enter a valid date.')
schema = flatland.DateTime('when', validators=[not_bogus])
```

Messages

incorrect
Emitted if the *value* is None.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ShorterThan(*maxlength=Unspecified*, **kw*)
Bases: *flatland.validation.base.Validator*

Validates the length of an element's string value is less than a bound.

Example:

```
import flatland
from flatland.validation import ShorterThan

valid_length = ShorterThan(8)
schema = flatland.String('password', validators=[valid_length])
```

Attributes

maxlength
A maximum character length for the *u*.

This attribute may be supplied as the first positional argument to the constructor.

Messages

exceeded
Emitted if the length of the element's string value exceeds *maxlength*.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

NoLongerThan
alias of *flatland.validation.scalars.ShorterThan*

class LongerThan(*minlength=Unspecified*, ***kw*)
Bases: *flatland.validation.base.Validator*

Validates the length of an element's string value is more than a bound.

Example:

```
import flatland
from flatland.validation import LongerThan

valid_length = LongerThan(4)
schema = flatland.String('password', validators=[valid_length])
```

Attributes

minlength
A minimum character length for the *u*.

This attribute may be supplied as the first positional argument to the constructor.

Messages

short
Emitted if the length of the element's string value falls short of *minlength*.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class LengthBetween(*minlength=Unspecified*, *maxlength=Unspecified*, ***kw*)
Bases: *flatland.validation.base.Validator*

Validates the length of an element's string value is within bounds.

Example:

```
import flatland
from flatland.validation import LengthBetween

valid_length = LengthBetween(4, 8)
schema = flatland.String('password', validators=[valid_length])
```

Attributes

minlength
A minimum character length for the *u*.

This attribute may be supplied as the first positional argument to the constructor.

maxlength
A maximum character length for the *u*.

This attribute may be supplied as the second positional argument to the constructor.

Messages

breached
Emitted if the length of the element's string value is less than *minlength* or greater than *maxlength*.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ValueLessThan(*boundary*, ***kw*)

Bases: *flatland.validation.base.Validator*

A validator that ensures that the value is less than a limit.

Example:

```
import flatland
from flatland.validation import ValueLessThan

schema = flatland.Integer('wishes', validators=[ValueLessThan(boundary=4)])
```

Attributes

boundary

Any comparable object.

Messages

failure

Emitted if the value is greater than or equal to *boundary*.

validate(*element*, *state*)

Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ValueAtMost(*maximum*, ***kw*)

Bases: *flatland.validation.base.Validator*

A validator that enforces a maximum value.

Example:

```
import flatland
from flatland.validation import ValueAtMost

schema = flatland.Integer('wishes', validators=[ValueAtMost(maximum=3)])
```

Attributes

maximum

Any comparable object.

Messages

failure

Emitted if the value is greater than *maximum*.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ValueGreaterThan(*boundary*, ***kw*)
Bases: *flatland.validation.base.Validator*

A validator that ensures that a value is greater than a limit.

Example:

```
import flatland
from flatland.validation import ValueGreaterThan

schema = flatland.Integer('wishes', validators=[ValueGreaterThan(boundary=4)])
```

Attributes

boundary
Any comparable object.

Messages

failure
Emitted if the value is greater than or equal to *boundary*.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ValueAtLeast(*minimum*, ***kw*)
Bases: *flatland.validation.base.Validator*

A validator that enforces a minimum value.

Example:

```
import flatland
from flatland.validation import ValueAtLeast

schema = flatland.Integer('wishes', validators=[ValueAtLeast(minimum=3)])
```

Attributes

minimum
Any comparable object.

Messages

failure
Emitted if the value is less than *minimum*.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ValueBetween(*minimum*, *maximum*, ***kw*)
Bases: *flatland.validation.base.Validator*

A validator that enforces minimum and maximum values.

Example:

```
import flatland
from flatland.validation import ValueBetween

schema = flatland.Integer('wishes',
                          validators=[ValueBetween(minimum=1, maximum=3)])
```

Attributes

minimum
Any comparable object.

maximum
Any comparable object.

inclusive
Boolean value indicating that *minimum* and *maximum* are included in the range. Defaults to True.

Messages

failure_inclusive
Emitted when *inclusive* is True if the expression *minimum* <= value <= *maximum* evaluates to False.

failure_exclusive
Emitted when *inclusive* is False if the expression *minimum* < value < *maximum* evaluates to False.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class MapEqual(**field_paths*, ***kw*)

Bases: *flatland.validation.base.Validator*

A general field equality validator.

Validates that two or more fields are equal.

Attributes

field_paths
A sequence of field names or field paths. Path names will be evaluated at validation time and relative path names are resolved relative to the element holding this validator. See *ValuesEqual* for an example.

transform
A 1-arg callable, passed a *Element*, returning a value for equality testing.

Messages

unequal
Emitted if the `transform(element)` of all elements are not equal. `labels` will substitute to a comma-separated list of the `label` of all but the last element; `last_label` is the label of the last.

Construct a MapEqual.

Parameters

- ***field_paths** – a sequence of 2 or more elements names or paths.
- ****kw** – passed to `Validator.__init__()`.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class ValuesEqual(**field_paths*, ***kw*)

Bases: *flatland.validation.scalars.MapEqual*

Validates that the values of multiple elements are equal.

A *MapEqual* that compares the *value* of each element.

Example:

```
from flatland import Schema, String
from flatland.validation import ValuesEqual

class MyForm(Schema):
    password = String
    password_again = String
    validators = [ValuesEqual('password', 'password_again')]
```

transform
attrgetter('value')

Construct a MapEqual.

Parameters

- ***field_paths** – a sequence of 2 or more elements names or paths.
- ****kw** – passed to `Validator.__init__()`.

class UnisEqual(**field_paths*, ***kw*)
Bases: *flatland.validation.scalars.MapEqual*

Validates that the Unicode values of multiple elements are equal.

A *MapEqual* that compares the *u* of each element.

transform
attrgetter('u')

Construct a MapEqual.

Parameters

- ***field_paths** – a sequence of 2 or more elements names or paths.
- ****kw** – passed to `Validator.__init__()`.

Containers

class NotDuplicated(***kw*)
Bases: *flatland.validation.base.Validator*

A sequence member validator that ensures all sibling values are unique.

Marks the second and any subsequent occurrences of a value as invalid. Only useful on immediate children of sequence fields such as `flatland.List`.

Example:

```
import flatland
from flatland.schema.containers import List
from flatland.validation import NotDuplicated

validator = NotDuplicated(failure="Please enter each color only once.")
schema = List.of(String.named('favorite_color')).\
             using(validators=[validator])
```

Attributes

comparator
A callable boolean predicate, by default `operator.eq`. Called positionally with two arguments, *element* and *sibling*.

Can be used as a filter, for example ignoring any siblings that have been marked as "deleted" by a checkbox in a web form:

```
from flatland import Schema, List, String, Integer, Boolean
from flatland.validation import NotDuplicated

def live_addrs(element, sibling):
    thisval, thatval = element.value, sibling.value
    # data marked as deleted is never considered a dupe
    if thisval['deleted'] or thatval['deleted']:
        return False
    # compare elements on 'street' & 'city', ignoring 'id'
    return (thisval['street'] == thatval['street'] and
            thisval['city'] == thatval['city'])

class Address(Schema):
    validators = [NotDuplicated(comparator=live_addrs)]

    id = Integer.using(optional=True)
    deleted = Boolean
    street = String
    city = String

schema = List.of(Address)
```

Messages

failure
Emitted on an element that has already appeared in a parent sequence. `container_label` will substitute the label of the container. `position` is the position of the element in the parent sequence, counting up from 1.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

comparator()
eq(a, b) – Same as a==b.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class HasAtLeast(***kw*)
Bases: *flatland.validation.base.Validator*

A sequence validator that ensures a minimum number of members.

May be applied to a sequence type such as a `List`.

Example:

```
from flatland import List, String
from flatland.validation import HasAtLeast

schema = List.of(String.named('wish')).\
              using(validators=[HasAtLeast(minimum=3)])
```

Attributes

minimum
Any positive integer.

Messages

failure
Emitted if the sequence contains less than *minimum* members. `child_label` will substitute the label of the child schema.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class HasAtMost(***kw*)
Bases: *flatland.validation.base.Validator*

A sequence validator that ensures a maximum number of members.

May be applied to a sequence type such as a `List`.

Example:

```
from flatland import List, String
from flatland.validation import HasAtMost

schema = List.of(String.named('wish')).\
              using(validators=[HasAtMost(maximum=3)])
```

Attributes

maximum
Any positive integer.

Messages

failure
Emitted if the sequence contains more than *maximum* members. `child_label` will substitute the label of the child schema.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class HasBetween(***kw*)
Bases: *flatland.validation.base.Validator*

Validates that the number of members of a sequence lies within a range.

May be applied to a sequence type such as a `List`.

Example:

```
from flatland import List, String
from flatland.validation import HasBetween

schema = List.of(String.named('wish')).\
              using(validators=[HasBetween(minimum=1, maximum=3)])
```

Attributes

minimum
Any positive integer.

maximum
Any positive integer.

Messages

range
Emitted if the sequence contains fewer than *minimum* members or more than *maximum* members. `child_label` will substitute the label of the child schema.

exact
Like *range*, however this message is emitted if *minimum* and *maximum* are the same.

```
schema = List.of(String.named('wish')).\
              using(validators=[HasBetween(minimum=3, maximum=3)])
```

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class SetWithKnownFields(**kw*)
Bases: *flatland.validation.base.Validator*

A mapping validator that ensures no unexpected fields were set().

May be applied to a mapping type such as a `Dict`.

Example:

```
from flatland import Dict, Integer
from flatland.validation import SetWithKnownFields

schema = Dict.of(Integer.named('x'), Integer.named('y')).\
             validated_by(SetWithKnownFields())
schema.policy = None
element = schema()

element.set({'x': 123, 'y': 456})
assert element.validate()

element.set({'x': 123, 'y': 456, 'z': 789})
assert not element.validate()
```

```
#from flatland import Dict, Integer
#from flatland.validation import SetWithKnownFields
#
#schema = Dict.of(Integer.named('x'), Integer.named('y')).\
#              validated_by(SetWithKnownFields())
#schema.policy = None
#element = schema()
#
#element.set({'x': 123})
#assert element.validate()  # assertion error, issue #25, FIXME!
#
#element.set({'x': 123, 'z': 789})
#assert not element.validate()  # no assertion error, but maybe due to #25 also.
```

This validator collects the keys from `raw` and compares them to the allowed keys for the element. Only elements in which `raw` is available and iterable will be considered for validation; all others are deemed valid.

Warning: This validator will not enforce policy on mappings initialized with `set_flat()` because raw is unset.

Note: This validator obsoletes and deprecates the `Dict.policy = 'subset'` feature. During the deprecation period, policy is still enforced by `set()` by default. To allow this validator to run, disable the default by

setting `policy = None` on the element or its schema.

Messages

unexpected
Emitted if the initializing value contains unexpected keys. `unexpected` will substitute a comma-separated list of unexpected keys, and `n_unexpected` a count of those keys.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

class SetWithAllFields(***kw*)
Bases: *flatland.validation.base.Validator*

A mapping validator that ensures all fields were set().

May be applied to a mapping type such as a `Dict`.

Example:

```
from flatland import Dict, Integer
from flatland.validation import SetWithAllFields

schema = Dict.of(Integer.named('x'), Integer.named('y')).\
             validated_by(SetWithAllFields())
schema.policy = None
element = schema()

element.set({'x': 123, 'y': 456})
assert element.validate()

element.set({'x': 123})
assert not element.validate()

element.set({'x': 123, 'y': 456, 'z': 789})
assert not element.validate()
```

This validator collects the keys from `raw` and compares them to the allowed keys for the element. Only elements in which `raw` is available and iterable will be considered for validation; all others are deemed valid.

Warning: This validator will not enforce policy on mappings initialized with `set_flat()` because raw is unset.

Note: This validator obsoletes and deprecates the `Dict.policy = 'strict'` feature. During the deprecation period, policy is still enforced by `set()` by default. To allow this validator to run, disable the default by setting `policy = None` on the element or its schema.

Messages

unexpected
Emitted if the initializing value contains unexpected keys. `unexpected` will substitute a comma-separated list of unexpected keys, and `n_unexpected` a count of those keys.

missing
Emitted if the initializing value did not contain all expected keys. `missing` will substitute a comma-separated list of missing keys, and `n_missing` a count of those keys.

both
Emitted if both of the previous conditions hold, and both sets of substitution keys are available.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

Numbers

class Luhn10(***kw*)
Bases: *flatland.validation.base.Validator*

True if a numeric value passes luhn10 checksum validation.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

validate(*element*, *state*)
Validate an element returning True if valid.

Abstract.

Parameters

- **element** – an *Element* instance.
- **state** – an arbitrary object. Supplied by *Element.validate*.

Returns True if valid

luhn10_check(*number*)
Return True if the number passes the Luhn checksum algorithm.

Email Addresses

class IsEmail(***kw*)
Bases: *flatland.validation.base.Validator*

Validates email addresses.

The default behavior takes a very permissive stance on allowed characters in the **local-part** and a relatively strict stance on the **domain**. Given **local-part@domain**:

- **local-part** must be present and contain at least one non-whitespace character. Any character is permitted, including international characters.
- **domain** must be preset, less than 253 characters and each dot-separated component must be 63 characters or less. **domain** may contain non-ASCII international characters, and will be converted to IDN representation before length assertions are applied. No top level domain validations are applied.

Attributes

non_local
Default `True`. When true, require at minimum two domain name components and reject local email addresses such as `postmaster@localhost` or `user@workstation`.

local_part_pattern
No default. If present, a compiled regular expression that will be matched to the **local-part**. Override this to implement more stringent checking such as RFC-compliant addresses.

domain_pattern
Defaults to a basic domain-validating regular expression with no notion of valid top level domains. Override this to require certain TLDs (or alternately and more simply, add another validator to your chain that checks the endings of the string against your list of TLDs.)

The default pattern rejects the valid but obscure quoted IP-address form (`[1.2.3.4]`).

Messages

invalid
Emitted if the email address is not valid.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

URLs

class URLValidator(***kw*)
Bases: *flatland.validation.base.Validator*

A general URL validator.

Validates that a URL is well-formed and may optionally restrict the set of valid schemes and other URL components.

Attributes

allowed_schemes
Restrict URLs to just this sequence of named schemes, or allow all schemes with ('*',). Defaults to all schemes. Example:

```
allowed_schemes = ('http', 'https', 'ssh')
```

allowed_parts
A sequence of 0 or more part names in `urlparse`'s vocabulary:

```
'scheme', 'netloc', 'path', 'params', 'query', 'fragment'
```

Defaults to all parts allowed.

urlparse
By default the `urlparse` module, but may be replaced by any object that implements `urlparse.urlparse()` and `urlparse.urlunparse()`.

Messages

bad_format
Emitted for an unparseable URL.

blocked_scheme
Emitted if the URL `scheme:` is not present in *allowed_schemes*.

blocked_part
Emitted if the URL has a component not present in *allowed_parts*.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

class HTTPURLValidator(***kw*)
Bases: *flatland.validation.base.Validator*

Validates `http` and `https` URLs.

Validates that an `http`-like URL is well-formed and may optionally require and restrict the permissible values of its components.

Attributes

all_parts
A sequence of known URL parts. Defaults to the full 10-tuple of names in `urlparse`'s vocabulary for HTTP-like URLs.

required_parts
A mapping of part names. If value is True, the part is required. The value may also be a sequence of strings; the value of the part must be present in this collection to validate.

The default requires a `scheme` of `'http'` or `'https'`.

forbidden_parts
A mapping of part names. If value is True, the part is forbidden and validation fails. The value may also be a sequence of strings; the value of the part must not be present in this collection to validate.

The default forbids `username` and `password` parts.

urlparse
By default the `urlparse` module, but may be replaced by any object that implements `urlparse.urlparse()` and `urlparse.urlunparse()`.

Messages

bad_format
Emitted for an unparseable URL.

required_part
Emitted if URL is missing a part present in *required_parts*.

forbidden_part
Emitted if URL contains a part present in *forbidden_parts*.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

class URLCanonicalizer(***kw*)
Bases: *flatland.validation.base.Validator*

A URL canonicalizing validator.

Given a valid URL, re-writes it with unwanted parts removed. The default implementation drops the `#fragment` from the URL, if present.

Attributes

discard_parts
A sequence of 0 or more part names in `urlparse`'s vocabulary:

```
'scheme', 'netloc', 'path', 'params', 'query', 'fragment'
```

urlparse
By default the `urlparse` module, but may be replaced by any object that implements `urlparse.urlparse()` and `urlparse.urlunparse()`.

Messages

bad_format
Emitted for an unparseable URL. This is impossible to hit with the Python's standard library implementation of urlparse.

Construct a validator.

Parameters ****kw** – override any extant class attribute on this instance.

1.4 HTML Forms and Markup

Flatland is not explicitly a form library, although it handles that task handily with powerful type conversion and validation error reporting. Dedicated form libraries often provide sophisticated "widget" or "control" features to render data fields as complex HTML markup with CSS and JavaScript support. The full expression of these features is outside Flatland's scope.

However! Properly generating HTML form tags and filling them with processed data is tedious and a common need, so Flatland ships with a minimalistic yet powerful toolset for tag generation. These tools are both highly usable as-is and also a solid base for constructing higher-level widgeting systems.

1.4.1 Markup Generation

The generation formula is simple: a desired tag (such as `input`) plus a flatland Element equals a complete HTML tag with attributes like `name` and `value` filled in using the element's state.

Operating on `Element` rather than the raw HTTP input or the polished final value provides a huge amount of expressive power. For example,

- Elements carry their validation errors- place these directly next to the form fields, or roll them up- your choice. Highlight failing fields with a `class="error"` CSS attribute right on the input element.
- (Re)populate form fields with exactly what the user typed, or the normalized version.
- Leverage the structure of the schema in template markup- if a form contains a list of 1 or more email addresses, loop over that list using your template language and render fields.
- Directly access Element properties and metadata, translation functions, and cross-element relations to implement complex view problems simply.

Flatland ships with two generator front-ends, both supporting the same features via a shared backend. The first, *Generator*, is for use in straight Python code, Jinja2, Mako, Genshi, or any other templating system. The second is a plugin for the Genshi templating library that integrates Flatland element binding directly and naturally into your existing <input/> tags.

Basic DWIM Binding

```
>>> from flatland.out.markup import Generator
>>> from flatland import Form, String
>>> html = Generator()
>>> class Login(Form):
...     username = String
...     password = String
...
>>> form = Login({'username': 'jek'})
```

Basic "Do What I Mean" form binding:

```
>>> print(html.input(form['username']))
<input name="username" value="jek" />
```

Likewise with Genshi.

```
<input form:bind="form.username"/>
```

and Genshi generates:

```
<input name="username" value="jek"/>
```

Attributes Too

Any HTML attribute can be included. Generated attributes can be overridden, too.

This time, the Generator is used in a Jinja2 template.

```
>>> from jinja2 import Template
>>> template = Template("""\
... {{ html.input(form.username, name="other", class_="custom") }}
... """)
>>> print(template.render(html=html, form=form))
<input name="other" value="jek" class="custom" />
```

These features are very similar in Genshi, too.

```
<input form:bind="form.username" name="other" class="custom"/>
```

Which generates the same output:

```
<input name="other" value="jek" class="custom"/>
```

Many Python templating systems allow you to replace the indexing operator (`form['username']`) with the attribute operator (`form.username`) to improve readability in templates. As shown above, this kind of rewriting trickery is generally not a problem for Flatland. Just keep name collisions in mind- if your form has a String field called `name`, is `form.name` the value of your form's name attribute or is it the String field? When writing macros or reusable functions, using the explicit `form[...]` index syntax is a good choice to protect against unexpected mangling by the template system no matter what the fields are named.

And More

The tag and attribute generation behavior can be configured and even post-processed just as you like it, affecting all of your tags, just one template, a block, or even individual tags.

1.4.2 Controlling Attribute Transformations

Out of the box, generation will do everything required for form element rendering and repopulation: filling `<textarea>`s, checking checkboxes, etc. Flatland can also generate some useful *optional* attributes, such as `id=` and `for=` linking for `<label>`s. Generation of attributes is controlled with markup options at several levels:

Global: Everything generated with a Generator instance or within a Genshi rendering operation.

Block: Options can be overridden within the scope of a block, reverting to their previous value at the end of the block.

Tag: Options can overridden on a per-tag basis.

Default: Finally, each tag has a set of sane default behaviors.

Boolean options may be True, or False, "on" or "off", or set to "auto" to revert to the transformation's built-in default setting.

1.4.3 Transformations

Most transforms require a Flatland element for context, such as setting an `input` tag's `value=` to the element's Unicode value. These tags can be said to be "bound" to the element.

Tags need not be bound, however. Here an unbound `textarea` can still participate in `tabindex=` generation.

```
>>> html = Generator(tabindex=100)
>>> print(html.textarea())
<textarea></textarea>
>>> print(html.textarea(auto_tabindex=True))
<textarea tabindex="100"></textarea>
>>> html.set(auto_tabindex=True)
u''
>>> print(html.textarea())
<textarea tabindex="101"></textarea>
```

Setting a boolean option to "on" or True on the tag itself will always attempt to apply the transform, allowing the transform to be applied to arbitrary tags that normally would not be transformed.

```
>>> print(html.tag('squiznart', auto_tabindex=True))
<squiznart tabindex="102" />
```

The Python APIs and the Generator tags use "_"-separated transform names (valid Python identifiers) as shown below, however please note that Genshi uses XML-friendly "-"-separated attribute names in markup.

auto-name

Default on

Tags button, form, input, select, textarea

Sets the tag `name=` to the bound element's `.name`. Takes no action if the tag already contains a `name=` attribute, unless forced.

Receives a `name=` attribute:

```
>>> print(html.input(form['username'], type="text"))
<input type="text" name="username" value="jek" />
```

Uses the explicitly provided `name="foo"`:

```
>>> print(html.input(form['username'], type="text", name='foo'))
<input type="text" name="foo" value="jek" />
```

Replaces `name="foo"` with the element's name:

```
>>> print(html.input(form['username'], type="text", name='foo', auto_name=True))
<input type="text" name="username" value="jek" />
```

auto-value

Default on

Tags button, input, select, textarea

Uses the bound element's `.u` Unicode value for the tag's value. The semantics of "value" vary by tag.

`<input>` types **text**, **hidden**, **button**, **submit** and **reset**:

Sets the `value=""` attribute of the tag, or omits the attribute if `.u` is the empty string.

Receives a `value=` attribute:

```
>>> print(html.input(form['username'], type="text"))
<input type="text" name="username" value="jek" />
```

Uses the explicitly provided `value="quux"`:

```
>>> print(html.input(form['username'], type="text", value='quux'))
<input type="text" name="username" value="quux" />
```

`<input>` types **password**, **image** and **file**:

No value is added unless forced by setting auto_value on the tag.

```
>>> print(html.input(form['password'], type="password"))
<input type="password" name="password" />
```

But this behavior can be forced:

```
>>> print(html.input(form['password'], type="password", auto_value=True))
<input type="password" name="password" value="secret" />
```

`<input>` type **radio**:

Radio buttons will add a `checked="checked"` attribute if the literal `value=` matches the element's value. Or, if the bind is a `Container`, `value=` will be compared against the `.u` of each of the container's children until a match is found.

If the tag lacks a `value=` attribute, no action is taken.

```
>>> print(form['username'].u)
jek
>>> print(html.input(form['username'], type="radio", value="quux"))
<input type="radio" name="username" value="quux" />
>>> print(html.input(form['username'], type="radio", value="jek"))
<input type="radio" name="username" value="jek" checked="checked" />
```

`<input>` type **checkbox**:

Check boxes will add a `checked="checked"` attribute if the literal `value=` matches the element's value.

```
>>> print(form['username'].u)
jek
>>> print(html.input(form['username'], type="checkbox", value="quux"))
<input type="checkbox" name="username" value="quux" />
>>> print(html.input(form['username'], type="checkbox", value="jek"))
<input type="checkbox" name="username" value="jek" checked="checked" />
```

Or, if the bind is a `Container`, `value=` will be compared against the `.u` of each of the container's children until a match is found.

```
>>> from flatland import Array
>>> Bag = Array.named('bag').of(String)
>>> bag = Bag(['a', 'c'])
>>> for value in 'a', 'b', 'c':
...     print(html.input(bag, type="checkbox", value=value))
...
<input type="checkbox" name="bag" value="a" checked="checked" />
<input type="checkbox" name="bag" value="b" />
<input type="checkbox" name="bag" value="c" checked="checked" />
```

If the tag lacks a `value=` attribute, no action is taken, unless the bind is a Boolean. The missing `value=` will be added using the schema's `Boolean.true` value.

```
>>> print(html.input(form['username'], type="checkbox"))
<input type="checkbox" name="username" />
>>> from flatland import Boolean
>>> toggle = Boolean.named('toggle')()
>>> print(html.input(toggle, type="checkbox"))
<input type="checkbox" name="toggle" value="1" />
>>> toggle.set(True)
True
>>> print(html.input(toggle, type="checkbox"))
<input type="checkbox" name="toggle" value="1" checked="checked" />
>>> toggle.true = "yes"
```

`<input>` types unknown:

For types unknown to flatland, no value is set unless forced by setting `form:auto-value="on"` on the tag.

`<textarea>`:

Textareas will insert the `Element.u` inside the tag pair. Content supplied with `contents=` for Generators or between Genshi tags will be preferred unless forced.

```
>>> print(html.textarea(form['username']))
<textarea name="username">jek</textarea>
>>> print(html.textarea(form['username'], contents="quux"))
<textarea name="username">quux</textarea>
```

Note that in Genshi, these two forms are equivalent.

```
<!-- these: -->
<textarea form:bind="form.username"/>
<textarea form:bind="form.username"></textarea>

<!-- will both render as -->
<textarea name="username">jek</textarea>
```

`<select>`:

Select tags apply a `selected="selected"` attribute to their `<option>` tags that match the `Element.u` or, if the bind is a `Container`, the `.u` of one of its children.

For this matching to work, the `<option>` tags must have a literal value set in the markup. The value may an explicit `value=` attribute, or it may be the text of the tag. Leading and trailing whitespace will be stripped when considering the text of the tag as the value.

The below will emit `selected="selected"` if form.field is equal to any of “a”, “b”, “c”, and “d”.

```
<select form:bind="form.field">
   <option>a</option>
   <option value="b"/>
   <option value="c">label</option>
   <option>
     d
   </option>
</select>
```

`<button/>` and `<button value=""/>`:

Regular <button/> tags will insert the Element.u inside the <button></button> tag pair. The output will **not** be XML-escaped, allowing any markup in the .u to render properly.

If the tag contains a literal value= attribute and a value override is forced by setting form:auto-value="on", the .u will be placed in the value= attribute, replacing the existing content. The value is escaped in this case.

```
<!-- set or replace the inner *markup* -->
<button form:bind="form.field"/>
<button form:bind="form.field" form:auto-value="on">xyz</button>

<!-- set the value, retaining the value= style used in the original -->
<button form:bind="form.field" value="xyz" form:auto-value="on"/>
```

auto-domid

Default off

Tags button, input, select, textarea

Sets the id= attribute of the tag. Takes no action if the markup already contains a id= unless forced by setting form:auto-domid="on".

The id is generated by combining the bound element's flattened_name with the domid-format in the current scope. The default format is **f_%s**.

auto-for

Default on

Tags label

Sets the for= attribute of the tag to the id of the bound element. The id is generated using the same process as *auto-domid*. No consistency checks are performed on the generated id value.

Defaults to "on", and will only apply if *auto-domid* is also "on". Takes no action if the markup already contains a id= unless forced by setting form:auto-for="on".

```
<form:with auto-domid="on">
  <fieldset py:with="field=form.field">
    <label form:bind="field">${field.label.x}</label>
    <input type="text" form:bind="field"/>
  </fieldset>
</form:with>
```

auto-tabindex

Default off

Tags button, input, select, textarea

Sets the tabindex attribute of tags with an incrementing integer.

Numbering starts at the scope's tabindex, which has no default. Assigning a value for tabindex will set the value for the next tabindex assignment, and subsequent assignments will increment by one.

A tabindex value of 0 will block the assignment of a tabindex and will not be incremented.

Takes no action if the markup already contains a tabindex= unless forced by setting form:auto-tabindex="on".

```
<form:with auto-tabindex="on" tabindex="1">
  <!-- assigns tabindex="1" -->
  <input type="text" form:bind="form.field"/>

  <!-- leaves existing tabindex in place -->
  <input type="text" tabindex="-1" form:bind="form.field"/>

  <!-- assigns tabindex="2" -->
  <a href="#" form:auto-tabindex="on"/>
</form:with>
```

1.4.4 Generator

class Generator(*markup=u'xhtml'*, ***settings*)

General XML/HTML tag generator

Create a generator.

Accepts any *Transformations*, as well as the following:

Parameters

- **markup** – tag output style: `'xml'`, `'xhtml'` or `'html'`
- **ordered_attributes** – if True (default), output markup attributes in a predictable order. Useful for tests and generally a little more pleasant to read.

begin(***settings*)

Begin a new *Transformations* context.

Puts **settings into effect until a matching *end()* is called. Each setting specified will mask the current value, reverting when *end()* is called.

end()

End a *Transformations* context.

Restores the settings that were in effect before *begin()*.

set(***settings*)

Change the *Transformations* in effect.

Change the **settings in the current scope. Changes remain in effect until another *set()* or a *end()* ends the current scope.

form

Generate a `<form/>` tag.

Parameters

- **bind** – optional, a flatland element.
- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

If provided with a bind, form tags can generate the *name* attribute.

input

Generate an `<input/>` tag.

Parameters

- **bind** – optional, a flatland element.

- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

If provided with a bind, input tags can generate the *name*, *value* and *id* attributes. Input tags support *tabindex* attributes.

textarea

Generate a <textarea/> tag.

Parameters

- **bind** – optional, a flatland element.
- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

If provided with a bind, textarea tags can generate the *name* and *id* attributes. If the bind has a value, it will be used as the tag body. Textarea tags support *tabindex* attributes. To provide an alternate tag body, either supply *contents* or use the *open()* and *close()* method of the returned tag.

button

Generate a <button/> tag.

Parameters

- **bind** – optional, a flatland element.
- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

If provided with a bind, button tags can generate the *name*, *value*, and *id* attributes. Button tags support *tabindex* attributes.

select

Generate a <select/> tag.

Parameters

- **bind** – optional, a flatland element.
- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

If provided with a bind, select tags can generate the *name* and *id* attributes. Select tags support *tabindex* attributes.

option

Generate an <option/> tag.

Parameters

- **bind** – optional, a flatland element.
- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

If provided with a bind, option tags can generate the *value* attribute. To provide tag body, either supply *contents* or use the *open()* and *close()* method of the returned tag:

```
print(generator.option.open(style='bold'))
print('<strong>contents</strong>')
print(generator.option.close())
```

label
Generate a <label/> tag.

Parameters

- **bind** – optional, a flatland element.
- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

If provided with a bind, label tags can generate the *for* attribute and fill in the tag body with the element's label, if present.

tag(*tagname*, *bind=None*, ***attributes*)
Generate any tag.

Parameters

- **tagname** – the name of the tag.
- **bind** – optional, a flatland element.
- ****attributes** – any desired XML/HTML attributes.

Returns a printable *Tag*

The attribute rules appropriate for *tagname* will be applied. For example, tag('input') is equivalent to input().

class Tag(*tagname*, *context*, *dangle*, *paired*)
A printable markup tag.

Tags are generated by *Generator* and are usually called immediately, returning a fully formed markup string:

```
print(generator.textarea(contents="hello!"))
```

For more fine-tuned control over your markup, you may instead choose to use the *open()* and *close()* methods of the tag:

```
print(generator.textarea.open())
print("hello!")
print(generator.textarea.close())
```

open(*bind=None*, ***attributes*)
Return the opening half of the tag, e.g. .

Parameters

- **bind** – optional, a flatland element.
- ****attributes** – any desired tag attributes.

close()
Return the closing half of the tag, e.g. .

1.4.5 Genshi Directives

```
http://ns.discorporate.us/flatland/genshi
```

1.5 Signals

Flatland can notify your code when events of interest occur during flatland processing using `Blinker` signals. These signals can be used for advanced customization in your application or simply as a means for tracing and logging flatland activity during development.

element_set = <blinker.base.NamedSignal object at 0x7f48bc53de50; 'element_set'>
Emitted after `set()` has been called on an element.

Parameters

- **sender** – the element
- **adapted** – boolean, True if the set() adapted successfully.

validator_validated = <blinker.base.NamedSignal object at 0x7f48bc53de90; 'validator_valida
Emitted after a validator has processed an element.

Parameters

- **sender** – the validator callable doing validation
- **element** – the element being validated
- **state** – the *state* passed to *validate()*
- **result** – the result of validator execution

1.6 Patterns

1.6.1 Widgets using Templates and Schema Properties

Unlike utilities more directly focused on processing Web forms, Flatland does not include any concept of "widgets" that render a field. It is however easy enough to employ Flatland's "properties" and markup generation support to build our own widget system. This also gives us complete control over the rendering.

```
from flatland import Form, String

Input = String.with_properties(widget='input', type='text')
Password = Input.with_properties(type='password')

class SignInForm(Form):
    username = Input.using(label='Username')
    password = Password.using(label='Password')
```

Rendering Widgets with Genshi

Macros via Genshi's `py:def` directive would be a good way to implement the actual widgets. For example:

```
<html
    xmlns:form="http://ns.discorporate.us/flatland/genshi"
    xmlns:py="http://genshi.edgewall.org/"
    py:strip=""
  >

  <py:def
```

(continues on next page)

(continued from previous page)

```
      function="widget(field)"
      py:with="macro = value_of(field.properties.widget + '_widget')"
      py:replace="macro(field)"
    />

  <fieldset py:def="input_widget(field)">
    <form:with
        auto-domid="on"
        auto-for="on"
      >

      <label
          form:bind="field"
          py:content="field.label"
        />

      <input
          form:bind="field"
          type="${field.properties.type}"
        />

    </form:with>
  </fieldset>
</html>
```

Typically we would call the `widget` macro manually for each field we want rendered, and in the desired order, but for demonstrative purposes we stub out widgets for each field in arbitrary order:

```
<html
    xmlns:py="http://genshi.edgewall.org/"
    xmlns:xi="http://www.w3.org/2001/XInclude"
  >
  <xi:include href="widgets.html"/>
  <body>
    <form>
      ${widget(form['username'])}
      ${widget(form['password'])}
    </form>
  </body>
</html>
```

Rendering with Jinja

If you're not using Genshi you can still benefit from Flatland's schema-aware markup generating support. With Jinja we might implement the macros as something resembling this:

```
{% set html = form_generator %}

{% macro widget(field) %}
  {%- set macro = {'input': input}[field.properties.widget] -%}
  {{- macro(field) -}}
{% endmacro %}

{% macro input(field) %}
  {%- do html.begin(auto_domid=true, auto_for=true) %}
```

(continues on next page)

(continued from previous page)

```
<fieldset>
  {{ html.label(field, contents=field.label) }}
  {{ html.input(field, type=field.properties.type) }}
</fieldset>
  {%- do html.end() %}
{% endmacro %}
```

Then we can simply import the `widget` macro to form templates:

```
{% from 'widgets.html' import widget -%}
<html>
  <body>
    <form>
      {{- widget(form['username']) }}
      {{- widget(form['password']) }}
    </form>
  </body>
</html>
```

Make sure to add a markup generator to the globals of your Jinja environment:

```
from flatland.out.markup import Generator
jinja_env.globals['form_generator'] = Generator('html')
```

1.7 API

class Scalar(*value=Unspecified*, ***kw*)

Bases: *flatland.schema.base.Element*

The base implementation of simple values such as a string or number.

Scalar subclasses are responsible for translating the most common data types in and out of Python-native form: strings, numbers, dates, times, Boolean values, etc. Any data which can be represented by a single `(name, value)` pair is a likely Scalar.

Scalar subclasses have two responsibilities: provide a method to adapt a value to native Python form, and provide a method to serialize the native form to a string.

This class is abstract.

set(*obj*)

Process *obj* and assign the native and text values.

Returns True if adaptation of *obj* was successful.

Attempts to adapt the given object and assigns this element's `value` and `u` attributes in tandem.

If adaptation succeeds, `.value` will contain the *adapted* native Python value and `.u` will contain a text *serialized* version of it. A native value of `None` will be represented as `u''` in `.u`.

If adaptation fails, `.value` will be `None` and `.u` will contain `str(obj)` (or unicode), or `u''` for none.

adapt(*obj*)

Given any object *obj*, try to coerce it into native format.

Returns the native format or raises AdaptationError on failure.

This abstract method is called by *set()*.

serialize(*obj*)
Given any object *obj*, coerce it into a text representation.

Returns **Must** return a Unicode text object, always.

No special effort is made to coerce values not of native or a compatible type.

This semi-abstract method is called by *set()*. The base implementation returns `str(obj)` (or unicode).

set_default()
set() the element to the schema default.

class Number(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Scalar*

Base for numeric fields.

Subclasses provide *type_* and *format* attributes for *adapt()* and *serialize()*.

type_ = None
The Python type for values, such as `int` or `float`.

signed = True
If true, allow negative numbers. Default `True`.

format = u'%s'
The `text` serialization format.

adapt(*value*)
Generic numeric coercion.

Returns an instance of *type_* or `None`

Attempt to convert *value* using the class's *type_* callable.

serialize(*value*)
Generic numeric serialization.

Returns Unicode text formatted with *format* or the `str()` (or unicode) of *value* if *value* is not of *type_*

Converts *value* to a string using Python's string formatting function and the *format* as the template. The *value* is provided to the format as a single, positional format argument.

class Temporal(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.scalars.Scalar*

Base for datetime-based date and time fields.

type_
Abstract. The native type for element values, will be called with positional arguments per *used* below.

regex
Abstract. A regular expression to parse datetime values from a string. Must supply named groupings.

used
Abstract. A sequence of regex match group names. These matches will be converted to ints and supplied to the *type_* constructor in the order specified.

format
Abstract. A Python string format for serializing the native value. The format will be supplied a dict containing all attributes of the native type.

adapt(*value*)
Coerces value to a native type.

If *value* is an instance of *type_*, returns it unchanged. If a string, attempts to parse it and construct a `type` as described in the attribute documentation.

serialize(*value*)
Serializes value to string.

If *value* is an instance of `type`, formats it as described in the attribute documentation. Otherwise returns `str(value)` (or unicode).

class Container(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.base.Element*

Holds other schema items.

Base class for elements that can contain other elements, such as *List* and *Dict*.

Parameters

- **descent_validators** – optional, a sequence of validators that will be run before contained elements are validated.
- **validators** – optional, a sequence of validators that will be run after contained elements are validated.
- ****kw** – other arguments common to *Element*.

descent_validators = ()

Todo: doc descent_validators

descent_validated_by(**validators*)
Return a class with descent validators set to **validators*.

Parameters ***validators** – one or more validator functions, replacing any descent validators present on the class.

Returns a new class

including_descent_validators(**validators*, ***kw*)
Return a class with additional descent **validators*.

Parameters

- ***validators** – one or more validator functions
- **position** – defaults to -1. By default, additional validators are placed after existing descent validators. Use 0 for before, or any other list index to splice in *validators* at that point.

Returns a new class

class Sequence(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.containers.Container*, `list`

Abstract base of sequence-like Containers.

Instances of *Sequence* hold other elements and operate like Python lists. Each sequence member will be an instance of *member_schema*.

Python list methods and operators may be passed instances of *member_schema* or plain Python values. Using plain values is a shorthand for creating an *member_schema* instance and *set()ting* it with the value:

```
>>> from flatland import Array, Integer
>>> Numbers = Array.of(Integer)
>>> ones = Numbers()
>>> ones.append(1)
>>> ones
[<Integer None; value=1>]
>>> another_one = Integer()
>>> another_one.set(1)
True
>>> ones.append(another_one)
>>> ones
[<Integer None; value=1>, <Integer None; value=1>]
```

member_schema = None
An *Element* class for sequence members.

prune_empty = True
If true, skip missing index numbers in `set_flat()`. Default True.

See **'Sequences'_** for more information.

of(**schema*)
Declare the class to hold a sequence of **schema*.

Params **schema* one or more *Element* classes

Returns *cls*

Configures the *member_schema* of *cls* to hold instances of **schema*.

```
>>> from flatland import Array, String
>>> Names = Array.of(String.named('name'))
>>> Names.member_schema
<class 'flatland.schema.scalars.String'>
>>> el = Names(['Bob', 'Biff'])
>>> el
[<String u'name'; value=u'Bob'>, <String u'name'; value=u'Biff'>]
```

If more than one *Element* is specified in **schema*, an anonymous *Dict* is created to hold them.

```
>>> from flatland import Integer
>>> Points = Array.of(Integer.named('x'), Integer.named('y'))
>>> Points.member_schema
<class 'flatland.schema.containers.Dict'>
>>> el = Points([dict(x=1, y=2)])
>>> point = el[0]
>>> point['x']
<Integer u'x'; value=1>
>>> point['y']
<Integer u'y'; value=2>
```

set(*iterable*)
Assign the native and Unicode value.

Attempts to adapt the given *iterable* and assigns this element's `value` and `u` attributes in tandem. Returns True if the adaptation was successful. See *Element.set()*.

Set must be supplied a Python sequence or iterable:

```
>>> from flatland import Integer, List
>>> Numbers = List.of(Integer)
>>> nums = Numbers()
>>> nums.set([1, 2, 3, 4])
True
>>> nums.value
[1, 2, 3, 4]
```

set_default()
set() the element to the schema default.

append(*value*)
Append *value* to end.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before appending.

extend(*iterable*)
Append *iterable* values to the end.

If values of *iterable* are not instances of *member_schema*, they will be wrapped in a new element of that type before extending.

insert(*index*, *value*)
Insert *value* at *index*.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before inserting.

remove(*value*)
Remove member with value *value*.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before searching for a matching element to remove.

index(*value*)
Return first index of *value*.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before searching for a matching element in the sequence.

count(*value*)
Return number of occurrences of *value*.

If *value* is not an instance of *member_schema*, it will be wrapped in a new element of that type before searching for matching elements in the sequence.

class Mapping(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.containers.Container*, dict

Base of mapping-like Containers.

field_schema = ()

Todo: doc field_schema

---

may_contain(*key*)
Return True if the element schema allows a field named **key**.

clear() → None. Remove all items from D.

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

update(**dictish*, ***kwargs*)
Update with keys from dict-like **dictish* and ***kwargs*

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

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

set(*value*)

Todo: doc set()

---

set_default()
set() the element to the schema default.

u
A string repr of the element.

value
The element as a regular Python dictionary.

is_empty
Mappings are never empty.

class Compound(*value=Unspecified*, ***kw*)
Bases: *flatland.schema.containers.Mapping*, *flatland.schema.scalars.Scalar*

A mapping container that acts like a scalar value.

Compound fields are dictionary-like fields that can assemble a `u` and `value` from their children, and can decompose a structured value passed to a *set()* into values for its children.

A simple example is a logical calendar date field composed of 3 separate Integer component fields, year, month and day. The Compound can wrap the 3 parts up into a single logical field that handles `datetime.date` values. Set a `date` on the logical field and its component fields will be set with year, month and day; alter the int value of the year component field and the logical field updates the `date` to match.

Compound is an abstract class. Subclasses must implement *compose()* and *explode()*.

Composites run validation after their children.

compose()
Return a text, native tuple built from children's state.

Returns a 2-tuple of text representation, native value. These correspond to the `serialize_element()` and `adapt_element()` methods of *Scalar* objects.

For example, a compound date field may return a '-' delimited string of year, month and day digits and a `datetime.date`.

explode(*value*)
Given a compound value, assign values to children.

Parameters **value** – a value to be adapted and exploded

For example, a compound date field may read attributes from a `datetime.date` value and *set()* them on child fields.

The decision to perform type checking on *value* is completely up to you and you may find you want different rules for different compound types.

serialize(*value*)
Not implemented for Compound types.

set(*value*)

Todo: doc set()

---

is_empty
True if all subfields are empty.

1.8 The Flatland Project

1.8.1 License

1.8.2 Authors & Contributors

Flatland was originally written by Jason Kirtland.

Contributors are:

- Dan Colish <dcolish@gmail.com>
- Jason Kirtland <jek@discorporate.us>
- Adam Lowry
- Dag Odenhall
- Michel Pelletier
- Ollie Rutherfurd
- Benjamin Stover
- Thomas Waldmann
- Scott Wilson

Portions derived from other open source works and are clearly marked.

1.8.3 History

Flatland is a Python implementation of techniques I've been using for form and web data processing for ages, in many different languages. It is an immediate conceptual descendant and re-write of "springy", a closed-source library used internally at Virtuous, Inc. The Genshi filter support was donated to the Flatland project by Virtuous.

1.8.4 Documentation Todo List

Todo: doc descent_validators

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/flatland/schema/containers.p
of flatland.schema.containers.Container.descent_validators, line 1.)

Todo: doc field_schema

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/flatland/schema/containers.p
of flatland.schema.containers.Mapping.field_schema, line 1.)

Todo: doc set()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/flatland/schema/containers.p
of flatland.schema.containers.Mapping.set, line 1.)

Todo: doc set()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/flatland/schema/compound.
of flatland.schema.compound.Compound.set, line 1.)

Todo: intro

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/docs/source/schema/dicts.rs
line 7.)

Todo: strict, duck, etc.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/docs/source/schema/dicts.rs
line 14.)

Todo: doc of()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/flatland/schema/containers.p
of flatland.schema.containers.Dict.of, line 1.)

Todo: doc set()

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/flatland/schema/containers.p of flatland.schema.containers.Dict.set, line 1.)

Todo: FIXME UPDATE:

FieldSchemas are a bit like Python `class` definitions: they need be defined only once and don't do much on their own. `FieldSchema.create_element()` produces *`Elements`*; closely related objects that hold and manipulate form data. Much like a Python `class`, a single `FieldSchema` may produce an unlimited number of `Element` instances.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/docs/source/schema/schema line 20.)

Todo: FIXME UPDATE:

FieldSchema instances may be freely composed and shared among many containers.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/docs/source/schema/schema line 36.)

Todo: FIXME UPDATE:

Elements can be supplied to template environments and used to great effect there: elements contain all of the information needed to display or redisplay a HTML form field, including errors specific to a field.

The *`u`*, *`x`*, *`xa`* and `el()` members are especially useful in templates and have shortened names to help preserve your sanity when used in markup.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/flatland/checkouts/latest/docs/source/schema/schema line 50.)

Python Module Index

f

Index

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X