Schematics Documentation

Release 2.1.0

Schematics Authors

Jun 26, 2018

Basics

Python Data Structures for Humans™.

CHAPTER 1

Install Guide

Tagged releases are available from PyPI:

```
$ pip install schematics
```

The latest development version can be obtained via git:

```
$ pip install git+https://github.com/schematics/schematics.git#egg=schematics
```

Schematics currently supports Python versions 2.7, 3.3, 3.4, 3.5 and 3.6.

1.1 Dependencies

The only dependency is six for Python 2+3 support.

1.2 Installing from GitHub

The canonical repository for Schematics is hosted on GitHub.

Getting a local copy is simple:

```
$ git clone https://github.com/schematics/schematics.git
```

If you are planning to contribute, first create your own fork of Schematics on GitHub and clone the fork:

```
$ git clone https://github.com/YOUR-USERNAME/schematics.git
```

Then add the main Schematics repository as another remote called *upstream*:

```
$ git remote add upstream https://github.com/schematics/schematics.git
```

See also *Developer's Guide*.

CHAPTER 2

Quickstart Guide

Working with Schematics begins with modeling the data, so this tutorial will start there.

After that we will take a quick look at serialization, validation, and what it means to save this data to a database.

2.1 Simple Model

Let's say we want to build a structure for storing weather data. At it's core, we'll need a way to represent some temperature information and where that temp was found.

```
import datetime
from schematics.models import Model
from schematics.types import StringType, DecimalType, DateTimeType

class WeatherReport(Model):
    city = StringType()
    temperature = DecimalType()
    taken_at = DateTimeType(default=datetime.datetime.now)
```

That'll do.

Here's what it looks like use it.

```
>>> t1 = WeatherReport({'city': 'NYC', 'temperature': 80})
>>> t2 = WeatherReport({'city': 'NYC', 'temperature': 81})
>>> t3 = WeatherReport({'city': 'NYC', 'temperature': 90})
>>> (t1.temperature + t2.temperature + t3.temperature) / 3
Decimal('83.66666666666666666666666667')
```

And remember that `DateTimeType` we set a default callable for?

```
>>> t1.taken_at
datetime.datetime(2013, 8, 21, 13, 6, 38, 11883)
```

2.2 Validation

Validating data is fundamentally important for many systems.

This is what it looks like when validation succeeds.

```
>>> t1.validate()
>>>
```

And this is what it looks like when validation fails.

```
>>> t1.taken_at = 'whatever'
>>> t1.validate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "schematics/models.py", line 229, in validate
    raise ModelValidationError(e.messages)
schematics.exceptions.ModelValidationError: {'taken_at': [u'Could not parse whatever.␣
↪Should be ISO8601.']}
```

2.3 Serialization

Serialization comes in two primary forms. In both cases the data is produced as a dictionary.

The `to_primitive()` function will reduce the native Python types into string safe formats. For example, the `DateTimeType` from above is stored as a Python `datetime`, but it will serialize to an ISO8601 format string.

```
>>> t1.to_primitive()
{'city': u'NYC', 'taken_at': '2013-08-21T13:04:19.074808', 'temperature': u'80'}
```

Converting to JSON is then a simple task.

```
>>> json_str = json.dumps(t1.to_primitive())
>>> json_str
'{"city": "NYC", "taken_at": "2013-08-21T13:04:19.074808", "temperature": "80"}'
```

Instantiating an instance from JSON is not too different.

```
>>> t1_prime = WeatherReport(json.loads(json_str))
>>> t1_prime.taken_at
datetime.datetime(2013, 8, 21, 13, 4, 19, 074808)
```

2.4 Persistence

In many cases, persistence can be as easy as converting the model to a dictionary and passing that into a query.

First, to get at the values we'd pass into a SQL database, we might call `to_native()`.

Let's get a fresh `WeatherReport` instance.

```
>>> wr = WeatherReport({'city': 'NYC', 'temperature': 80})
>>> wr.to_native()
{'city': u'NYC', 'taken_at': datetime.datetime(2013, 8, 27, 0, 25, 53, 185279),
↪'temperature': Decimal('80')}
```

2.4.1 With PostgreSQL

You'll want to create a table with this query:

```
CREATE TABLE weatherreports(
    city varchar,
    taken_at timestamp,
    temperature decimal
);
```

Inserting

Then, from Python, an insert statement could look like this:

```
>>> query = "INSERT INTO weatherreports (city, taken_at, temperature) VALUES (%s, %s,
↪%s);"
>>> params = (wr.city, wr.taken_at, wr.temperature)
```

Let's insert that into PostgreSQL using the `psycopg2` driver.

```
>>> import psycopg2
>>> db_conn = psycopg2.connect("host='localhost' dbname='mydb'")
>>> cursor = db_conn.cursor()
>>> cursor.execute(query, params)
>>> db_conn.commit()
```

Reading

Reading isn't much different.

```
>>> query = "SELECT city,taken_at,temperature FROM weatherreports;"
>>> cursor = db_conn.cursor()
>>> cursor.execute(query)
>>> rows = dbc.fetchall()
```

Now to translate that data into instances

```
>>> instances = list()
>>> for row in rows:
...     (city, taken_at, temperature) = row
...     instance = WeatherReport()
...     instance.city = city
...     instance.taken_at = taken_at
...     instance.temperature = temperature
...     instances.append(instance)
...
>>> instances
[<WeatherReport: WeatherReport object>]
```

Please note that the documentation is currently somewhat out of date.

CHAPTER 3

About

Schematics is a Python library to combine types into structures, validate them, and transform the shapes of your data based on simple descriptions.

The internals are similar to ORM type systems, but there is no database layer in Schematics. Instead, we believe that building a database layer is made significantly easier when Schematics handles everything but writing the query.

Further, it can be used for a range of tasks where having a database involved may not make sense.

Some common use cases:

- Design and document specific *data structures*
- *Convert structures* to and from different formats such as JSON or MsgPack
- *Validate* API inputs
- *Remove fields based on access rights* of some data's recipient
- Define message formats for communications protocols, like an RPC
- Custom *persistence layers*

CHAPTER 4

Example

This is a simple Model.

```
>>> from schematics.models import Model
>>> from schematics.types import StringType, URLType
>>> class Person(Model):
...     name = StringType(required=True)
...     website = URLType()
...
>>> person = Person({'name': u'Joe Strummer',
...                  'website': 'http://soundcloud.com/joestrummer'})
>>> person.name
u'Joe Strummer'
```

Serializing the data to JSON.

```
>>> import json
>>> json.dumps(person.to_primitive())
{"name": "Joe Strummer", "website": "http://soundcloud.com/joestrummer"}
```

Let's try validating without a name value, since it's required.

```
>>> person = Person()
>>> person.website = 'http://www.amontobin.com/'
>>> person.validate()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "schematics/models.py", line 231, in validate
    raise DataError(e.messages)
schematics.exceptions.DataError: {'name': ['This field is required.']}
```

Add the field and validation passes:

```
>>> person = Person()
>>> person.name = 'Amon Tobin'
```

(continues on next page)

(continued from previous page)

```
>>> person.website = 'http://www.amontobin.com/'
>>> person.validate()
>>>
```

CHAPTER 5

Installing

Install stable releases of Schematics with pip.

```
$ pip install schematics
```

See the *Install Guide* for more detail.

CHAPTER 6

Getting Started

New Schematics users should start with the *Quickstart Guide*. That is the fastest way to get a look at what Schematics does.

CHAPTER 7

Documentation

Schematics exists to make a few concepts easy to glue together. The types allow us to describe units of data, models let us put them together into structures with fields. We can then import data, check if it looks correct, and easily serialize the results into any format we need.

The User's Guide provides the high-level concepts, but the API documentation and the code itself provide the most accurate reference.

7.1 Types

Types are the smallest definition of structure in Schematics. They represent structure by offering functions to inspect or mutate the data in some way.

According to Schematics, a type is an instance of a way to do three things:

1. Coerce the data type into an appropriate representation in Python
2. Convert the Python representation into other formats suitable for serialization
3. Offer a precise method of validating data of many forms

These properties are implemented as `to_native`, `to_primitive`, and `validate`.

7.1.1 Coercion

A simple example is the `DateTimeType`.

```
>>> from schematics.types import DateTimeType
>>> dt_t = DateTimeType()
```

The `to_native` function transforms an ISO8601 formatted date string into a Python `datetime.datetime`.

```
>>> dt = dt_t.to_native('2013-08-31T02:21:21.486072')
>>> dt
datetime.datetime(2013, 8, 31, 2, 21, 21, 486072)
```

7.1.2 Conversion

The `to_primitive` function changes it back to a language agnostic form, in this case an ISO8601 formatted string, just like we used above.

```
>>> dt_t.to_primitive(dt)
'2013-08-31T02:21:21.486072'
```

7.1.3 Validation

Validation can be as simple as successfully calling `to_native`, but sometimes more is needed. data or behavior during a typical use, like serialization.

Let's look at the `StringType`. We'll set a `max_length` of 10.

```
>>> st = StringType(max_length=10)
>>> st.to_native('this is longer than 10')
u'this is longer than 10'
```

It converts to a string just fine. Now, let's attempt to validate it.

```
>>> st.validate('this is longer than 10')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "schematics/types/base.py", line 164, in validate
    raise ValidationError(errors)
schematics.exceptions.ValidationError: [u'String value is too long.']
```

7.1.4 Custom types

If the types provided by the schematics library don't meet all of your needs, you can also create new types. Do so by extending `schematics.types.BaseType`, and decide which based methods you need to override.

to_native

By default, this method on `schematics.types.BaseType` just returns the primitive value it was given. Override this if you want to convert it to a specific native value. For example, suppose we are implementing a type that represents the net-location portion of a URL, which consists of a hostname and optional port number:

```
>>> from schematics.types import BaseType
>>> class NetlocType(BaseType):
...     def to_native(self, value):
...         if ':' in value:
...             return tuple(value.split(':', 1))
...         return (value, None)
```

to_primitive

By default, this method on `schematics.types.BaseType` just returns the native value it was given. Override this to convert any non-primitive values to primitive data values. The following types can pass through safely:

- int
- float
- bool
- basestring
- NoneType
- lists or dicts of any of the above or containing other similarly constrained lists or dicts

To cover values that fall outside of these definitions, define a primitive conversion:

```
>>> from schematics.types import BaseType
>>> class NetlocType(BaseType):
...     def to_primitive(self, value):
...         host, port = value
...         if port:
...             return u'{0}:{1}'.format(host, port)
...         return host
```

validation

The base implementation of *validate* runs individual validators defined:

- At type class definition time, as methods named in a specific way
- At instantiation time as arguments to the type's init method.

The second type is explained by `schematics.types.BaseType`, so we'll focus on the first option.

Declared validation methods take names of the form *validate_constraint(self, value)*, where *constraint* is an arbitrary name you give to the check being performed. If the check fails, then the method should raise `schematics.exceptions.ValidationError`:

```
>>> from schematics.exceptions import ValidationError
>>> from schematics.types import BaseType
>>> class NetlocType(BaseType):
...     def validate_netloc(self, value):
...         if ':' not in value:
...             raise ValidationError('Value must be a valid net location of the form␣
↪host[:port]')
```

However, schematics types do define an organized way to define and manage coded error messages. By defining a *MESSAGES* dict, you can assign error messages to your constraint name. Then the message is available as *self.message['my_constraint']* in validation methods. Sub-classes can add messages for new codes or replace messages for existing codes. However, they will inherit messages for error codes defined by base classes.

So, to enhance the prior example:

```
>>> from schematics.exceptions import ValidationError
>>> from schematics.types import BaseType
>>> class NetlocType(BaseType):
...     MESSAGES = {
```

(continues on next page)

(continued from previous page)

```
...         'netloc': 'Value must be a valid net location of the form host[:port]'
...     }
...     def validate_netloc(self, value):
...         if ':' not in value:
...             raise ValidationError(self.messages['netloc'])
```

Parameterizing types

There may be times when you want to override *__init__* and parameterize your type. When you do so, just ensure two things:

- Don't redefine any of the initialization parameters defined for `schematics.types.BaseType`.
- After defining your specific parameters, ensure that the base parameters are given to the base init method. The simplest way to ensure this is to accept **args* and ***kwargs* and pass them through to the super init method, like so:

```
>>> from schematics.types import BaseType
>>> class NetlocType(BaseType):
...     def __init__(self, verify_location=False, *args, **kwargs):
...         super(NetlocType, self).__init__(*args, **kwargs)
...         self.verify_location = verify_location
```

7.1.5 More Information

To learn more about **Types**, visit the *Types API*

7.2 Models

Schematics models are the next form of structure above types. They are a collection of types in a class. When a *Type* is given a name inside a *Model*, it is called a *field*.

7.2.1 Simple Model

Let's say we want to build a social network for weather. At its core, we'll need a way to represent some temperature information and where that temperature was found.

```
import datetime
from schematics.models import Model
from schematics.types import StringType, DecimalType, DateTimeType

class WeatherReport(Model):
    city = StringType()
    temperature = DecimalType()
    taken_at = DateTimeType(default=datetime.datetime.now)
```

That'll do. Let's try using it.

```
>>> wr = WeatherReport({'city': 'NYC', 'temperature': 80})
>>> wr.temperature
Decimal('80.0')
```

And remember that `DateTimeType` we set a default callable for?

```
>>> wr.taken_at
datetime.datetime(2013, 8, 21, 13, 6, 38, 11883)
```

7.2.2 Model Configuration

Models offer a few configuration options. Options are attached in the form of a class.

```
class Whatever(Model):
    ...
    class Options:
        option = value
```

`namespace` is a namespace identifier that can be used with persistence layers.

```
class Whatever(Model):
    ...
    class Options:
        namespace = "whatever_bucket"
```

`roles` is a dictionary that stores whitelists and blacklists.

```
class Whatever(Model):
    ...
    class Options:
        roles = {
            'public': whitelist('some', 'fields'),
            'owner': blacklist('some', 'internal', 'stuff'),
        }
```

`serialize_when_none` can be `True` or `False`. It's behavior is explained here: *Serialize When None*.

```
class Whatever(Model):
    ...
    class Options:
        serialize_when_none = False
```

7.2.3 Model Mocking

Testing typically involves creating lots of fake (but plausible) objects. Good tests use random values so that multiple tests can run in parallel without overwriting each other. Great tests exercise many possible valid input values to make sure the code being tested can deal with various combinations.

Schematics models can help you write great tests by automatically generating mock objects. Starting with our `WeatherReport` model from earlier:

```
class WeatherReport(Model):
    city = StringType()
    temperature = DecimalType()
    taken_at = DateTimeType(default=datetime.datetime.now)
```

we can ask Schematic to generate a mock object with reasonable values:

```
>>> WeatherReport.get_mock_object().to_primitive()
{'city': u'zLmeEt7OAGOWI', 'temperature': u'8', 'taken_at': '2014-05-06T17:34:56.
↪396280'}
```

If you've set a constraint on a field that the mock can't satisfy - such as putting a `max_length` on a URL field so that it's too small to hold a randomly-generated URL - then `get_mock_object` will raise a `MockCreationError` exception:

```
from schematics.types import URLType

class OverlyStrict(Model):
    url = URLType(max_length=11, required=True)

>>> OverlyStrict.get_mock_object()
...
schematics.exceptions.MockCreationError: url: This field is too short to hold the␣
↪mock data
```

7.2.4 More Information

To learn more about **Models**, visit the *Models API*

7.3 Exporting

To export data is to go from the Schematics representation of data to some other form. It's also possible you want to adjust some things along the way, such as skipping over some fields or providing empty values for missing fields.

The general mechanism for data export is to call a function on every field in the model. The function probably converts the field's value to some other format, but you can easily modify it.

We'll use the following model for the examples:

```
from schematics.models import Model
from schematics.types import StringType, DateTimeType
from schematics.transforms import blacklist

class Movie(Model):
    name = StringType()
    director = StringType()
    release_date = DateTimeType
    personal_thoughts = StringType()
    class Options:
        roles = {'public': blacklist('personal_thoughts')}
```

7.3.1 Terminology

To *serialize* data is to convert from the way it's represented in Schematics to some other form. That might be a reduction of the `Model` into a `dict`, but it might also be more complicated.

A field can be serialized if it is an instance of `BaseType` or if a function is wrapped with the `@serializable` decorator.

A `Model` instance may be serialized with a particular *context*. A context is a `dict` passed through the model to each of its fields. A field may use values from the context to alter how it is serialized.

7.3.2 Converting Data

To export data is basically to convert from one form to another. Schematics can convert data into simple Python types or a language agnostic format. We refer to the native serialization as *to_native*, but we refer to the language agnostic format as *primitive*, since it has removed all dependencies on Python.

Native Types

The fields in a model attempt to use the best Python representation of data whenever possible. For example, the DateTimeType will use Python's `datetime.datetime` module.

You can reduce a model into the native Python types by calling `to_native`.

```
>>> trainspotting = Movie()
>>> trainspotting.name = u'Trainspotting'
>>> trainspotting.director = u'Danny Boyle'
>>> trainspotting.release_date = datetime.datetime(1996, 7, 19, 0, 0)
>>> trainspotting.personal_thoughts = 'This movie was great!'
>>> trainspotting.to_native()
{
  'name': u'Trainspotting',
  'director': u'Danny Boyle',
  'release_date': datetime.datetime(1996, 7, 19, 0, 0),
  'personal_thoughts': 'This movie was great!'
}
```

Primitive Types

To present data to clients we have the `Model.to_primitive` method. Default behavior is to output the same data you would need to reproduce the model in its current state.

```
>>> trainspotting.to_primitive()
{
  'name': u'Trainspotting',
  'director': u'Danny Boyle',
  'release_date': '1996-07-19T00:00:00.000000',
  'personal_thoughts': 'This movie was great!'
}
```

Great. We got the primitive data back. It would be easy to convert to JSON from here.

```
>>> import json
>>> json.dumps(trainspotting.to_primitive())
'{
   "name": "Trainspotting",
   "director": "Danny Boyle",
   "release_date": "1996-07-19T00:00:00.000000",
   "personal_thoughts": "This movie was great!"
 }'
```

Using Contexts

Sometimes a field needs information about its environment to know how to serialize itself. For example, the `MultilingualStringType` holds several translations of a phrase:

```
>>> class TestModel(Model):
...     mls = MultilingualStringType()
...
>>> mls_test = TestModel({'mls': {
...     'en_US': 'Hello, world!',
...     'fr_FR': 'Bonjour tout le monde!',
...     'es_MX': '¡Hola, mundo!',
... }})
```

In this case, serializing without knowing which localized string to use wouldn't make sense:

```
>>> mls_test.to_primitive()
[...]
schematics.exceptions.ConversionError: [u'No default or explicit locales were given.']
```

Neither does choosing the locale ahead of time, because the same MultilingualStringType field might be serialized several times with different locales inside the same method.

However, it could use information in a *context* to return a useful representation:

```
>>> mls_test.to_primitive(context={'locale': 'en_US'})
{'mls': 'Hello, world!'}
```

This allows us to use the same model instance several times with different contexts:

```
>>> for user, locale in [('Joe', 'en_US'), ('Sue', 'es_MX')]:
...     print '%s says %s' % (user, mls_test.to_primitive(context={'locale': locale})[
↪'mls'])
...
Joe says Hello, world!
Sue says ¡Hola, mundo!
```

7.3.3 Compound Types

Let's complicate things and observe what happens with data exporting. First, we'll define a collection which will have a list of `Movie` instances.

First, let's instantiate another movie.

```
>>> total_recall = Movie()
>>> total_recall.name = u'Total Recall'
>>> total_recall.director = u'Paul Verhoeven'
>>> total_recall.release_date = datetime.datetime(1990, 6, 1, 0, 0)
>>> total_recall.personal_thoughts = 'Old classic.  Still love it.'
```

Now, let's define a collection, which has a list of movies in it.

```
from schematics.types.compound import ListType, ModelType

class Collection(Model):
    name = StringType()
    movies = ListType(ModelType(Movie))
    notes = StringType()
    class Options:
        roles = {'public': blacklist('notes')}
```

Let's instantiate a collection.

```
>>> favorites = Collection()
>>> favorites.name = 'My favorites'
>>> favorites.notes = 'These are some of my favorite movies'
>>> favorites.movies = [trainspotting, total_recall]
```

Here is what happens when we call `to_primitive()` on it.

```
>>> favorites.to_primitive()
{
    'notes': 'These are some of my favorite movies',
    'name': 'My favorites',
    'movies': [{
        'name': u'Trainspotting',
        'director': u'Danny Boyle',
        'personal_thoughts': 'This movie was great!',
        'release_date': '1996-07-19T00:00:00.000000'
    }, {
        'name': u'Total Recall',
        'director': u'Paul Verhoeven',
        'personal_thoughts': 'Old classic.  Still love it.',
        'release_date': '1990-06-01T00:00:00.000000'
    }]
}
```

7.3.4 Customizing Output

Schematics offers many ways to customize the behavior of serialization:

Roles

Roles offer a way to specify whether or not a field should be skipped during export. There are many reasons this might be desirable, such as access permissions or to not serialize more data than absolutely necessary.

Roles are implemented as either white lists or black lists where the members of the list are field names.

```
>>> r = blacklist('private_field', 'another_private_field')
```

Imagine we are sending our movie instance to a random person on the Internet. We probably don't want to share our personal thoughts. Recall earlier that we added a role called `public` and gave it a blacklist with `personal_thoughts` listed.

```
class Movie(Model):
    personal_thoughts = StringType()
    ...
    class Options:
        roles = {'public': blacklist('personal_thoughts')}
```

This is what it looks like to use the role, which should simply remove `personal_thoughts` from the export.

```
>>> movie.to_primitive(role='public')
{
    'name': u'Trainspotting',
    'director': u'Danny Boyle',
    'release_date': '1996-07-19T00:00:00.000000'
}
```

This works for compound types too, such as the list of movies in our `Collection` model above.

```
class Collection(Model):
    notes = StringType()
    ...
    class Options:
        roles = {'public': blacklist('notes')}
```

We expect the `personal_thoughts` field to removed from the movie data and we also expect the `notes` field to be removed from the collection data.

```
>>> favorites.to_primitive(role='public')
{
    'name': 'My favorites',
    'movies': [{
        'name': u'Trainspotting',
        'director': u'Danny Boyle',
        'release_date': '1996-07-19T00:00:00.000000'
    }, {
        'name': u'Total Recall',
        'director': u'Paul Verhoeven',
        'release_date': '1990-06-01T00:00:00.000000'
    }]
}
```

If no role is specified, the default behavior is to export all fields. This behavior can be overridden by specifying a `default` role. Renaming the `public` role to `default` in the example above yields equivalent results without having to specify `role` in the export function.

```
>>> favorites.to_primitive()
{
    'name': 'My favorites',
    'movies': [{
        'name': u'Trainspotting',
        'director': u'Danny Boyle',
        'release_date': '1996-07-19T00:00:00.000000'
    }, {
        'name': u'Total Recall',
        'director': u'Paul Verhoeven',
        'release_date': '1990-06-01T00:00:00.000000'
    }]
}
```

Serializable

Earlier we mentioned a `@serializable` decorator. You can write a function that will produce a value used during serialization with a field name matching the function name.

That looks like this:

```
...
from schematics.types.serializable import serializable

class Song(Model):
    name = StringType()
    artist = StringType()
```

(continues on next page)

(continued from previous page)

```
    url = URLType()

    @serializable
    def id(self):
        return u'%s/%s' % (self.artist, self.name)
```

This is what it looks like to use it.

```
>>> song = Song()
>>> song.artist = 'Fiona Apple'
>>> song.name = 'Werewolf'
>>> song.url = 'http://www.youtube.com/watch?v=67KGSJVkix0'
>>> song.id
'Fiona Apple/Werewolf'
```

Or here:

```
>>> song.to_native()
{
    'id': u'Fiona Apple/Werewolf',
    'artist': u'Fiona Apple'
    'name': u'Werewolf',
    'url': u'http://www.youtube.com/watch?v=67KGSJVkix0',
}
```

Serialized Name

There are times when you have one name for a field in one place and another name for it somewhere else. Schematics tries to help you by letting you customize the field names used during serialization.

That looks like this:

```
class Person(Model):
    name = StringType(serialized_name='person_name')
```

Notice the effect it has on serialization.

```
>>> p = Person()
>>> p.name = 'Ben Weinman'
>>> p.to_native()
{'person_name': u'Ben Weinman'}
```

Serialize When None

If a value is not required and doesn't have a value, it will serialize with a None value by default. This can be disabled.

```
>>> song = Song()
>>> song.to_native()
{'url': None, 'name': None, 'artist': None}
```

You can disable at the field level like this:

```
class Song(Model):
    name = StringType(serialize_when_none=False)
    artist = StringType()
```

And this produces the following:

```
>>> s = Song()
>>> s.to_native()
{'artist': None}
```

Or you can disable it at the class level:

```
class Song(Model):
    name = StringType()
    artist = StringType()
    class Options:
        serialize_when_none=False
```

Using it:

```
>>> s = Song()
>>> s.to_native()
>>>
```

7.3.5 More Information

To learn more about **Exporting**, visit the *Transforms API*

7.4 Importing

The general mechanism for data import is to call a function on every field in the data and coerce it into the most appropriate representation in Python. A date string, for example, would be converted to a `datetime.datetime`.

Perhaps we're writing a web API that receives song data. Let's model the song.

```
class Song(Model):
    name = StringType()
    artist = StringType()
    url = URLType()
```

This is what successful validation of the data looks like.

```
>>> song_json = '{"url": "http://www.youtube.com/watch?v=67KGSJVkix0", "name":
↪"Werewolf", "artist": "Fiona Apple"}'
>>> fiona_song = Song(json.loads(song_json))
>>> fiona_song.url
u'http://www.youtube.com/watch?v=67KGSJVkix0'
```

7.4.1 Compound Types

We could define a simple collection of songs like this:

```
class Collection(Model):
    songs = ListType(ModelType(Song))
```

Some JSON data for this type of a model might look like this:

```
>>> songs_json = '{"songs": [{"url": "https://www.youtube.com/watch?v=UeBFEanVsp4",
↪"name": "When I Lost My Bet", "artist": "Dillinger Escape Plan"}, {"url": "http://
↪www.youtube.com/watch?v=67KGSJVkix0", "name": "Werewolf", "artist": "Fiona Apple"}]}
↪'
```

The collection has a list of models for songs, so when we import that list, that data should be converted to model instances.

```
>>> song_collection = Collection(json.loads(songs_json))
>>> song_collection.songs[0]
<Song: Song object>
>>> song_collection.songs[0].artist
u'Dillinger Escape Plan'
```

7.4.2 More Information

To learn more about **Importing**, visit the *Transforms API*

7.5 Validation

To validate data in Schematics is to have both a data model and some input data. The data model describes what valid data looks like in different forms.

Here's a quick glance and some of the ways you can tweak validation.

```
>>> from schematics.models import Model
>>> from schematics.types import StringType
>>> class Person(Model):
...     name = StringType()
...     bio = StringType(required=True)
...
>>> p = Person()
>>> p.name = 'Fiona Apple'
>>> p.validate()
Traceback (most recent call last):
...
ModelValidationError: {'bio': [u'This field is required.']}
```

7.5.1 Validation Errors

Validation failures throw an exception called `ValidationError`. A description of what failed is stored in `messages`, which is a dictionary keyed by the field name with a list of reasons the field failed.

```
>>> from schematics.exceptions import ValidationError
>>> try:
...     p.validate()
... except ValidationError, e:
```

(continues on next page)

(continued from previous page)

```
...     print e.messages
{'bio': [u'This field is required.']}
```

7.5.2 Extending Validation

Validation for both types and models can be extended. Whatever validation system you require is probably expressable via Schematics.

Type-level Validation

Here is a function that checks if a string is uppercase and throws a `ValidationError` if it is not.

```
>>> from schematics.exceptions import ValidationError
>>> def is_uppercase(value):
...     if value.upper() != value:
...         raise ValidationError(u'Please speak up!')
...     return value
...
```

And we can attach it to our StringType like this:

```
>>> class Person(Model):
...     name = StringType(validators=[is_uppercase])
...
```

Using it is built into validation.

```
>>> me = Person({'name': u'Jökull'})
>>> me.validate()
Traceback (most recent call last):
...
ModelValidationError: {'name': [u'Please speak up!']}
```

It is also possible to define new types with custom validation by subclassing a type, like `BaseType`, and implementing instance methods that start with `validate_`.

```
>>> from schematics.exceptions import ValidationError
>>> class UppercaseType(StringType):
...     def validate_uppercase(self, value):
...         if value.upper() != value:
...             raise ValidationError("Value must be uppercase!")
...
```

Just like before, using it is now built in.

```
>>> class Person(Model):
...     name = UppercaseType()
...
>>> me = Person({'name': u'Jökull'})
>>> me.validate()
Traceback (most recent call last):
...
ModelValidationError: {'name': ['Value must be uppercase!']}
```

Model-level Validation

What about field validation based on other model data? The order in which fields are declared is preserved inside the model. So if the validity of a field depends on another field's value, just make sure to declare it below its dependencies:

```
>>> from schematics.models import Model
>>> from schematics.types import StringType, BooleanType
>>> from schematics.exceptions import ValidationError
>>>
>>> class Signup(Model):
...     name = StringType()
...     call_me = BooleanType(default=False)
...     def validate_call_me(self, data, value):
...         if data['name'] == u'Brad' and data['call_me'] is True:
...             raise ValidationError(u'He prefers email.')
...         return value
...
>>> Signup({'name': u'Brad'}).validate()
>>> Signup({'name': u'Brad', 'call_me': True}).validate()
Traceback (most recent call last):
...
ModelValidationError: {'call_me': [u'He prefers email.']}
```

7.5.3 More Information

To learn more about **Validation**, visit the *Validation API*

7.6 Extending

For most non trivial cases, the base types may not be enough. Schematics is designed to be flexible to allow for extending data types in order to accomodate custom logic.

7.6.1 Simple Example

A simple example is allowing for value transformations.

Say that there is a model that requires email validation. Since emails are case insenstive, it might be helpful to convert the input email to lower case before continuing to validate.

This can be achieved by Extending the Email class

```
>>> from schematics.types import EmailType
>>> class LowerCaseEmailType(EmailType):
...
...     # override convert method
...     def convert(self, value, context=None):
...        value = super().convert(value, context)
...        return value.lower() # value will be converted to lowercase
```

Our `LowerCaseEmailType` can now be used as an ordinary field.

```
>>> from schematics.models import Model
>>> from schematics.types import StringType
>>> class Person(Model):
...     name = StringType()
...     bio = StringType(required=True)
...     email = LowerCaseEmailType(required=True)
...
>>> p = Person()
>>> p.name = 'Mutoid Man'
>>> p.email = 'MutoidMan@Example.com' # technically correct email,but should be
↪'cleaned'
>>> p.validate()
>>> p.to_native()
>>> {'bio': 'Mutoid Man',
>>> 'email': 'mutoidman@example.com', # the email was converted to lowercase
>>> 'name': 'Mutoid Man'}
```

7.6.2 Taking it a step further

It is also possible that you may have several different kinds of cleaning required. In such cases, it may not be ideal to subclass a type every time (like the previous example).

We can use the same logic from above and define a `Type` that can apply a set of arbitrary functions.

```
>>> class CleanedStringType(StringType):
...     converters = []
...
...     def __init__(self, **kwargs):
...         """
...         This takes in all the inputs as String Type, but takes in an extra
...         input called converters.
...
...         Converters must be a list of functions, and each of those functions
...         must take in exactly 1 value , and return the transformed input
...         """
...         if 'converters' in kwargs:
...             self.converters = kwargs['converters']
...         del kwargs['converters']
...         super().__init__(**kwargs)
...
...     def convert(self, value, context=None):
...         value = super().convert(value, context)
...         for func in self.converters:
...             value = func(value)
...         return value # will have a value after going through all the conversions␣
↪in order
```

Now that we have defined our new Type, we can use it.

```
>>> from schematics.models import Model
>>> from schematics.types import StringType
>>> class Person(Model):
...     name = StringType()
...     bio = CleanedStringType(required=True,
...                             converters = [lambda x: x.upper(),
...                             lambda x: x.split(" ")[0]]) # convert to uppercase,␣
↪then split on " " and just take the first of the split
```

(continues on next page)

(continued from previous page)

```
...     email = CleanedStringType(required=True, converts = [lambda x:x.lower()]) #␣
↪same functionality as LowerCaseEmailType
...
>>> p = Person()
>>> p.name = 'Mutoid Man'
>>> p.bio = 'good man'
>>> p.email = 'MutoidMan@Example.com' # technically correct email,but should be
↪'cleaned'
>>> p.validate()
>>> p.to_native()
>>> {'bio': 'GOOD', # was converted as we specified
>>> 'email': 'mutoidman@example.com', # was converted to lowercase
>>> 'name': 'Mutoid Man'}
```

7.7 Models

7.7.1 Usage

To learn more about how **Models** are used, visit *Using Models*

7.8 Validation

7.8.1 Usage

To learn more about how **Validation** is used, visit *Using Validation*

7.9 Transforms

7.9.1 Usage

To learn more about how **Transforms** are used, visit *Using Importing* and *Using Exporting*

7.10 Types

class BaseType(*required=False*, *default=Undefined*, *serialized_name=None*, *choices=None*, *validators=None*, *deserialize_from=None*, *export_level=None*, *serialize_when_none=None*, *messages=None*, *metadata=None*)

A base class for Types in a Schematics model. Instances of this class may be added to subclasses of `Model` to define a model schema.

Validators that need to access variables on the instance can be defined be implementing methods whose names start with `validate_` and accept one parameter (in addition to `self`)

Parameters

- **required** – Invalidate field when value is None or is not supplied. Default: False.

- **default** – When no data is provided default to this value. May be a callable. Default: None.
- **serialized_name** – The name of this field defaults to the class attribute used in the model. However if the field has another name in foreign data set this argument. Serialized data will use this value for the key name too.
- **deserialize_from** – A name or list of named fields for which foreign data sets are searched to provide a value for the given field. This only effects inbound data.
- **choices** – A list of valid choices. This is the last step of the validator chain.
- **validators** – A list of callables. Each callable receives the value after it has been converted into a rich python type. Default: []
- **serialize_when_none** – Dictates if the field should appear in the serialized data even if the value is None. Default: None.
- **messages** – Override the error messages with a dict. You can also do this by subclassing the Type and defining a *MESSAGES* dict attribute on the class. A metaclass will merge all the *MESSAGES* and override the resulting dict with instance level *messages* and assign to *self.messages*.
- **metadata** – Dictionary for storing custom metadata associated with the field. To encourage compatibility with external tools, we suggest these keys for common metadata: - *label* : Brief human-readable label - *description* : Explanation of the purpose of the field. Used for

  help, tooltips, documentation, etc.

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

validate(*value*, *context=None*)
Validate the field and return a converted value or raise a `ValidationError` with a list of errors raised by the validation chain. Stop the validation process from continuing through the validators by raising `StopValidationError` instead of `ValidationError`.

class UUIDType(***kwargs*)
A field that stores a valid UUID value.

native_type
alias of `uuid.UUID`

primitive_type
alias of `builtins.str`

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

class StringType(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A Unicode string field.

native_type
alias of `builtins.str`

primitive_type
alias of `builtins.str`

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

class MultilingualStringType(*regex=None*, *max_length=None*, *min_length=None*, *default_locale=None*, *locale_regex='^[a-z]{2}(:?_[A-Z]{2})?$'*, ***kwargs*)
A multilanguage string field, stored as a dict with {'locale': 'localized_value'}.

Minimum and maximum lengths apply to each of the localized values.

At least one of `default_locale` or `context.app_data['locale']` must be defined when calling `.to_primitive`.

native_type
alias of `builtins.str`

primitive_type
alias of `builtins.str`

to_native(*value*, *context=None*)
Make sure a MultilingualStringType value is a dict or None.

to_primitive(*value*, *context=None*)
Use a combination of `default_locale` and `context.app_data['locale']` to return the best localized string.

class NumberType(*min_value=None*, *max_value=None*, *strict=False*, ***kwargs*)
A generic number field. Converts to and validates against *number_type* parameter.

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

class IntType(***kwargs*)
A field that validates input as an Integer

native_type
alias of `builtins.int`

primitive_type
alias of `builtins.int`

LongType
alias of *schematics.types.base.IntType*

class FloatType(***kwargs*)
A field that validates input as a Float

native_type
alias of `builtins.float`

primitive_type
alias of `builtins.float`

class DecimalType(*min_value=None*, *max_value=None*, *strict=False*, ***kwargs*)
A fixed-point decimal number field.

native_type
alias of `decimal.Decimal`

primitive_type
alias of `builtins.str`

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

class HashType(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

class MD5Type(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that validates input as resembling an MD5 hash.

class SHA1Type(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that validates input as resembling an SHA1 hash.

class BooleanType(***kwargs*)
A boolean field type. In addition to `True` and `False`, coerces these values:

- For `True`: "True", "true", "1"
- For `False`: "False", "false", "0"

native_type
alias of `builtins.bool`

primitive_type
alias of `builtins.bool`

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

class GeoPointType(*required=False*, *default=Undefined*, *serialized_name=None*, *choices=None*, *validators=None*, *deserialize_from=None*, *export_level=None*, *serialize_when_none=None*, *messages=None*, *metadata=None*)
A list storing a latitude and longitude.

native_type
alias of `builtins.list`

primitive_type
alias of `builtins.list`

to_native(*value*, *context=None*)
Make sure that a geo-value is of type (x, y)

class DateType(*formats=None*, ***kwargs*)
Defaults to converting to and from ISO8601 date values.

native_type
alias of `datetime.date`

primitive_type
alias of `builtins.str`

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

class DateTimeType(*formats=None*, *serialized_format=None*, *parser=None*, *tzd='allow'*, *convert_tz=False*, *drop_tzinfo=False*, ***kwargs*)
A field that holds a combined date and time value.

The built-in parser accepts input values conforming to the ISO 8601 format `<YYYY>-<MM>-<DD>T<hh>:<mm>[:<ss.ssssss>][<z>]`. A space may be substituted for the delimiter `T`. The time zone designator `<z>` may be either `Z` or ±`<hh>[:][<mm>]`.

Values are stored as standard `datetime.datetime` instances with the time zone offset in the `tzinfo` component if available. Raw values that do not specify a time zone will be converted to naive `datetime` objects unless `tzd='utc'` is in effect.

Unix timestamps are also valid input values and will be converted to UTC datetimes.

Parameters

- **formats** – (Optional) A value or iterable of values suitable as `datetime.datetime.strptime` format strings, for example `('%Y-%m-%dT%H:%M:%S', '%Y-%m-%dT%H:%M:%S.%f')`. If the parameter is present, `strptime()` will be used for parsing instead of the built-in parser.
- **serialized_format** – The output format suitable for Python `strftime`. Default: `'%Y-%m-%dT%H:%M:%S.%f%z'`
- **parser** – (Optional) An external function to use for parsing instead of the built-in parser. It should return a `datetime.datetime` instance.
- **tzd** – Sets the time zone policy. Default: `'allow'`

`'require'`	Values must specify a time zone.
`'allow'`	Values both with and without a time zone designator are allowed.
`'utc'`	Like `allow`, but values with no time zone information are assumed to be in UTC.
`'reject'`	Values must not specify a time zone. This also prohibits timestamps.

- **convert_tz** – Indicates whether values with a time zone designator should be automatically converted to UTC. Default: `False`
  - `True`: Convert the datetime to UTC based on its time zone offset.
  - `False`: Don't convert. Keep the original time and offset intact.
- **drop_tzinfo** – Can be set to automatically remove the `tzinfo` objects. This option should generally be used in conjunction with the `convert_tz` option unless you only care about local wall clock times. Default: `False`
  - `True`: Discard the `tzinfo` components and make naive `datetime` objects instead.
  - `False`: Preserve the `tzinfo` components if present.

class fixed_timezone

dst(*dt*)
datetime -> DST offset in minutes east of UTC.

fromutc(*dt*)
datetime in UTC -> datetime in local time.

tzname(*dt*)
datetime -> string name of time zone.

utcoffset(*dt*)
datetime -> timedelta showing offset from UTC, negative values indicating West of UTC

native_type
alias of `datetime.datetime`

class offset_timezone(*hours=0*, *minutes=0*)

primitive_type
alias of `builtins.str`

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

class utc_timezone

class UTCDateTimeType(*formats=None*, *parser=None*, *tzd='utc'*, *convert_tz=True*, *drop_tzinfo=True*, ***kwargs*)
A variant of `DateTimeType` that normalizes everything to UTC and stores values as naive `datetime` instances. By default sets `tzd='utc'`, `convert_tz=True`, and `drop_tzinfo=True`. The standard export format always includes the UTC time zone designator `"Z"`.

class TimestampType(*formats=None*, *parser=None*, *drop_tzinfo=False*, ***kwargs*)
A variant of `DateTimeType` that exports itself as a Unix timestamp instead of an ISO 8601 string. Always sets `tzd='require'` and `convert_tz=True`.

primitive_type
alias of `builtins.float`

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

class TimedeltaType(*precision='seconds'*, ***kwargs*)
Converts Python Timedelta objects into the corresponding value in seconds.

native_type
alias of `datetime.timedelta`

primitive_type
alias of `builtins.float`

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

class CompoundType(***kwargs*)

to_native(*value*, *context=None*)
Convert untrusted data to a richer Python construct.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

MultiType
alias of *schematics.types.compound.CompoundType*

class ModelType(*model_spec*, ***kwargs*)
A field that can hold an instance of the specified model.

primitive_type
alias of `builtins.dict`

class ListType(*field*, *min_size=None*, *max_size=None*, ***kwargs*)
A field for storing a list of items, all of which must conform to the type specified by the `field` parameter.

Use it like this:

```
...
categories = ListType(StringType)
```

native_type
alias of `builtins.list`

primitive_type
alias of `builtins.list`

class DictType(*field*, *coerce_key=None*, ***kwargs*)
A field for storing a mapping of items, the values of which must conform to the type specified by the `field` parameter.

Use it like this:

```
...
categories = DictType(StringType)
```

native_type
alias of `builtins.dict`

primitive_type
alias of `builtins.dict`

class PolyModelType(*model_spec*, ***kwargs*)
A field that accepts an instance of any of the specified models.

find_model(*data*)
Finds the intended type by consulting potential classes or *claim_function*.

primitive_type
alias of `builtins.dict`

class IPAddressType(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that stores a valid IPv4 or IPv6 address.

class IPv4Type(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that stores a valid IPv4 address.

class IPv6Type(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that stores a valid IPv6 address.

class MACAddressType(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that stores a valid MAC address.

to_primitive(*value*, *context=None*)
Convert internal data to a value safe to serialize.

class URLType(*fqdn=True*, *verify_exists=False*, ***kwargs*)
A field that validates the input as a URL.

Parameters

- **fqdn** – if `True` the validation function will ensure hostname in URL is a Fully Qualified Domain Name.
- **verify_exists** – if `True` the validation function will make sure the URL is accessible (server responds with HTTP 2xx).

class EmailType(*regex=None*, *max_length=None*, *min_length=None*, ***kwargs*)
A field that validates input as an E-Mail-Address.

7.10.1 Usage

To learn more about how **Types** are used, visit *Using Types*

7.11 Contrib

CHAPTER 8

Development

We welcome ideas and code. We ask that you follow some of our guidelines though.

See the *Developer's Guide* for more information.

8.1 Developer's Guide

Schematics development is currently led by Kalle Tuure, but this project is very much a sum of the work done by a community.

8.1.1 List of Contributors

```
$ cd schematics
$ git shortlog -sne
```

Schematics has a few design choices that are both explicit and implicit. We care about these decisions and have probably debated them on the mailing list. We ask that you honor those and make them known in this document.

8.1.2 Get the code

Please see the *Installing from GitHub* section of the *Install Guide* page for details on how to obtain the Schematics source code.

8.1.3 Commit Message Guidelines

We use a standard format for the commit messages that allows more readable browsing of the project history, and specially to help in generating the change log.

Commit Message Format

Each commit message consists of a **header**, a **body** and a **footer**. The header has a special format that includes a **type**, a **scope** and a **subject**:

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

The **header** is mandatory and the **scope** of the header is optional.

Any line of the commit message cannot be longer 100 characters! This allows the message to be easier to read on GitHub as well as in various git tools.

The footer should contain a closing reference to an issue if any.

Allowed **type** values:

- build: Changes that affect the build system or external dependencies
- ci: Changes to CI configuration files and scripts
- docs: Documentation only changes
- feat: A new feature
- fix: A bug fix
- perf: A code change that improves performance
- refactor: A code change that neither fixes a bug nor adds a feature (eg. renaming a variable)
- style: Changes that do not affect the meaning of the code (formatting, missing semi colons, etc)
- test: Adding missing tests or correcting existing tests

Example **scope** values:

The scope should be the name of the module affected.

- types
- models
- serializable
- schema
- transforms
- etc.

Subject

The subject contains a succinct description of the change:

- use the imperative, present tense: “change” not “changed” nor “changes”
- don’t capitalize the first letter
- no dot (.) at the end

Body

Just as in the subject, use the imperative, present tense: "change" not "changed" nor "changes". The body should include the motivation for the change and contrast this with previous behavior.

Footer

The footer should contain any information about Breaking Changes and is also the place to reference GitHub issues that this commit Closes.

Example:

```
Closes #123, #234, #456
```

Breaking Changes should start with the word BREAKING CHANGE: with a space or two newlines. The rest of the commit message is then used for this.

Example:

```
BREAKING CHANGE: convert and validate functions for Types now need to
  accept an optional context parameter and pass it to any calls to super.

  To migrate the code follow the example below:

  Before:

  def convert(self, value):
      return super().convert(value)

  After:

  def convert(self, value, context=None):
      return super().convert(value, context)

  Refer to the documentation regarding using the context data.
```

8.1.4 Tests

Using pytest:

```
$ py.test
```

Naming

Schematics has the tradition of naming examples after music bands and artists so you can use your favorite ones when creating examples in the docs and for test fixtures.

If you are not feeling particularly creative, you can use one of @jmsdnns selections below:

- Mutoid Man
- Pulled Apart By Horses
- Fiona Apple
- Julia Holter

- Lifetime
- Nujabes
- Radiohead
- Stars Of The Lid

8.1.5 Writing Documentation

Documentation is essential to helping other people understand, learn, and use Schematics. We would appreciate any help you can offer in contributing documentation to our project.

Schematics uses the .rst (reStructuredText) format for all of our documentation. You can read more about .rst on the reStructuredText Primer page.

8.1.6 Installing Documentation

Just as you verify your code changes in your local environment before committing, you should also verify that your documentation builds and displays properly on your local environment.

First, install Sphinx:

```
$ pip install sphinx
```

Next, run the Docs builder:

```
$ cd docs
$ make html
```

The docs will be placed in the `./_build` folder and you can view them from any standard web browser. (Note: the `./_build` folder is included in the `.gitignore` file to prevent the compiled docs from being included with your commits).

Each time you make changes and want to see them, re-run the Docs builder and refresh the page.

Once the documentation is up to your standards, go ahead and commit it. As with code changes, please be descriptive in your documentation commit messages as it will help others understand the purpose of your adjustment.

8.1.7 Release Guide

To prepare a new release, follow this procedure:

- Update version number in `schematics/__init__.py`
- Add signed tag with version number in git, ex: `git tag -s v1.1.3 -m "Release v1.1.3"`
- Create distribution archives `python setup.py sdist bdist_wheel`
- Sign the generated archives:

:: gpg –detach-sign -u GPGKEYID -a dist/schematics-1.1.3-py2.py3-none-any.whl gpg –detach-sign -u GPGKEYID -a dist/schematics-1.1.3.tar.gz

- Upload to PyPI `twine upload dist/schematics-1.1.3*`

8.2 Community

Schematics was created in Brooklyn, NY by James Dennis. Since then, the code has been worked on by folks from around the world. If you have ideas, we encourage you to share them!

Special thanks to Hacker School, Plain Vanilla, Quantopian, Apple, Johns Hopkins University, and everyone who has contributed to Schematics.

8.2.1 Bugs & Features

We track bugs, feature requests, and documentation requests with Github Issues.

8.2.2 Mailing list

We discuss the future of Schematics and upcoming changes in detail on schematics-dev.

If you've read the documentation and still haven't found the answer you're looking for, you should reach out to us here too.

8.2.3 Contributing

If you're interested in contributing code or documentation to Schematics, please visit the *Developer's Guide* for instructions.

CHAPTER 9

Testing & Coverage

Run `coverage` and check the missing statements.

```
$ coverage run --source schematics -m py.test && coverage report
```

Python Module Index

S

Index

S

T

U

V