Typerighter Documentation

Jms Dnns

Jun 09, 2021

GETTING STARTED

Data Types for Cynical Humans.

Welcome to TypeRighter, a project that wants to make it easier to deal with data that is messier than we'd prefer.

CHAPTER

ONE

OVERVIEW

A user can create *types* or *records* and use those to validate data, convert it to and from native Python types, or adjust its structure somehow.

TypeRighter is a fundamentally a system for structuring data such that it can be used inside Python's metaprogramming API.

If you are new to TypeRighter, the quickstart guide should be your first stop.

Happy hacking.

1.1 Installation

Tagged releases are available from PyPI:

```
$ pip install typerighter
```

1.1.1 Python Version

TypeRighter is strictly Python 3 and up.

1.1.2 Optional Packages

The optional packages are for running unittests or generation documentation and are explained in the *Environment* doc.

1.2 Quickstart

We start with *types*. A lot has been said about this concept and folks sometimes have different mental models for what one is.

This library defines it as something that can audit some data for coherence. We can take this further too, for some helpful tooling, but we start with this simple foundation.

For this library, a *type* is essentially something with a validation function.

From there, we create structures of data by using associating names with types in a *record*.

We then use some of the builtin functions to convert, filter, and repackage various messes of data that may or may not resemble the messes typical of working with data.

1.2.1 Types

We'll start by creating a type for short strings.

```
>>> short = types.StringType(max_length=12)
```

A type *instance* can then be used for validating data.

```
>>> short.validate('short enough')
```

Exceptions are thrown for validation errors.

```
>>> short.validate('not short enough')
...
typerighter.exceptions.ValidationException: Value length above max: short enoughhhh > 12
```

Conversion

They can also convert between Python native types and something safely language agnostic.

```
>>> dt = types.DateTimeType()
>>> dt.to_native('2021-05-28T23:39:30.989377')
datetime.datetime(2021, 5, 28, 23, 39, 30, 989377)
```

To native and back again.

```
>>> native_datetime = dt.to_native('2021-05-28T23:39:30.989377')
>>> dt.to_primitive(native_datetime)
'2021-05-28T23:39:30.989377'
```

1.2.2 Records

A Record is a structure consisting of fields, or *named type instances*.

```
>>> class Artist(types.Record):
...     name = types.StringType(required=True)
...     website = types.URLType()
...     created_at = types.DateTimeType()
...
>>> artist_type = Artist()
```

A Record is a Type, so it doesnt store data, but knows how to validate it.

```
>>> artist_data = {
...     'name': u'American Food',
...     'website': 'http://soundcloud.com/americanfood',
...     'created_at': '2021-05-28T23:39:30.989377'
... }
>>> artist_type.validate(artist_data)
```

Conversion

And it knows how to convert back and forth between native Python types and values suitable for serialization and sharing with non-Python languages.

```
>>> artist_type.to_native(artist_data)
{'name': 'American Food', 'website': 'http://soundcloud.com/americanfood', 'created_at':␣
↪datetime.datetime(2021, 5, 28, 23, 39, 30, 989377)}
>>> artist_type.to_primitive(artist_type.to_native(artist_data))
{'name': 'American Food', 'website': 'http://soundcloud.com/americanfood', 'created_at':
↪'2021-05-28T23:39:30.989377'}
```

Filtering

Pushing data through a record with a filter list

```
>>> artist_type.to_native(data, filter=['name', 'website'])
{
    'name': 'American Food',
    'website': 'https://soundcloud.com/americanfood'
}
```

Nested Records

Let's consider a record, that has a list of records, which each contain a list of records, eg. something messy.

We'll start with a record for a song.

```
class Song(types.Record):
    name = types.StringType(required=True)
    created_at = types.DateTimeType()
    file = types.UnixPathType()
    lyrics = types.StringType(max_length=255)
```

An album as a list of songs.

```
class Album(types.Record):
    name = types.StringType(required=True)
    created_at = types.DateTimeType()
    songs = types.ListType(Song())
```

And an artist has a list of albums.

```
class Artist(types.Record):
    name = types.StringType(required=True)
    created_at = types.DateTimeType()
    website = types.URLType()
    albums = types.ListType(Album())
```

Make an artist type instance.

```
>>> artist_type = Artist()
```

Structure some example data and validate it.

```
>>> artist_data = {
...     'name': 'American Food',
...     'created_at': '2021-05-29T00:00:01.001337',
...     'albums': [{
...         'name': 'Internet On The TV',
...         'created_at': '2021-05-29T00:00:01.001337',
...         'songs': [{
...             'name': 'Cane Spiders (mispoke)',
...             'created_at': '2021-05-29T00:00:00.001337',
...             'lyrics': 'Oh my gawd! It\'s that red dot! Gonna catch that...'
...         }, {
...             'name': 'My Take On Take On Me',
...             'created_at': '2021-05-30T00:00:00.001337',
...             'lyrics': 'I know. I know. I talk in numbers...'
...         }]
...     }]
... }
>>> artist_type.validate(artist_data)
```

Dot syntax is used for listing fields in nested records.

```
>>> fields = ['name', 'albums.songs.name', 'albums.songs.created_at']
>>> some_type.to_primitive(data, fields=fields)
{
    'name': 'American Food',
    'albums': [
        'songs': [{
            'name': 'Cane Spiders (mispoke)',
            'created_at': '2021-05-29T00:00:00.001337',
        }, {
            'name': 'My Take On Take On Me',
            'created_at': '2021-05-30T00:00:00.001337',
        }]
    ]
}
```

1.2.3 Views

A View is a mutable, configurable structure that stores Record data. Views behave the way classes usually behave in Python, letting Types focus on the definition and configuration of data structures.

```
>>> artist = artist_type.to_view(artist_data)
```

Working with a view looks about the same as if it were any Python class.

```
>>> artist.name
'American Food'
>>> artist.website
'https://soundcloud.com/americanfood/my-take-on-take-on-me'
>>> artist.created_at
'2021-05-29T00:00:01.001337'
```

It also knows how to validate data, but assumes it validates itself.

```
>>> artist.validate()
```

It also knows how to serialize the data it stores.

```
>>> artist.to_native()
{
    'name': 'American Food',
    'created_at': datetime.datetime(2021, 5, 29, 0, 0, 1, 1337),
    'albums': [{
        'name': 'Internet On The TV',
        'created_at': datetime.datetime(2021, 5, 29, 0, 0, 1, 1337),
        'songs': [{
            'name': 'Cane Spiders (mispoke)',
            'created_at': datetime.datetime(2021, 5, 29, 0, 0, 0, 1337),
            'lyrics': "Oh my gawd! It's that red dot! Gonna catch that..."
        }, {
            'name': 'My Take On Take On Me',
            'created_at': datetime.datetime(2021, 5, 30, 0, 0, 0, 1337),
            'lyrics': 'I know. I know. I talk in numbers...'
        }]
    }]
}
```

Filtering works with views too.

```
>>> artist.to_native(fields=['name', 'created_at'])
{
    'name': 'American Food',
    'created_at': datetime.datetime(2021, 5, 29, 0, 0, 1, 1337)
}
```

1.3 Types Guide

The atomic unit of TypeRighter is the `Type`.

A *Type* is an object that can validate data and convert it from the basic types used across the Internet, which we call "primitives", and convert it to the usual native types we'd use in Python.

It uses a metaclass to process *Type* implementations. This metaclass looks for any functions that start with *validate_* and it puts them in a list of functions that get called whenever validation occurs. This makes creating new types easy and keeps the complexity of the framework driving it out of view, much like the way a car's engine hides behind a steering wheel and pedals.

The classes implemented here are built around the metaprogramming and the way it accumulates a list of validation functions, including across subclasses.

1.3.1 Primitives

Primitive Types represent the types that are found all over the Internet, like in JSON, and implements the basics:

- *bool*
- *str*
- *int*
- *float*.

Here is a boolean validator.

```
>>> true_or_false = types.BooleanType()
>>> true_or_false.validate(True)
True
>>> true_or_false.validate(1)
True
>>> true_or_false.validate("")
False
```

Here is an integer validator.

```
>>> int_type = types.IntegerType()
>>> int_type.validate(123)
>>> int_type.validate("foo")
...
typerighter.exceptions.ValidationException: Value doesnt match type format foo
```

class typerighter.types.primitives.**BooleanType**(**a*, ***kw*)
This validator implements booleans as falsy values. This is done by passing the value directly into Python's *bool* and using Python's native behavior.

NATIVE
alias of `bool`

to_primitive(*value*)
Attempts to convert the input value into a Python *bool*.

validate_boolean(*value*)
Confirms the value is compatible with *bool*.

class typerighter.types.primitives.**FloatType**(**a*, ***kw*)
A *Number* implementation based on Python *float*.

NATIVE
alias of `float`

class typerighter.types.primitives.**IntegerType**(**a*, ***kw*)
A *Number* implementation based on Python *int*.

NATIVE
alias of `int`

class typerighter.types.primitives.**Number**(**a*, ***kw*)
Used for tracking the functionality common to numbers. The current implementation simply supports ranges of numbers.

class typerighter.types.primitives.**Primitive**(**a*, ***kw*)

A *Primitive* is the first *Type* that validates data by looking at its contents. It is intended as a common base class among primitive types and is not intended for regular use.

It extends validation by checking if input values match a list of possible choices.

It inherits *object* as its native type, allowing any data to pass validation.

validate_choices(*value*)

Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.primitives.**StringType**(**a*, ***kw*)

A type that captures the core needs for validating strings, which can then be extended in subclasses for validating specific types of strings.

Validation can be extended by using *min_length* or *max_length*, or by providing a regular expression compatible with Python's *re* module.

NATIVE

alias of `str`

is_falsy(*value*)

Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

to_primitive(*value*)

Converts the value to *str*.

1.3.2 Records

Records are a definition of a data structure that consists of one or more named types, eg. fields.

```
>>> class SomeRecord(types.Record):
...     name = types.StringType()
...     date = types.DateTimeType()
...
>>> sr = SomeRecord()
>>> sr.validate({'name': 'Jms Dnns', 'date': '2021-04-01T12:34:56.001337'})
```

1.3.3 Composite Types

class typerighter.types.composites.**Container**(**a*, ***kw*)

A *Container* is a foundational type, like *Primitive*, that allows some number of things to be held in a group, with no additional type checking past what a *Primitive* does.

New composite types should subclass this base type. It is not meant to be used directly.

NATIVE

alias of `object`

is_coercible(*value*)

Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate(*value*)
This validation function is the primary function responsible for calling all associated validators and for managing any details related to aggregation of validation results.

Parameters **value** (*object*) – The value to convert

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.composites.**ListType**(**a*, ***kw*)
A *ListType* is a *Container* implemented with a *list*.

NATIVE
alias of `list`

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*, ***kw*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*, ***kw*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate(*value*)
This validation function is the primary function responsible for calling all associated validators and for managing any details related to aggregation of validation results.

Parameters **value** (*object*) – The value to convert

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.composites.**SumType**(**a*, ***kw*)
Some languages call this a *Union Type*. The idea is to allow validation to pass if just one validator, from a list of two or more types, accepts it.

NATIVE
alias of `object`

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*, ***kw*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*, ***kw*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate(*value*)
This validation function is the primary function responsible for calling all associated validators and for managing any details related to aggregation of validation results.

Parameters **value** (*object*) – The value to convert

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

1.3.4 Time Keeping

class typerighter.types.timekeeping.**DateTimeType**(**a*, ***kw*)

NATIVE
alias of datetime.datetime

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.timekeeping.**TimeType**(**a*, ***kw*)

NATIVE
alias of datetime.time

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

1.3.5 Network Addresses

class typerighter.types.net.**EmailType**(**a*, ***kw*)

NATIVE
alias of str

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.net.**IPAddressType**(**a*, ***kw*)

NATIVE
alias of str

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.net.**IPv4Type**(**a*, ***kw*)

NATIVE
alias of str

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.net.**IPv6Type**(**a*, ***kw*)

NATIVE
alias of str

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.net.**MACAddressType**(**a*, ***kw*)

NATIVE
alias of str

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

class typerighter.types.net.**URLType**(**a*, ***kw*)

NATIVE
alias of str

is_coercible(*value*)
Checks a value for whether or not it can be converted to the correct type. Falls back to the stricter *is_type_match* if *self.strict* is True.

Parameters **value** (*object*) – The value to inspect

is_falsy(*value*)
Checks a value and responds saying whether the *Type* considers it falsy.

Parameters **value** (*object*) – The value to inspect

Returns True or False

is_type_match(*value*)
Checks if a value is an instance of this Type's native type. :param object value: The value to inspect

to_native(*value*)
Converts a value to the native form of this type

Parameters **value** (*object*) – The value to convert

to_primitive(*value*)
Converts a value to the primitive form of this type

Parameters **value** (*object*) – The value to convert

to_schematic()
Returns a Type's Schematic

validate_choices(*value*)
Checks if a choices list has been set and then if *value* is in that list.

1.3.6 Spatial Types

1.3.7 Cryptography

1.4 Working With Views

Views are mutable structures that store data based on `Record` instances. The field names are the same, except instead of getting type instances for all the attributes, you get fields that can store and delete data.

In addition to essentially being a *Mutable Record*, a View makes it easy to drop in place of existing modeling systems, *like maybe, Schematics…*

Views are created with `Type` instances, and they can be instantiated with data.

```
>>> view = SomeRecord().to_view({'foo': 'bar'})
>>> view.foo
'bar'
```

1.4.1 API

class typerighter.views.**Field**(*name*)
A descriptor used to join a mutable View instance, that stores data, with the immutable Type instance, that only defines methods for operating on data.

class typerighter.views.**View**(*record*, *data=None*, *native=True*, *primitive=False*)
A View combines a *Record* with a dictionary to provide an object modeled after the record that can store data in a familiar object oriented manner.

to_native(***convert_args*)
Uses the record's `to_native` to convert view data.

to_primitive(***convert_args*)
Uses the record's `to_primitive` to convert view data.

validate()
Uses the record's `validate` function with view data.

typerighter.views.**to_view**(*record*, *data=None*, ***view_config*)
Takes both a record and some data and produces View instance.

Parameters

- **record** (*Type*) – The type that defines the view's shape
- **data** (*dict*) – Any initial data for the view's fields

1.5 Type Schemas

Or, a *schematic*.

class typerighter.schematics.**Schematic**(*klass*)
A Schematic is a object that maintains a Type's argspec. It exists as a class to provide a namespace for relevant values.

typerighter.schematics.**extract_argspec**(*klass*)
Inspects a klass and creates a dict of keyword arguments and their default values.

Parameters **klass** (*class*) – The class definition to inspect

Returns a dictionary of default values found in argspec for class's init

typerighter.schematics.**init_arg_capture**(*method*)
A decorator that wraps a Type's *__init__* method for the purpose of capturing the arguments used when a Type is instantiated so it can then update the instance's argspec with what was actually used.

1.6 Project Architecture

TypeRighter is a toolkit for structuring, validating, and reshaping data.

Using the toolkit means using one or more of the following things:

- `Type`: a classification of some data which describes how to verify arbitrary data for coherence.
- `Record`: a structure of data that has type instances, called _fields_, for attributes.
- `Schematic`: the map of arguments used to instantiate either a `Type` or a `Record`.

- `View`: a class that let's you interact with a `Record` and some data as though it were an object instance.

1.6.1 Metaprogramming

The design of both `Type` and `Record` relies on metaprogramming to collect information about the way you choose to use them.

Generally speaking, metaprogramming is a way for programs to treat code like data. We can read, generate, analyze, or transform code, or modify it while running.

More specifically, TypeRighter can inspect the attributes and functions on any type at the moment a user creates one. This allows it to:

- make lists of all member variables
- make a list of all functions that start with *someprefix_*

And with that metadata users can:

- map out the steps for complex data validation
- generate a SQL statement automatically
- easily define datatype conversion pipelines

1.6.2 Metadata

All `Type` and `Record` definitions have values for:

- `_validate_functions`
- `_schematic`

Records use two extra fields:

- `_fields`
- `_field_functions`

1.6.3 Types

A type's `validate()` function will call each function listed in `_validate_functions` on its input.

The metaclass can be told about new validation functions by adding functions with `validate_` as a prefix, ie. `validate_uppercase`.

```
class StrictStringType(StringType):
    def validate_strict(self, value):
        ...
```

1.6.4 Records

Records introduce the concept of a field by associating a name with a type. It adds two fields of metadata to the class definition.

Let's define a record with a string stored as field `s`.

```
class Foo(Record):
    s = StringType(required=True)
```

Fields defined like this are stored as `_fields`.

It is also possible to use a function to generate field values.

```
class Foo(Record):
    def field_s(self):
        return 'an actual string'
```

Functions that behave like fields have a prefix `field_`, similar to the behavior for validation functions. This field is stored as `_field_functions`.

1.6.5 Views

We get views

1.6.6 Errors

Validation throws exceptions and we always throw exceptions

1.7 Working with CSVs

Word

1.8 JSON APIs

Word

1.9 Environment

I like to use simple setups, so I use a virtualenv and install a development copy of TypeRighter into that, along with the tools I use for dev and building the docs.

1.9.1 Get Repo

Get the source.

```
$ cd ~/Code
$ git clone https://github.com/jmsdnns/typerighter
$ cd typerighter
```

Virtual Env

If you dont have an environment setup already, create one in the project directory.

```
$ python3 -m venv venv
$ source venv/bin/activate
```

1.9.2 Install w/ Extras

Install typerighter into the virtualenv as a pointer to this working directory.

```
$ git clone https://github.com/jmsdnns/typerighter
$ cd typerighter
$ pip install -e .[dev,docs]
```

1.10 Testing

Typerighter uses *pytest*.

```
$ pytest tests
============================== test session starts ==============================
platform darwin -- Python 3.7.4, pytest-5.2.1, py-1.8.0, pluggy-0.13.0
rootdir: ...
plugins: cov-2.8.1
collected 69 items

tests/test_booleantypes.py ....                                          [  5%]
tests/test_cache.py .                                                    [  7%]
tests/test_datetimetypes.py ..                                           [ 10%]
tests/test_emailtype.py ..                                               [ 13%]
tests/test_ipaddresstypes.py ....                                        [ 18%]
tests/test_listtypes.py ......                                           [ 27%]
tests/test_macaddresses.py ..                                            [ 30%]
tests/test_primitives.py .........                                       [ 43%]
tests/test_records.py ............                                       [ 60%]
tests/test_schematics.py .                                               [ 62%]
tests/test_stringtypes.py .......                                        [ 72%]
tests/test_sumtypes.py ....                                              [ 78%]
tests/test_types.py ..........                                           [ 92%]
tests/test_urltypes.py ..                                                [ 95%]
tests/test_views.py ...                                                  [100%]
```

(continues on next page)

(continued from previous page)

```
================================ 69 passed in 0.16s ================================
```

Nice.

1.11 Writing Docs

Our docs are hosted on Read The Docs at https://typerighter.readthedocs.io/

Build the docs like so.

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

Committing doc changes to master will alert Read The Docs to do a new build.

1.12 Schematics

I am the original author of The Schematics Project.

The community took over the project many years ago and I spent more time with languages that weren't Python. My curiosity made me wonder what Schematics would look like if I built it from scratch today…

In particular, it felt important to me to put all of the type validation and conversion functions somewhere separate from where all the data is kept. This could be seen as a thorny opinion with a preference for functional programming models, which is basically right… But there's more to it, too.

The Schematics project also showed me that programming models can *feel* heavy from the cognitive load of a namespace that contains everything necessary for managing a metaprogramming framework & complex data together. I wanted something that would feel more focused, and thus more straight forward and simple.

I got started.

1.12.1 Types & Data

By the time I had built a rough draft, I started to believe I had found a new way of thinking about data that deserved to be articulated as code.

TypeRighter would put all of the logic for validating and converting data into a simple structure, a `Type`, and types would only *operate* on data, *never* storing it.

The namespace of a type is concerned just with configuring a type's behavior.

Records took this pretty far. They actually ignore attempts to set attributes if the name is already associated with a typerighter `Type`.

1.12.2 Mutability as Views

Schematics structured itself such that it behaved a lot like a typical Python class. You could store data on the model and then call methods which would operate on that data.

TypeRighter's approach is to consider any kind of mutable object a `View`. It is a way of working with data that will look familiar to both Schematics users and Python users.

In using different terminology, a *view*, for a familiar concept, a mutable class instance, we make it clear the structure is just one way of looking at the data, and this structure can be configured for unique behaviors that other views of the same data do not have, etc.

1.12.3 You Should Try It

If you are here from the Schematics community, you're gonna love this library!

1.13 License

aww yeah

PYTHON MODULE INDEX

t

INDEX

L

M

N

P

S

T

U

V