ValidX Documentation

Release 0.6.1

Cottonwood Technology

May 29, 2020

Contents:

ValidX is *fast*, powerful, and flexible validator with sane syntax.

```
from validx import Dict, Str

schema = Dict({"message": Str()})
data = {"message": "ValidX is cool!"}

print(schema(data))
```

```
{'message': 'ValidX is cool!'}
```

CHAPTER 1

Usage

1.1 Installation

The library is shipped with two interchangeable parts: pure Python and optimized Cython versions. In the most cases, you will get a ready to use binary wheel during installation from PyPI.

```
pip install validx
```

However, if it fails to find a wheel compatible with your OS, it will try to install source code tarball and compile it on the fly. To get the optimized version, you have to have Cython (in addition to C/C++ compiler and Python header files) installed **before** ValidX installation. If it fails to import Cython during setup, no compilation will be done. And you will get the pure Python version of the library.

You can check which version has been installed using the following code:

```
>>> import validx
>>> validx.__impl__
'Cython'
```

1.2 Quick Start

Let's build a simple validator for some web-application endpoint, which performs full-text search with optional filtering by tags:

```
from validx import Dict, List, Str, Int

search_params = Dict(
    {
        "query": Str(minlen=3, maxlen=500),     # Search query
        "tags": List(Str(pattern=r"^[\w]+$")), # Optional list of tags
        "limit": Int(min=0, max=100),           # Pagination parameters
```

(continues on next page)

(continued from previous page)

```
        "offset": Int(min=0),
    },
    defaults={
        "limit": 100,
        "offset": 0,
    },
    optional=["tags"],
)
```

And test it:

```
assert search_params({"query": "Craft Beer"}) == {
    "query": "Craft Beer",
    "limit": 100,
    "offset": 0,
}
assert search_params({"query": "Craft Beer", "offset": 100}) == {
    "query": "Craft Beer",
    "limit": 100,
    "offset": 100,
}
assert search_params({"query": "Craft Beer", "tags": ["APA"]}) == {
    "query": "Craft Beer",
    "tags": ["APA"],
    "limit": 100,
    "offset": 0,
}
```

See *Reference* for complete list of available validators and their parameters.

1.3 Error Handling

Each validator tries to handle as much as possible. It means, a result exception raised by the validator may contain many errors.

```
from validx import exc

try:
    search_params({"limit": 200})
except exc.ValidationError as e:
    error = e

error.sort()
print(error)
```

```
<SchemaError(errors=[
    <limit: MaxValueError(expected=100, actual=200)>,
    <query: MissingKeyError()>
])>
```

As you can see, the result exception `error` has type *validx.exc.SchemaError*, which contains two errors: *validx.exc.MaxValueError* and *validx.exc.MissingKeyError*.

To unify error handling, each exception provides Sequence interface. It means, you can iterate them, get by index and sort nested errors.

```
# SchemaError iteration is done over its nested errors
for suberror in error:
    print(suberror)
```

```
<limit: MaxValueError(expected=100, actual=200)>
<query: MissingKeyError()>
```

```
# Error of other class just yields itself during iteration
for suberror in error[0]:
    print(suberror)
```

```
<limit: MaxValueError(expected=100, actual=200)>
```

Take a note on calling `error.sort()` before printing the error. It sorts nested errors by their contexts.

Context is a full path to the failed member of validated structure. For example, let's add an `order` parameter to the `search_params` schema, which accepts list of tuples `[(field_name, sort_direction), ...]`:

```
from validx import exc, Dict, List, Tuple, Str, Int

search_params = Dict(
    {
        "query": Str(minlen=3, maxlen=500),
        "tags": List(Str(pattern=r"^[\w]+$")),
        "limit": Int(min=0, max=100),
        "offset": Int(min=0),
        "order": List(
            Tuple(
                Str(options=["name", "added"]),  # Field name
                Str(options=["asc", "desc"]),    # Sort direction
            ),
        ),
    },
    defaults={
        "limit": 100,
        "offset": 0,
        "order": [("added", "desc")],
    },
    optional=["tags"],
)
```

And pass invalid value into it:

```
try:
    search_params({
        "query": "Craft Beer",
        "order": [("name", "ascending"), ("description", "asc")],
    })
except exc.ValidationError as e:
    error = e

error.sort()
print(error)
```

```
<SchemaError(errors=[
    <order.0.1: OptionsError(expected=frozenset({...}), actual='ascending')>,
```

(continues on next page)

(continued from previous page)

```
    <order.1.0: OptionsError(expected=frozenset({...}), actual='description')>
])>
```

Take a note on contexts, for example `order.0.1`. It means, that the error has occurred at `order` dictionary key, at the first element of the list (index `0`), and at the second element of the tuple (index `1`).

Technically error context is a deque, so it can be easily inspected:

```
print(error[0].context)
```

```
deque(['order', 0, 1])
```

The library also provides special context markers, to distinguish special cases (such as failed pipeline steps) from dictionary keys and list/tuple indexes. See *Context Markers* section for details.

There is also *Error Formatter*, that returns a human friendly error messages.

```
try:
    search_params({"limit": 200})
except exc.ValidationError as e:
    for context, message in exc.format_error(e):
        print("%s: %s" % (context, message))
```

```
limit: Expected value  100, got 200.
query: Required key is not provided.
```

It is probably not what you want. It does not provide any localization, for instance, but you can look over its sources and figure out how to build your own one. So its purpose is mostly to be an example rather than a useful tool.

1.4 Reusable Validators

There is a quite common task to create a bunch of basic validators in a project, and then build complex ones from them.

For example, you have validators for handling resource IDs and names:

```
from validx import Int, Str

resource_id = Int(min=1)
resource_name = Str(minlen=1, maxlen=200)
```

You can use them directly in a complex validator, because they work as pure functions and produce no side effects during validation.

```
from validx import Dict

resource_update_params = Dict({
    "id": resource_id,
    "name": resource_name,
})
```

However, importing each basic validator might be tedious. So you can use *Instance Registry* provided by the library.

```
from validx import instances, Int, Str, Dict

Int(alias="resource_id", min=1)
Str(alias="resource_name", minlen=1, maxlen=200)

resource_update_params = Dict({
    "id": instances.get("resource_id"),
    "name": instances.get("resource_name"),
})
```

1.5 Cloning Validators

Another common task is to create a new validator, based on existent one with slightly different parameters. You can use cloning for such purpose.

Cloning might look a bit tricky, so here is the list of examples, that covers the most possible use cases.

Example 1. Create a validator adding a new constraint to existent one.

```
from validx import Int

resource_id = Int(min=1)

print(resource_id)
print(resource_id.clone(nullable=True))
```

```
<Int(min=1)>
<Int(nullable=True, min=1)>
```

Example 2. Create a validator updating options constraint of base one.

```
from validx import Str

resource_action = Str(options=("create", "update", "read", "delete"))
email_action = resource_action.clone(
    {
        "options-": ["update"],            # Remove "update" from options
        "options+": ["spam", "archive"],   # Add "spam" and "archive" to options
    }
)

print(sorted(resource_action.options))
print(sorted(email_action.options))
```

```
['create', 'delete', 'read', 'update']
['archive', 'create', 'delete', 'read', 'spam']
```

Example 3. Create a validator updating constraint of nested validator of base one.

```
from validx import Tuple, Str

resource_order = Tuple(
    Str(options=("name", "added")),  # Field name
    Str(options=("asc", "desc")),    # Sort direction
```

(continues on next page)

(continued from previous page)

```
)
article_order = resource_order.clone(
    {
        "items.0.options+": ["title"],
        "items.0.options-": ["name"],
    },
)

print(sorted(resource_order.items[0].options))
print(sorted(article_order.items[0].options))
```

```
['added', 'name']
['added', 'title']
```

In general, clone syntax looks like this.

```
validator.clone({
    "path.to.validator.param": "new value",         # set param of validator
    "path.to.validator+": {"param": "new value"},   # update several params
    "path.to.validator-": ["param_1", "param_2"],   # unset params of validator
    "path.to.dict.key": "value",                    # set key of dict
    "path.to.dict+": {"key": "value"},              # update several keys of dict
    "path.to.dict-": ["key_1", "key_2"],            # remove keys from dict
    "path.to.set+": ["value 1", "value 2"],         # update set
    "path.to.set-": ["value 1", "value 2"],         # remove values from set
    "path.to.list+": ["value 1", "value 2"],        # extend list
    "path.to.list-": ["value 1", "value 2"],        # remove values from list
})
```

If path to parameter doesn't contain any dot, it can be passed as keyword argument:

```
validator.clone(param_1="new value 1", param_2="new value 2")
```

1.6 Dumping & Loading Validators

Each validator can be dumped into a dictionary and loaded from such dictionary. It might be useful to serialize validators into JSON or load them from configuration.

```
from pprint import pprint
from validx import Validator, Int

resource_id = Int(min=1)
dumped = resource_id.dump()

pprint(dumped)
print(Validator.load(dumped))
```

```
{'__class__': 'Int', 'min': 1}
<Int(min=1)>
```

You can register validators using aliases, and use them or clone them later during loading process.

```
print(
    Validator.load({
        "__class__": "Int",
        "alias": "resource_id",
        "min": 1,
    })
)
print(
    Validator.load({
        "__clone__": "resource_id",
        "update": {
            "alias": "nullable_resource_id",
            "nullable": True,
        },
    })
)
print(Validator.load({"__use__": "nullable_resource_id"}))
```

```
<Int(min=1)>
<Int(nullable=True, min=1)>
<Int(nullable=True, min=1)>
```

1.7 MultiDict Validation

Popular web-frameworks parse `application/x-www-form-urlencoded` data into so-called `MultiDict` structures. There is no standard interface, but implementations more or less compatible. The main purpose of the structure is to pass arrays using key-value pairs, where values with the same key are grouped into an array.

The search query from *Quick Start* section can look like this:

```
GET /catalog/search?query=Craft+Beer&tags=APA&tags=IPA HTTP/1.1
```

Let's rewrite the validator to handle such query:

```
from validx import Dict, List, Str, Int

search_params = Dict(
    {
        "query": Str(minlen=3, maxlen=500),
        "tags": List(Str(pattern=r"^[\w]+$")),
        "limit": Int(min=0, max=100, coerce=True), # Coerce ``str`` to ``int``
        "offset": Int(min=0, coerce=True),
    },
    defaults={
        "limit": 100,
        "offset": 0,
    },
    optional=["tags"],
    multikeys=["tags"],  # Handle ``tags`` as a sequence
)
```

And it can be used like this:

```
# AIOHTTP request handler
async def catalog_search(request):
```

(continues on next page)

(continued from previous page)

```
    params = search_schema(request.url.query)
    # params == {
    #     "query": "Craft Beer",
    #     "tags": ["APA", "IPA"],
    #     "limit": 0,
    #     "offset": 0,
    # }
    ...
```

ValidX has been tested against the following implementations of `MultiDict`:

- WebOb MultiDict;
- Werkzeug MultiDict;
- MultiDict (that has been extracted from AIOHTTP).

1.8 Multiple-Step Validation

Sometimes you need to split up validation process into several steps. Prevalidate some common structure on the first one, and make final validation on the latter one.

For example, here is the schema for validation of JSON-RPC 2.0 request:

```
from validx import Dict, Int, Str, Const, OneOf, Any

jsonrpc = Dict(
    {
        "jsonrpc": Const("2.0"),
        "id": OneOf(
            Int(nullable=True),
            Str(minlen=1, maxlen=100),
        ),
        "method": Str(minlen=1, maxlen=100),
        "params": Any(),
    },
    optional=("id", "params"),
)
```

Take note of *`validx.py.Any`* usage. It accepts literally any value, just like as we need here, because parameters of concrete method will be validated on the next step.

```
login_params = Dict({
    "username": Str(minlen=1, maxlen=100),
    "password": Str(minlen=1, maxlen=100),
})

request = {
    "jsonrpc": "2.0",
    "id": 1,
    "method": "login",
    "params": {"username": "jdoe", "password": "qwerty"},
}

assert jsonrpc(request) == request
assert login_params(request["params"]) == request["params"]
```

1.9 Recursive Structure Validation

Let's see a real-world example. A web application accepts search query as JSON in the following notation:

```
{"<function>": ["<arg_1>", "<arg_2>", ...]}
```

Simple comparison function accepts only two arguments: field name and some value to compare with. For example:

```
{"eq": ["type", "whiskey"]}                    # type == "whiskey"
{"ne": ["status", "out_of_stock"]}             # status != "out_of_stock"
{"in": ["origin", ["Scotland", "Ireland"]]}    # origin in ["Scotland", "Ireland"]
{"gt": ["age", 10]}                            # age > 10
{"lt": ["age", 20]}                            # age < 20
```

And there is also compound functions, that can combine simple and other compound ones. For example:

```
# type == "whiskey" and age > 10 and age < 20
{
    "and": [
        {"eq": ["type", "whiskey"]},
        {"gt": ["age", 10]},
        {"lt": ["age", 20]},
    ]
}
```

There is obviously recursive validator needed. Here is how it can be built:

```
from validx import Dict, List, Tuple, OneOf, Any, LazyRef, Str

# Validator for simple function
simple_query = Dict(
    extra=(
        # accept dict key as the following function names
        Str(options=("eq", "ne", "in", "lt", "gt")),

        # accept dict value as a tuple of two elements
        Tuple(
            Str(),  # field name
            Any(),  # parameter,
                    # that will be validated on the next step,
                    # taking into account type of specified field
                    # and comparison function
        ),
    ),
    minlen=1,  # at least one function should be specified
)

# Validator for compound function
compound_query = Dict(
    extra=(
        # accept dict key as the following function names
        Str(options=("and", "or", "not")),

        # accept dict value as a list of other functions
        List(
            # make a lazy reference on ``query_dsl`` validator,
            # which is defined below,
```

(continues on next page)

(continued from previous page)

```
            # and allow maximum 5 levels of recursion
            LazyRef("query_dsl", maxdepth=5)
        ),
    ),
    minlen=1,  # again, at least one function should be specified
)

# And the final validator
query_dsl = OneOf(
    simple_query,
    compound_query,

    # register the validator under ``query_dsl`` alias,
    # so it will be accessible via ``LazyRef`` above
    alias="query_dsl",
)
```

Here we use *validx.py.LazyRef* to create circular reference on the parent validator. Each time it is called, it increments its recursive call depth and checks the limit. If the limit is reached, it raises *validx.exc.RecursionMaxDepthError*.

Warning: Be careful cloning such validators. You should register a clone using new alias, and also update `use` parameter of `LazyRef` to the same new alias. If you don't do this, you will definitely get some fun chasing a bunch of sneaky bugs.

Let's validate a sample query:

```
# (
#   type == "whiskey"
#   and origin in ["Scotland", "Ireland"]
#   and age > 10
#   and age < 20
#   and status != "out_of_stock"
# )
query = {
    "and": [
        {"eq": ("type", "whiskey")},
        {"in": ("origin", ["Scotland", "Ireland"])},
        {"gt": ("age", 10)},
        {"lt": ("age", 20)},
        {"ne": ("status", "out_of_stock")},
    ],
}
assert query_dsl(query) == query
```

CHAPTER 2

Reference

2.1 Abstract

class validx.py.**Validator**(*alias=None*, *replace=False*)

Abstract Base Validator

Parameters

- **alias** (*str*) – if it specified, the instance will be added into registry, see *validx.py.instances.add()*.
- **replace** (*bool*) – if it is `True` and `alias` specified, the instance will be added into registry, replacing any existent validator with the same alias, see *validx.py.instances.put()*.
- ****kw** – concrete validator attributes.

__call__(*value*, *_Validator__context=None*)

Validate value.

This is an abstract method, and it should be implemented by descendant class.

static load(*params*, *update=None*, *unset=None*, ***kw*)

Load validator.

```
>>> Validator.load({
...     "__class__": "Int",
...     "min": 0,
...     "max": 100,
... })
<Int(min=0, max=100)>

>>> # Add into registry
>>> some_int = Validator.load({
...     "__class__": "Int",
...     "min": 0,
```

(continues on next page)

(continued from previous page)

```
...     "max": 100,
...     "alias": "some_int",
... })
>>> some_int
<Int(min=0, max=100)>

>>> # Load from registry by alias
>>> Validator.load({"__use__": "some_int"}) is some_int
True

>>> # Clone from registry by alias
>>> Validator.load({
...     "__clone__": "some_int",
...     "update": {
...         "min": -100,
...     },
... })
<Int(min=-100, max=100)>
```

dump()

Dump validator.

```
>>> Int(min=0, max=100).dump() == {
...     "__class__": "Int",
...     "min": 0,
...     "max": 100,
... }
True
```

clone(*update=None*, *unset=None*, ***kw*)

Clone validator.

```
>>> some_enum = Int(options=[1, 2, 3])
>>> some_enum.clone(
...     {
...         "nullable": True,
...         "options+": [4, 5],
...         "options-": [1, 2],
...     }
... ) == Int(nullable=True, options=[3, 4, 5])
True
```

In fact, the method is a shortcut for:

```
self.load(self.dump(), update, **kw)
```

2.2 Numbers

class validx.py.**Int**(*nullable=False*, *coerce=False*, *min=None*, *max=None*, *options=None*, *alias=None*, *replace=False*)

Integer Number Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.

- **coerce** (*bool*) – try to convert non-integer value to `int`.
- **min** (*int*) – lower limit.
- **max** (*int*) – upper limit.
- **options** (*iterable*) – explicit enumeration of valid values.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `not isinstance(value, int)` and `not self.coerce`;
  - if `int(value)` raises `ValueError` or `TypeError`.
- ***MinValueError*** – if `value < self.min`.
- ***MaxValueError*** – if `value > self.max`.
- ***OptionsError*** – if `value not in self.options`.

Note It implicitly converts `float` to `int`, if `value.is_integer() is True`.

class validx.py.**Float**(*nullable=False*, *coerce=False*, *nan=False*, *inf=False*, *min=None*, *max=None*, *alias=None*, *replace=False*)

Floating Point Number Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.
- **coerce** (*bool*) – try to convert non-float value to `float`.
- **nan** (*bool*) – accept `Not-a-Number` as a valid value.
- **inf** (*bool*) – accept `Infinity` as a valid value.
- **min** (*float*) – lower limit.
- **max** (*float*) – upper limit.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `not isinstance(value, float)` and `not self.coerce`;
  - if `float(value)` raises `ValueError` or `TypeError`.
- ***FloatValueError*** –
  - if `math.isnan(value)` and `not self.nan`;
  - if `math.isinf(value)` and `not self.inf`.
- ***MinValueError*** – if `value < self.min`.
- ***MaxValueError*** – if `value > self.max`.

Note It always converts `int` to `float`.

2.3 Chars

class validx.py.**Str**(*nullable=False*, *encoding=None*, *minlen=None*, *maxlen=None*, *pattern=None*, *options=None*, *alias=None*, *replace=False*)

Unicode String Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.
- **encoding** (*str*) – try to decode byte-string to `str/unicode`, using specified encoding.
- **minlen** (*int*) – lower length limit.
- **maxlen** (*int*) – upper length limit.
- **pattern** (*str*) – validate string using regular expression.
- **options** (*iterable*) – explicit enumeration of valid values.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `not isinstance(value, str)` (Python 3.x) or `not isinstance(value, unicode)` (Python 2.x).
- ***StrDecodeError*** – if `value.decode(self.encoding)` raises `UnicodeDecodeError`.
- ***MinLengthError*** – if `len(value) < self.minlen`.
- ***MaxLengthError*** – if `len(value) > self.maxlen`.
- ***PatternMatchError*** – if `value` does not match `self.pattern`.
- ***OptionsError*** – if `value not in self.options`.

class validx.py.**Bytes**(*nullable=False*, *minlen=None*, *maxlen=None*, *alias=None*, *replace=False*)

Byte String Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.
- **minlen** (*int*) – lower length limit.
- **maxlen** (*int*) – upper length limit.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `not isinstance(value, bytes)`.
- ***MinLengthError*** – if `len(value) < self.minlen`.
- ***MaxLengthError*** – if `len(value) > self.maxlen`.

2.4 Date and Time

class validx.py.**Date**(*nullable=False*, *unixts=False*, *format=None*, *parser=None*, *min=None*, *max=None*, *relmin=None*, *relmax=None*, *tz=None*, *alias=None*, *replace=False*)

Date Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.
- **unixts** (*bool*) – convert Unix timestamp (`int` or `float`) to `date`.
- **format** (*str*) – try to parse `date` from `str` (Python 3.x) or `basestring` (Python 2.x), using `datetime.strptime(value, self.format).date()`.
- **parser** (*callable*) – try to parse `date` from `str` (Python 3.x) or `basestring` (Python 2.x), using `self.parser(value).date()`.
- **min** (*date*) – absolute lower limit.
- **max** (*date*) – absolute upper limit.
- **relmin** (*timedelta*) – relative lower limit.
- **relmax** (*timedelta*) – relative upper limit.
- **tz** (*tzinfo*) – timezone, see notes below.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `isinstance(value, (int, float))` and `not self.unixts`;
  - if `not isinstance(value, date)`.
- ***DatetimeParseError*** –
  - if `datetime.strptime(value, self.format)` raises `ValueError`;
  - if `self.parser(value)` raises `ValueError`.
- ***MinValueError*** –
  - if `value < self.min`;
  - if `value < date.today() + self.relmin`.
- ***MaxValueError*** –
  - if `value > self.max`;
  - if `value > date.today() + self.relmax`.

Note Relative limits are calculated adding deltas to current date, use negative `relmin`/`relmax` to specify date in the past.

Note It implicitly converts `datetime` to `date`. If timezone is specified and `datetime` object is timezone-aware, it will be arranged to specified timezone first.

Note If timezone is specified, it will be used in conversion from Unix timestamp. In fact, it will create `datetime` object in UTC, using `datetime.fromtimestamp(value, UTC)`. And then arrange it to specified timezone, and extract date part.

class validx.py.**Time**(*nullable=False*, *format=None*, *parser=None*, *min=None*, *max=None*, *alias=None*, *replace=False*)

Time Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.
- **format** (*str*) – try to parse `time` from `str` (Python 3.x) or `basestring` (Python 2.x), using `datetime.strptime(value, self.format).time()`.
- **parser** (*callable*) – try to parse `time` from `str` (Python 3.x) or `basestring` (Python 2.x), using `self.parser(value).time()`.
- **min** (*time*) – lower limit.
- **max** (*time*) – upper limit.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `not isinstance(value, time)`.
- ***DatetimeParseError*** –
  - if `datetime.strptime(value, self.format)` raises `ValueError`;
  - if `self.parser(value)` raises `ValueError`.
- ***MinValueError*** – if `value < self.min`.
- ***MaxValueError*** – if `value > self.max`.

class validx.py.**Datetime**(*nullable=False*, *unixts=False*, *format=None*, *parser=None*, *min=None*, *max=None*, *relmin=None*, *relmax=None*, *tz=None*, *alias=None*, *replace=False*)

Date & Time Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.
- **unixts** (*bool*) – convert Unix timestamp (`int` or `float`) to `datetime`.
- **format** (*str*) – try to parse `datetime` from `str` (Python 3.x) or `basestring` (Python 2.x), using `datetime.strptime(value, self.format)`.
- **parser** (*callable*) – try to parse `datetime` from `str` (Python 3.x) or `basestring` (Python 2.x), using `self.parser(value)`.
- **min** (*datetime*) – absolute lower limit.
- **max** (*datetime*) – absolute upper limit.
- **relmin** (*timedelta*) – relative lower limit.
- **relmax** (*timedelta*) – relative upper limit.
- **tz** (*tzinfo*) – timezone.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;

  - if `isinstance(value, (int, float))` and `not self.unixts`;
  - if `not isinstance(value, datetime)`.
- ***DatetimeParseError*** –
  - if `datetime.strptime(value, self.format)` raises `ValueError`;
  - if `self.parser(value)` raises `ValueError`.
- ***DatetimeTypeError*** –
  - if `self.tz is None and value.tzinfo is not None`;
  - if `self.tz is not None and value.tzinfo is None`;
- ***MinValueError*** –
  - if `value < self.min`;
  - if `self.tz is None and value < datetime.now() + self.relmin`.
  - if `self.tz is not None and value < datetime.now(UTC).astimezone(self.tz) + self.relmin`.
- ***MaxValueError*** –
  - if `value > self.max`;
  - if `self.tz is None and value > datetime.now() + self.relmax`.
  - if `self.tz is not None and value > datetime.now(UTC).astimezone(self.tz) + self.relmax`.

Note Relative limits are calculated adding deltas to midnight of current date, use negative `relmin`/`relmax` to specify date and time in the past.

2.5 Boolean

class validx.py.**Bool**(*nullable=False*, *coerce_str=False*, *coerce_int=False*, *alias=None*, *replace=False*)

Boolean Validator

Parameters

- **nullable** (*bool*) – accept `None` as a valid value.
- **coerce_str** (*bool*) –
  - accept values `["1", "true", "yes", "y", "on"]` as `True`;
  - accept values `["0", "false", "no", "n", "off"]` as `False`.
- **coerce_int** (*bool*) – accept `int` as valid value.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `isinstance(value, str)` and `not self.coerce_str`;
  - if `isinstance(value, int)` and `not self.coerce_int`;
  - if `not isinstance(value, bool)`.

- ***OptionsError*** – when string value is not valid name, see `coerce_str`.

2.6 Containers

class validx.py.**List**(*item*, *nullable=False*, *minlen=None*, *maxlen=None*, *unique=False*, *alias=None*, *replace=False*)

List Validator

Parameters

- **item** (`Validator`) – validator for list items.
- **nullable** (*bool*) – accept `None` as a valid value.
- **minlen** (*int*) – lower length limit.
- **maxlen** (*int*) – upper length limit.
- **unique** (*bool*) – drop duplicate items.

Raises

- ***InvalidTypeError*** – if `not isinstance(value, (list, tuple))`.
- ***MinLengthError*** – if `len(value) < self.minlen`.
- ***MaxLengthError*** – if `len(value) > self.maxlen`.
- ***SchemaError*** – with all errors, raised by item validator.

class validx.py.**Tuple**(**args*, ***kw*)

Tuple Validator

Parameters

- ***items** (`Validator`) – validators for tuple members.
- **nullable** (*bool*) – accept `None` as a valid value.

Raises

- ***InvalidTypeError*** – if `not isinstance(value, (list, tuple))`.
- ***TupleLengthError*** – if `len(value) != len(self.items)`.
- ***SchemaError*** – with all errors, raised by member validators.

class validx.py.**Dict**(*schema=None*, *nullable=False*, *minlen=None*, *maxlen=None*, *extra=None*, *defaults=None*, *optional=None*, *dispose=None*, *multikeys=None*, *alias=None*, *replace=False*)

Dictionary Validator

Parameters

- **schema** (*dict*) – schema validator in format `{<key>: <validator>}`.
- **nullable** (*bool*) – accept `None` as a valid value.
- **minlen** (*int*) – lower length limit.
- **maxlen** (*int*) – upper length limit.
- **extra** (*tuple*) – validators for extra keys and values in format `(<key_validator>, <value_validator>)`, it is used for keys are not presented in `schema`.
- **defaults** (*dict*) – default values for missing keys.

- **optional** (*list or tuple*) – list of optional keys.
- **dispose** (*list or tuple*) – list of keys that have to be silently removed.
- **multikeys** (*list or tuple*) – list of keys that have to be treated as lists of values, if input value is a `MultiDict` (see notes below), i.e. value of these keys will be extracted using `val = value.getall(key)` or `val = value.getlist(key)`.

Raises

- ***InvalidTypeError*** – if `not isinstance(value, collections.abc.Mapping)`.
- ***MinLengthError*** – if `len(value) < self.minlen`.
- ***MaxLengthError*** – if `len(value) > self.maxlen`.
- ***SchemaError*** – with all errors, raised by schema validators, extra validators, and missing required and forbidden extra keys.

Note on error raised by `extra` validators, context marker *validx.exc.Extra* will be used to indicate, which part of key/value pair is failed.

It has been tested against the following implementations of `MultiDict`:

- WebOb MultiDict;
- Werkzeug MultiDict;
- MultiDict.

However, it should work fine for other implementations, if the implementation is subclass of `collections.abc.Mapping`, and provides `getall()` or `getlist()` methods.

2.7 Pipelines

class validx.py.**AllOf**(**args*, ***kw*)
AND-style Pipeline Validator

All steps must be succeeded. The last step returns result.

Parameters ***steps** (Validator) – nested validators.

Raises ***ValidationError*** – raised by the first failed step.

Note it uses *validx.exc.Step* marker to indicate, which step is failed.

class validx.py.**OneOf**(**args*, ***kw*)
OR-style Pipeline Validator

The first succeeded step returns result.

Parameters ***steps** (Validator) – nested validators.

Raises ***SchemaError*** – if all steps are failed, so it contains all errors, raised by each step.

Note it uses *validx.exc.Step* marker to indicate, which step is failed.

2.8 Special

class validx.py.**LazyRef**(*use*, *maxdepth=None*, *alias=None*, *replace=False*)
Lazy Referenced Validator

It is useful to build validators for recursive structures.

```
>>> schema = Dict(
...     {
...         "foo": Int(),
...         "bar": LazyRef("schema", maxdepth=1),
...     },
...     optional=("foo", "bar"),
...     minlen=1,
...     alias="schema",
... )

>>> schema({"foo": 1})
{'foo': 1}

>>> schema({"bar": {"foo": 1}})
{'bar': {'foo': 1}}

>>> schema({"bar": {"bar": {"foo": 1}}})
Traceback (most recent call last):
    ...
validx.exc.errors.SchemaError: <SchemaError(errors=[
    <bar.bar: RecursionMaxDepthError(expected=1, actual=2)>
])>
```

Parameters

- **use** (*str*) – alias of referenced validator.
- **maxdepth** (*int*) – maximum recursion depth.

Raises ***RecursionMaxDepthError*** – if `self.maxdepth is not None` and current recursion depth exceeds the limit.

class validx.py.**Type**(*tp*, *nullable=False*, *coerce=False*, *min=None*, *max=None*, *minlen=None*, *maxlen=None*, *options=None*, *alias=None*, *replace=False*)

Custom Type Validator

Parameters

- **tp** (*type*) – valid value type.
- **nullable** (*bool*) – accept `None` as a valid value.
- **coerce** (*bool*) – try to convert value to `tp`.
- **min** (*tp*) – lower limit, makes sense only if `tp` provides comparison methods.
- **max** (*tp*) – upper limit, makes sense only if `tp` provides comparison methods.
- **minlen** (*int*) – lower length limit, makes sense only if `tp` provides `__len__()` method.
- **maxlen** (*int*) – upper length limit, makes sense only if `tp` provides `__len__()` method.
- **options** (*iterable*) – explicit enumeration of valid values.

Raises

- ***InvalidTypeError*** –
  - if `value is None` and `not self.nullable`;
  - if `not isinstance(value, self.tp)` and `not self.coerce`;

– if `self.tp(value)` raises `ValueError` or `TypeError`.

- ***MinValueError*** – if `value < self.min`.
- ***MaxValueError*** – if `value > self.max`.
- ***MinLengthError*** – if `len(value) < self.minlen`.
- ***MaxLengthError*** – if `len(value) > self.maxlen`.
- ***OptionsError*** – if `value not in self.options`.

class validx.py.**Const**(*value*, *alias=None*, *replace=False*)
Constant Validator

It only accepts single predefined value.

Parameters **value** – expected valid value.

Raises ***OptionsError*** – if `value != self.value`.

class validx.py.**Any**(*alias=None*, *replace=False*)
Pass-Any Validator

It literally accepts any value.

2.9 Class Registry

validx.py.classes.**add**(*class_*)
Add validator class into the registry

Parameters **class** (Validator) – class to register.

Raises **AssertionError** – if there is a class in the registry with the same name, i.e. `class_.__name__`.

Returns unmodified passed class, so the function can be used as a decorator.

validx.py.classes.**get**(*classname*)
Get validator class from the registry

Parameters **classname** (*str*) – name of class to get.

Raises **KeyError** – if there is no class in the registry with the specified name.

Returns previously registered class.

2.10 Instance Registry

validx.py.instances.**add**(*alias*, *instance*)
Add validator into the registry

Parameters

- **alias** (*str*) – alias of the validator.
- **instance** (Validator) – instance of the validator.

Raises **AssertionError** – if there is an instance in the registry with the same alias.

Returns unmodified instance of passed validator.

validx.py.instances.**put**(*alias*, *instance*)
Put validator into the registry

The function silently replaces any instance with the same alias.

Parameters

- **alias** (*str*) – alias of the validator.
- **instance** (Validator) – instance of the validator.

Returns unmodified instance of passed validator.

validx.py.instances.**get**(*alias*)
Get validator from the registry

Parameters **alias** (*str*) – alias of the validator.

Raises **KeyError** – if there is no registered validator under the specified alias.

Returns previously registered validator.

validx.py.instances.**clear**()
Clear the registry

2.11 Errors

The class hierarchy for exceptions is:

- ValueError (built-in)
  - *ValidationError*
    - *ConditionError*
      - *InvalidTypeError*
      - *OptionsError*
      - *MinValueError*
      - *MaxValueError*
      - *FloatValueError*
      - *StrDecodeError*
      - *MinLengthError*
      - *MaxLengthError*
      - *TupleLengthError*
      - *PatternMatchError*
      - *DatetimeParseError*
      - *DatetimeTypeError*
      - *RecursionMaxDepthError*
    - *MappingKeyError*
      - *ForbiddenKeyError*
      - *MissingKeyError*

* *SchemaError*

class validx.exc.**ValidationError**(*context=None*, **args*, ***kw*)

Validation Error Base Class

Parameters

- **context** (*deque*) – error context, empty `deque` by default.
- ****kw** – concrete error attributes.

Since validators try to process as much as possible, they can raise multiple errors (wrapped by *validx.exc.SchemaError*). To unify handling of such errors, each validation error provides `Sequence` interface. It means, you can iterate them, get their length, get nested errors by index, and sort nested errors by context.

Error context is a full path, that indicates where the error occurred. It contains mapping keys, sequence indexes, and special markers (see *validx.exc.Extra* and *validx.exc.Step*).

```
>>> from validx import exc, Dict, List, Int

>>> schema = Dict({"foo": List(Int(max=100))})
>>> try:
...     schema({"foo": [1, 2, 200, 250], "bar": None})
... except exc.ValidationError as e:
...     error = e

>>> error.sort()
>>> error
<SchemaError(errors=[
    <bar: ForbiddenKeyError()>,
    <foo.2: MaxValueError(expected=100, actual=200)>,
    <foo.3: MaxValueError(expected=100, actual=250)>
])>

>>> len(error)
3

>>> error[1]
<foo.2: MaxValueError(expected=100, actual=200)>
>>> error[1].context
deque(['foo', 2])
>>> error[1].format_context()
'foo.2'
>>> error[1].format_error()
'MaxValueError(expected=100, actual=200)'

>>> error.sort(reverse=True)
>>> error
<SchemaError(errors=[
    <foo.3: MaxValueError(expected=100, actual=250)>,
    <foo.2: MaxValueError(expected=100, actual=200)>,
    <bar: ForbiddenKeyError()>
])>
```

add_context(*node*)

Add error context

Parameters **node** – key or index of member, where error is raised.

Returns the error itself, so that the method is suitable for chaining.

Example:

```
>>> from validx.exc import ValidationError

>>> e = ValidationError()
>>> e
<ValidationError()>
>>> e.context
deque([])

>>> e.add_context("foo")
<foo: ValidationError()>
>>> e.context
deque(['foo'])
```

class validx.exc.**ConditionError**(*context=None*, **args*, ***kw*)
Base Class for Condition Errors

It has a couple of attributes `expected` and `actual`, that gives info of what happens and why the error is raised.

See derived classes for details.

class validx.exc.**InvalidTypeError**(*context=None*, **args*, ***kw*)
Invalid Type Error

Parameters

- **expected** (*type or tuple*) – expected type (types).
- **actual** (*type*) – actual type of value.

class validx.exc.**OptionsError**(*context=None*, **args*, ***kw*)
Options Error

Parameters

- **expected** (*list or tuple*) – list of valid values.
- **actual** – actual value.

class validx.exc.**MinValueError**(*context=None*, **args*, ***kw*)
Minimum Value Error

Parameters

- **expected** – minimal allowed value.
- **actual** – actual value.

class validx.exc.**MaxValueError**(*context=None*, **args*, ***kw*)
Maximum Value Error

Parameters

- **expected** – maximal allowed value.
- **actual** – actual value.

class validx.exc.**FloatValueError**(*context=None*, **args*, ***kw*)
Float Value Error

Parameters

- **expected** (*str*) –
  - `"number"` on test for `Not-a-Number`;

  - `"finite"` on test for `Infinity`.

- **actual** (*float*) – actual value.

class validx.exc.**StrDecodeError**(*context=None*, **args*, ***kw*)
String Decode Error

Parameters

- **expected** (*str*) – encoding name.
- **actual** (*bytes*) – actual byte-string value.

class validx.exc.**MinLengthError**(*context=None*, **args*, ***kw*)
Minimum Length Error

Parameters

- **expected** (*int*) – minimal allowed length.
- **actual** (*int*) – actual value length.

class validx.exc.**MaxLengthError**(*context=None*, **args*, ***kw*)
Maximum Length Error

Parameters

- **expected** (*int*) – maximal allowed length.
- **actual** (*int*) – actual value length.

class validx.exc.**TupleLengthError**(*context=None*, **args*, ***kw*)
Tuple Length Error

Parameters

- **expected** (*int*) – tuple length.
- **actual** (*int*) – actual value length.

class validx.exc.**PatternMatchError**(*context=None*, **args*, ***kw*)
Pattern Match Error

Parameters

- **expected** (*str*) – pattern, i.e. regular expression.
- **actual** (*str*) – actual value.

class validx.exc.**DatetimeParseError**(*context=None*, **args*, ***kw*)
Date & Time Parse Error

Parameters

- **expected** (*str*) – format.
- **actual** (*str*) – actual value.

class validx.exc.**DatetimeTypeError**(*context=None*, **args*, ***kw*)
Date & Time Type Error

Parameters

- **expected** (*str*) – expected type of datetime: “naive” or “tzaware”.
- **actual** (*datetime*) – actual value.

class validx.exc.**RecursionMaxDepthError**(*context=None*, **args*, ***kw*)
Recursion Maximum Depth Error

Parameters

- **expected** (*int*) – maximal allowed depth.
- **actual** (*int*) – actual recursion depth.

class validx.exc.**MappingKeyError**(*context=None*, *key=None*)
Base Class for Mapping Key Errors

Parameters **key** – failed key, that goes into error context.

```
>>> e = MappingKeyError("foo")
>>> e
<foo: MappingKeyError()>
```

class validx.exc.**ForbiddenKeyError**(*context=None*, *key=None*)
Forbidden Mapping Key Error

class validx.exc.**MissingKeyError**(*context=None*, *key=None*)
Missing Mapping Key Error

class validx.exc.**SchemaError**(*context=None*, *errors=None*)
Schema Error

It is an error class, that wraps multiple errors occurred during complex structure validation.

Parameters **errors** (*list*) – list of all errors occurred during complex structure validation.

2.12 Context Markers

class validx.exc.**Extra**(*name*)
Extra Key Context Marker

It is a special context marker, that is used by mapping validators to indicate, which part of extra key/value pair is failed.

There are two constants in the module:

- `EXTRA_KEY` indicates that key validation is failed;
- `EXTRA_VALUE` indicates that value validation is failed.

It has special representation, to be easily distinguished from other string keys.

Parameters **name** (*str*) – name of pair part, i.e. `KEY` or `VALUE`.

```
>>> from validx import exc, Dict, Str

>>> schema = Dict(extra=(Str(maxlen=2), Str(maxlen=4)))
>>> try:
...     schema({"xy": "abc", "xyz": "abcde"})
... except exc.ValidationError as e:
...     error = e

>>> error
<SchemaError(errors=[
    <xyz.@KEY: MaxLengthError(expected=2, actual=3)>,
    <xyz.@VALUE: MaxLengthError(expected=4, actual=5)>
])>
```

(continues on next page)

(continued from previous page)

```
>>> repr(error[0].context[1])
'@KEY'
>>> error[0].context[1].name
'KEY'

>>> error[0].context[1] is exc.EXTRA_KEY
True
>>> error[1].context[1] is exc.EXTRA_VALUE
True
```

class validx.exc.**Step**(*num*)

Step Number Context Marker

It is a special context marker, that is used by pipeline validators to indicate, which validation step is failed. It has special representation, to be easily distinguished from sequence indexes.

Parameters **num** (*int*) – number of failed step.

```
>>> from validx import exc, OneOf, Int

>>> schema = OneOf(Int(min=0, max=10), Int(min=90, max=100))
>>> try:
...     schema(50)
... except exc.ValidationError as e:
...     error = e

>>> error
<SchemaError(errors=[
    <#0: MaxValueError(expected=10, actual=50)>,
    <#1: MinValueError(expected=90, actual=50)>
])>

>>> repr(error[0].context[0])
'#0'
>>> error[0].context[0].num
0
>>> isinstance(error[0].context[0], exc.Step)
True
```

2.13 Error Formatter

class validx.exc.**Formatter**(*templates*)

Error Formatter

Parameters **templates** (*dict*) – templates that will be used to format errors.

Each key of `templates` should be a subclass of *validx.exc.ValidationError*.

Each value of `templates` should be a string, i.e. simple template, or list of conditional templates.

Conditional template is a tuple `(predicate, string)`. Where `predicate` is a callable, that accepts *validx.exc.ValidationError* and returns boolean value. When the predicate evaluates to `True`, its corresponding string will be used as a template.

Last value of list of conditional templates can be a string, i.e. default simple template.

See `format_error` object, defined within the module, as an example.

__call__(*error*)
Format Error

Parameters **error** (ValidationError) – error to format.

Returns list of context/message pairs: `[(str, str), ...]`.

CHAPTER 3

Benchmarks

When ValidX had been released, it was the fastest validation library among the following competitors. It isn't true anymore. However, it would be unfair to remove this section from the documentation. Once the challenge has been thrown down, the competition must go on.

- Cerberus 1.3.2 ~130x slower
- Colander 1.7.0 ~2.8x slower
- JSONSchema 3.2.0 ~11x slower
- Marshmallow 3.6.0 ~11x slower
- Schema 0.7.2 ~29x slower
- Valideer 0.4.2 ~2x slower, but it has compatible performance with pure-Python implementation of ValidX. In some rounds it is a bit faster, in other is a bit slower.
- Validr 1.2.0 has compatible performance with Cython implementation of ValidX. In some rounds it is a bit faster, in other is a bit slower.
- Voluptuous 0.11.7 ~3x slower

Use the following command to run benchmarks:

```
tox -ebm
```

I got the following results on my laptop:

- CPU Intel i5-5200U 2.20GHz
- RAM 8GB
- OS Xubuntu 18.04, Linux core 4.15.0-101-generic
- Python 3.8.2

```
---------------------------------------------------- benchmark: 10 tests ------------
↪-----------------------------------------
Name (time in us)           Min                    Max                Mean            ␣
↪ StdDev            OPS (Kops/s)
```

(continues on next page)

(continued from previous page)

```
-------------------------------------------------------------------------------------------
↪-----------------------------------------
test_validx_cy           5.8240 (1.0)          41.5310 (1.0)         6.1031 (1.0)       ␣
↪ 0.9631 (1.0)          163.8516 (1.0)
test_validr              6.6680 (1.14)        191.4440 (4.61)        7.0912 (1.16)      ␣
↪ 1.7589 (1.83)         141.0205 (0.86)
test_validx_py          12.1390 (2.08)         63.1280 (1.52)       12.5840 (2.06)      ␣
↪ 1.5104 (1.57)          79.4661 (0.48)
test_valideer           12.9530 (2.22)        101.5800 (2.45)       13.4117 (2.20)      ␣
↪ 1.4033 (1.46)          74.5618 (0.46)
test_colander           16.6500 (2.86)         94.6480 (2.28)       17.3843 (2.85)      ␣
↪ 2.1166 (2.20)          57.5232 (0.35)
test_voluptuous         18.4060 (3.16)         69.2420 (1.67)       19.2751 (3.16)      ␣
↪ 2.3339 (2.42)          51.8804 (0.32)
test_marshmallow        67.0080 (11.51)       291.5290 (7.02)       69.9401 (11.46)     ␣
↪ 8.1468 (8.46)          14.2980 (0.09)
test_jsonschema         70.4520 (12.10)       242.8030 (5.85)       73.0181 (11.96)     ␣
↪ 7.5669 (7.86)          13.6952 (0.08)
test_schema            171.8870 (29.51)       332.5750 (8.01)      177.0150 (29.00)    ␣
↪10.8829 (11.30)          5.6492 (0.03)
test_cerberus          725.5970 (124.59)   11,228.9250 (270.37)   801.7096 (131.36)  ␣
↪561.3559 (582.86)        1.2473 (0.01)
-------------------------------------------------------------------------------------------
↪-----------------------------------------
```

3.1 Why you should care about performance

Note: I got tired to update the numbers in this section on each release. So I decided to give up. Let it be as it is. The numbers here are outdated and not based on the benchmark above anymore. But it doesn't change the main point — performance is important.

I have been asked by my colleagues: "Why should we care about performance? Data validation is not a bottleneck usually." And it is correct. But let's look on it from other side.

Let's say you have a web application that uses Cerberus for data validation, because Cerberus is the number one in 7 Best Python Libraries for Validating Data. How much will you earn replacing Cerberus by ValidX?

According to the benchmark above Cerberus spends 808 μs for each request, while ValidX only 2 μs. So that you will save 806 μs for each request. How much is it?

If you have a small webserver that takes about 200 requests per second (I took the number from this discussion on Stack Overflow), you will save:

```
806 µs × 200 × 60 × 60 × 24 = 13927.68 s/day
13927.68 ÷ 60 ÷ 60 = 3.8688 h/day
```

Yes, you will save almost 4 hours of server time daily, or almost 5 days monthly! It is about $5 monthly for each general purpose `t3.medium` instance on AWS, which costs $0.0416 per hour.

And now it is time to look at your logs, calculate number of requests you got in the last month, and compare it with a bill from your hosting provider.

CHAPTER 4

Internals

The following document is intended to be used by ValidX developers only.

4.1 Contracts

Contracts are used to validate parameters of validator itself during its initialization.

validx.contracts.**expect**(*obj*, *attr*, *value*, *nullable=False*, *types=None*, *not_types=None*, *convert_to=None*)

Check, whether the value satisfies expectations

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.
- **types** (*None, type or tuple*) – define acceptable types of the value. Default: `None` — accept any type.
- **not_types** – define implicitly unacceptable types of the value. Default: `None` — accept any type.
- **convert_to** (*type*) – convert the value to specified type. Default: `None` — does not convert the value.

Raises **TypeError** –

- if `types is not None` and `not isinstance(value, types)`;
- if `not_types is not None` and `isinstance(value, not_types)`.

validx.contracts.**expect_flag**(*obj*, *attr*, *value*)
Check, whether the value satisfies expectations of boolean flag

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.

Raises **TypeError** – if `not isinstance(value, (bool, int, type(None)))`.

validx.contracts.**expect_length**(*obj*, *attr*, *value*, *nullable=False*)
Check, whether the value satisfies expectations of integer length

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.

Raises

- **TypeError** – if `not isinstance(value, int)`.
- **ValueError** – if `value < 0`.

validx.contracts.**expect_basestr**(*obj*, *attr*, *value*, *nullable=False*)
Check, whether the value satisfies expectations of base string

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.

Raises **TypeError** –

- if `not isinstance(value, str)` (Python 3.x);
- if `not isinstance(value, basestring)` (Python 2.x).

validx.contracts.**expect_callable**(*obj*, *attr*, *value*, *nullable=False*)
Check, whether the value satisfies expectations of callable

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.

Raises **TypeError** – if `not isinstance(value, collections.abc.Callable)`.

validx.contracts.**expect_container**(*obj*, *attr*, *value*, *nullable=False*, *empty=False*, *item_type=None*)

Check, whether the value satisfies expectations of container

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.
- **empty** (*bool*) – accept empty container as a valid value. Default: `False` — does not accept empty container.
- **item_type** (*type*) – check, whether each item of the container has specified type. Default: `None` — does not check items.

Raises

- **TypeError** –
  - if `not isinstance(value, collections.abc.Container)`;
  - if `isinstance(value, (str, bytes))` (Python 3.x);
  - if `isinstance(value, (unicode, bytes))` (Python 2.x);
  - if `item_type is not None` and `isinstance(item, item_type), for item in value`.
- **ValueError** – if `not empty` and `not value`.

Returns passed container converted to `frozenset`, if items are hashable, otherwise to `tuple`.

validx.contracts.**expect_sequence**(*obj*, *attr*, *value*, *nullable=False*, *empty=False*, *item_type=None*)

Check, whether the value satisfies expectations of sequence

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.

- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.
- **empty** (*bool*) – accept empty sequence as a valid value. Default: `False` — does not accept empty sequence.
- **item_type** (*type*) – check, whether each item of the sequence has specified type. Default: `None` — does not check items.

Raises

- **TypeError** –
  - if `not isinstance(value, collections.abc.Sequence)`;
  - if `isinstance(value, (str, bytes))` (Python 3.x);
  - if `isinstance(value, (unicode, bytes))` (Python 2.x);
  - if `item_type is not None` and `isinstance(item, item_type), for item in value`.
- **ValueError** – if `not empty` and `not value`.

Returns passed sequence converted to `tuple`.

`validx.contracts.`**expect_mapping**(*obj*, *attr*, *value*, *nullable=False*, *empty=False*, *value_type=None*)

Check, whether the value satisfies expectations of mapping

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.
- **empty** (*bool*) – accept empty mapping as a valid value. Default: `False` — does not accept empty mapping.
- **value_type** (*type*) – check, whether each value of the mapping has specified type. Default: `None` — does not check items.

Raises

- **TypeError** –
  - if `not isinstance(value, collections.abc.Sequence)`;
  - if `isinstance(value, (str, bytes))` (Python 3.x);
  - if `isinstance(value, (unicode, bytes))` (Python 2.x);
  - if `value_type is not None` and `isinstance(val, value_type), for key, val in value.items()`.
- **ValueError** – if `not empty` and `not value`.

Returns passed mapping converted to `frozendict`.

`validx.contracts.`**expect_tuple**(*obj*, *attr*, *value*, *struct*, *nullable=False*)

Check, whether the value satisfies expectations of tuple of specific structure

Parameters

- **obj** – an object, which will set the value to its attribute. It is used to make error messages more specific.
- **attr** (*str*) – name of an attribute of the object. It is used to make error messages more specific.
- **value** – checked value itself.
- **struct** (*tuple*) – tuple of types.
- **nullable** (*bool*) – accept `None` as a valid value. Default: `False` — does not accept `None`.

Raises

- **TypeError** –
  - if `not isinstance(value, collections.abc.Sequence)`;
  - if `isinstance(value, (str, bytes))` (Python 3.x);
  - if `isinstance(value, (unicode, bytes))` (Python 2.x);
  - if `not isinstance(item, item_type), for item_type, item in zip(struct, value)`.
- **ValueError** – if `len(value) != len(struct)`.

Returns passed sequence converted to `tuple`.

CHAPTER 5

Changes

5.1 0.6.1

- Fixed type declarations for `validx.py.Validator.clone()` method.

5.2 0.6

- Added Python 3.8 into test matrix.
- Made validators immutable.
- Added contracts checks on validator initialization.
- Added new simplified syntax for *Cloning Validators*.
- Got rid of global state within `validx.py.LazyRef` validator. It now acts like a pure function.
- Fixed raising of ambiguous `validx.exc.MinLengthError` on `validx.py.List` and `validx.py.Dict` validation.

5.3 0.5.1

- Fixed type declarations. Again. One does not simply make mypy happy.

5.4 0.5

- Removed confusing nullable check from `validx.py.Any` validator.
- Fixed type declarations.

5.5 0.4

- Fixed library objects pickling.
- Fixed checking of length within *validx.py.List* validator.

5.6 0.3

- Fixed handling of default values and length validation within *validx.py.Dict* validator.

5.7 0.2

- Added support of timezones into *validx.py.Date* and *validx.py.Datetime* validators.
- Added support of custom parsers into *validx.py.Date*, *validx.py.Time*, and *validx.py.Datetime* validators.
- Added *validx.py.Type* validator for custom types.

5.8 0.1

- Initial release.

CHAPTER 6

Contribution

The project sources are hosted on GitHub as well, as its bug tracker. Pull requests, bug reports, and feedback are welcome.

CHAPTER 7

License

The code is licensed under the terms of BSD 2-Clause license. The full text of the license can be found at the root of the sources.

CHAPTER 8

Indices and tables

- genindex
- search

Index

Symbols

A

B

C

D

E

F

G

I

L

M

O

P

R

S

T

V