Trafaret Documentation

Release 2.1.0

Mikhail Krivushin

November 26, 2020

Table of Contents

1 Installation **3**
- 1.1 Stable release . . . 3
- 1.2 From sources . . . 3

2 Introducing **5**
- 2.1 Types . . . 5
- 2.2 Atom . . . 14
- 2.3 Operations . . . 17
- 2.4 Other . . . 18

3 Errors **21**
- 3.1 DataError . . . 21

4 Web use **23**
- 4.1 Introduction . . . 23
- 4.2 Schemes . . . 24
- 4.3 Handlers . . . 25
- 4.4 Errors . . . 26

5 API **29**
- 5.1 Trafaret . . . 29
- 5.2 Subclassing . . . 29

6 API docs **31**
- 6.1 `trafaret` — Validation atoms definition . . . 31
- 6.2 `trafaret.keys` — custom Dict keys implementations . . . 41
- 6.3 `trafaret.utils` — utils for unfolding netsted dict syntax . . . 42
- 6.4 `trafaret.constructor` — methods to access object's attribute/netsted key by path . . . 42

7 Changelog **43**

8 2.1.0 **45**

9 2.0.2 **47**
- 9.1 2.0.0 . . . 47
- 9.2 1.0.3 . . . 47
- 9.3 1.0.1 . . . 48
- 9.4 1.0.0 . . . 48

9.5 2017-08-04 . . . 48
9.6 2017-05-12 . . . 48
9.7 2017-03-25 0.9.0 . . . 48
9.8 0.8.1 . . . 48
9.9 2016-09-25 . . . 49
9.10 2016-08-03 . . . 49
9.11 2016-03-31 . . . 49
9.12 2016-03-18 . . . 49
9.13 2014-09-17 . . . 49
9.14 2012-05-30 . . . 49
9.15 2012-05-28 . . . 49
9.16 2012-05-21 . . . 49
9.17 2012-05-16 . . . 49
9.18 2012-05-11 . . . 50
9.19 2012-05-10 . . . 50
9.20 2012-04-12 . . . 50

10 Indices and tables **51**

Python Module Index **53**

Index **55**

Contents:

CHAPTER 1

Installation

1.1 Stable release

To install trafaret, run this command in your terminal:

```
$ pip install trafaret
```

This is the preferred method to install trafaret, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for trafaret can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/Deepwalker/trafaret
```

Or download the tarball:

```
$ curl  -OL https://github.com/Deepwalker/trafaret/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 2

Introducing

Trafaret is validation library with support to convert data structures. Sample usage:

```
import datetime
import trafaret as t

date = t.Dict(year=t.Int, month=t.Int, day=t.Int) & (lambda d: datetime.datetime(**d))

def validate_date(data):
    try:
        return date.check(data), False
    except t.DataError as e:
        return False, e.as_dict()

validate_date({'year': 2012, 'month': 1})
# (False, {'day': 'is required'})

validate_date({'year': 2012, 'month': 1, 'day': 12})
# (datetime.datetime(2012, 1, 12, 0, 0), False)
```

`t.Dict` creates new dict structure validator with three `t.Int` elements. `&` operation combines trafaret with other trafaret or with a function.

2.1 Types

2.1.1 String

The `String` is base checker in trafaret which just test that value is string. Also `String` has a lot of helpful modification like `Email` and `Url`.

```
t.String().check('this is my string')
# 'this is my string'
```

Options:

- **allow_blank** *(boolean)* - indicates if string can be blank or not
- **min_length** *(integer)* - validation for minimum length of receive string
- **max_length** *(integer)* - validation for maximum length of receive string

The simple examples of usage:

```
t.String(allow_blank=True).check('')
# ''
t.String(min_length=1, max_length=10).check('no so long')
# 'no so long'
```

`Email` and `URL` just provide regular expressions and a bit of logic for IDNA domains. Default converters return email and domain, but you will get re match object in converter.

Email

This checker test that a received string is an valid email address.

```
t.Email.check('someone@example.net')
# 'someone@example.net'
```

URL

This checker test that a received string is an valid URL address. This URL can include get params and anchors.

```
t.URL.check('http://example.net/resource/?param=value#anchor')
# 'http://example.net/resource/?param=value#anchor'
```

IPv4

This checker test that a received string is IPv4 address.

```
t.IPv4.check('127.0.0.1')
# '127.0.0.1'
```

IPv6

This checker test that a received string is IPv6 address.

```
t.IPv6.check('2001:0db8:0000:0042:0000:8a2e:0370:7334')
# '2001:0db8:0000:0042:0000:8a2e:0370:7334'
```

IP

This checker test that a received string is IP address (IPv4 or IPv6).

```
t.IP.check('127.0.0.1')
# '127.0.0.1'
t.IP.check('2001:0db8:0000:0042:0000:8a2e:0370:7334')
# '2001:0db8:0000:0042:0000:8a2e:0370:7334'
```

2.1.2 Regexp

The checker test that a received string match with given regexp. With this Regexp you can write you own checker like Email or URL.

Note that this uses *re.match()* and will return the matching group at the start of the string. If you want to ensure a full match ensure you add *$* to the end of the expression.

```
t.Regexp(regexp=r"\w{3}-\w{3}-\w{4}").check('544-343-7564')
# '544-343-7564'
```

2.1.3 RegexpRaw

With this checker you can use all `re.match` power to extract from strings dicts and other higher level datastructures.

```
name_checker = t.RegexpRaw(r'^name=(\w+)$') >> (lambda m: m.groups()[0])
name_checker.check('name=Jeff')
# 'Jeff'
```

or more interesting example:

```
from datetime import datetime

def to_datetime(m):
   return datetime(*[int(i) for i in m.groups()])

date_checker = t.RegexpRaw(regexp='^year=(\d+), month=(\d+), day=(\d+)$') & to_
↪datetime

date_checker.check('year=2019, month=07, day=23')
# datetime.datetime(2019, 7, 23, 0, 0)
```

2.1.4 Bytes

Also if you want to check, is value bytes string or no you can use this checker.

```
t.Bytes().check(b'bytes string')
```

2.1.5 AnyString

If you need to check value which can be string or bytes string, you can use `AnyString`.

```
for item in ['string', b'bytes string']:
    print(t.AnyString().check(item))
```

(continues on next page)

(continued from previous page)

```
# string
# b'bytes string'
```

2.1.6 FromBytes

If you need to convert bytestring to `utf-8` or to the other standard you can use this checker. If receive value can't be converted to standard then trafaret raise an error. This often can be useful when receive value can be a `string` or a `bytestring`.

```
unicode_or_utf16 = t.String | t.FromBytes(encoding='utf-16')

unicode_or_utf16.check(b'\xff\xfet\x00r\x00a\x00f\x00a\x00r\x00e\x00t\x00')
# 'trafaret'

unicode_or_utf16.check('trafaret')
# 'trafaret'
```

The default encoding is `utf-8`.

```
t.FromBytes().check(b'trafaret')
# 'trafaret'
```

2.1.7 Dict and Keys

The `Dict` checker is needed to validate a dictionaries. For use `Dict` you need to describe your dictionary as dictionary where instead of values are checkers of this values.

```
login_validator = t.Dict({'username': t.String(max_length=10), 'email': t.Email})
login_validator.check({'username': 'Misha', 'email': 'misha@gmail.com'})
# {'username': 'Misha', 'email': 'misha@gmail.com'}
```

`Dict` has a lot of helpful methods:

- `allow_extra` - when you need to validate only a part of keys you can use allow_extra to allow to do that:

```
data = {'username': 'Misha', 'age': 12, 'email': 'm@gmail.com', 'is_superuser
↪': True}

user_validator = t.Dict({'username': t.String, 'age': t.Int})

# generate a new checker with allow any extra keys
new_user_validator = user_validator.allow_extra('*')
new_user_validator.check(data)
# {'username': 'Misha', 'age': 12, 'email': 'm@gmail.com', 'is_superuser':␣
↪True}
```

Also if you want to allow only some concretical kyes you cat set them:

```
user_validator.allow_extra('email', 'is_superuser')
```

If when you need to specify type of extra keys you can use `trafaret` keyword argument for that *(by default trafaret is Any)*:

```
user_validator.allow_extra('email', 'is_superuser', trafaret=t.String)
```

Also you can specify extra keys when you create your `Dict` checker:

```
user_validator = t.Dict({'username': t.String, 'age': t.Int}, allow_extra=['*
↪'])
```

- `ignore_extra` - when you need to remove nececary data from result you can use it. This method has similar signature like in `allow_extra`.

```
data = {'username': 'Misha', 'age': 12, 'email': 'm@gmail.com', 'is_superuser
↪': True}

user_validator = t.Dict({'username': t.String, 'age': t.Int}).ignore_extra('*
↪')
user_validator.check(data)
# {'username': 'Misha', 'age': 12}
```

- `merge` - where argument can be other `Dict`, dict like provided to `Dict`, or list of `Key` s. Also provided as __add__, so you can add `Dict` s, like `dict1 + dict2`.

  This can be so useful when you have two large dictionaries with so similar structure. As example it possible when you do validation for create and update some instance whan for create instance you don't need *id* but for update do.

```
user_create_validator = t.Dict({'username': t.String, 'age': t.Int})

user_update_validator = user_create_validator + {'id': t.Int}
user_update_validator.check({'username': 'misha', 'age': 12, 'id': 1})
# {'username': 'misha', 'age': 12, 'id': 1}
```

Some time we need to change name of key in initial dictionary. For that trafaret provides `Key`. This can be very useful. As example when you receive form from frontend with keys in camel case and you want to convert this keys to snake case.

```
login_validator = t.Dict({t.Key('userName') >> 'user_name': t.String})
login_validator.check({'userName': 'Misha'})
# {'user_name': 'Misha'}
```

Also we can to receive input data like this:

```
data = {"title": "Glue", "authorFirstName": "Irvine", "authorLastName": "Welsh"}
```

and want to split data which connected with author and book. For that we can use `fold`.

```
from trafaret.utils import fold

book_validator = t.Dict({
    "title": t.String,
    t.Key('authorFirstName') >> 'author__first_name': t.String,
    t.Key('authorLastName') >> 'author__last_name': t.String,
}) >> fold

book_validator.check(data)
# {'author': {'first_name': 'Irvine', 'last_name': 'Welsh'}, 'title': 'Glue'}
```

Key

Special class to create dict keys. Parameters are:

- *name* - key name
- *default* - default if key is not present
- *optional* - if True the key is optional
- *to_name* - allows to rename the key

Below you can to see a good example of usage all of these parameters:

```
import hashlib

hash_md5 = lambda d: hashlib.md5(d.encode()).hexdigest()
comma_to_list = lambda d: [s.strip() for s in d.split(',')]

converter = t.Dict({
   t.Key('userNameFirst') >> 'name': t.String,
   t.Key('userNameSecond') >> 'second_name': t.String,
   t.Key('userPassword') >> 'password': hash_md5,
   t.Key('userEmail', optional=True, to_name='email'): t.String,
   t.Key('userTitle', default='Bachelor', to_name='title'): t.String,
   t.Key('userRoles', to_name='roles'): comma_to_list,
})
```

We can rewrite it to:

```
converter = t.Dict(
   t.Key('userNameFirst', to_name='name', trafaret=t.String),
   t.Key('userNameSecond', to_name='second_name', trafaret=t.String),
   t.Key('userPassword', to_name='password', trafaret=hash_md5),
   t.Key('userEmail', optional=True, to_name='email', trafaret=t.String),
   t.Key('userTitle', default='Bachelor', to_name='title', trafaret=t.String),
   t.Key('userRoles', to_name='roles', trafaret=comma_to_list),
)
```

It provides method `__call__(self, data)` that extract key value from data through mapping `get` method. Key `__call__` method yields `(key name, Maybe(DataError), [touched keys])` triples. You can redefine `get_data(self, data, default)` method in subclassed `Key` if you want to use something other then `.get(...)` method. Like this for the aiohttp's *MultiDict* class:

```
class MDKey(t.Key):
    def get_data(data, default):
        return data.get_all(self.name, default)

t.Dict({MDKey('users'): t.List(t.String)})
```

Moreover, instead of `Key` you can use any callable, say a function:

```
def simple_key(value):
    yield 'simple', 'simple data', []

check_args = t.Dict(simple_key)
```

DictKeys

If you need to check just that dictionary has all given keys so `DictKeys` is a good approach for that.

```
t.DictKeys(['a', 'b']).check({'a': 1, 'b': 2})
# {'a': 1, 'b': 2}
```

KeysSubset

We have some example of enhanced `Key` in extras:

```
from trafaret.extras import KeysSubset

cmp_pwds = lambda x: {
    'pwd': x['pwd'] if x.get('pwd') == x.get('pwd1') else DataError('Not equal')
}

d = Dict({KeysSubset('pwd', 'pwd1'): cmp_pwds, 'key1': String})

d.check({'pwd': 'a', 'pwd1': 'a', 'key1': 'b'}).keys()
# {'pwd': 'a', 'key1': 'b'}
```

2.1.8 Mapping

This checker test that a received dictionary has current types of keys and values.

```
t.Mapping(t.String, t.Int).check({"foo": 1, "bar": 2})
# {'foo': 1, 'bar': 2}
```

Where a first argument is a type of keys and second is type of values.

2.1.9 Bool

The checker test that a received value is a boolean type.

```
t.Bool().check(True)
# True
```

2.1.10 ToBool

If you need to check value that can be equivalent to a boolean type, you can use `ToBool`. **Letter case doesn't matter.**

Sample with all supported equivalents:

```
equivalents  = ('t', 'true', 'y', 'yes', 'on', '1', '1.0',\
                'false', 'n', 'no', 'off', '0', '0.0', 'none')

for value in equivalents:
  print("%s is %s" % (value, t.ToBool().check(value)))

# t is True
# true is True
```

(continues on next page)

(continued from previous page)

```
# y is True
# yes is True
# on is True
# 1 is True
# 1.0 is True
# false is False
# n is False
# no is False
# off is False
# 0 is False
# 0.0 is False
# none is False
```

Also, function can take `1` and `0` as integers, `booleans` and `None`.

```
t.ToBool().check(1)
# True

t.ToBool().check(False)
# False

t.ToBool().check(None)
# False
```

2.1.11 Float

Check if value is a float or can be converted to a float. Supports `lte`, `gte`, `lt`, `gt` parameters, <=, >=, <, > operators and `Float[0:10]` slice notation:

```
t.Float(gt=3.5).check(4)
# 4

(t.Float >= 3.5).check(4)
# 4

t.Float[3.5:].check(4)
# 4
```

2.1.12 ToFloat

Similar to `Float`, but converting to `float`:

```
t.ToFloat(gte=3.5).check(4)
# 4.0
```

2.1.13 ToDecimal

Similar to `ToFloat`, but converting to `Decimal`:

```
from decimal import Decimal, ROUND_HALF_UP
import trafaret as t
```

(continues on next page)

(continued from previous page)

```
validator = t.Dict({
    "name": t.String,
    "salary": t.ToDecimal(gt=0) & (
        lambda value: value.quantize(
                Decimal('.0000'), rounding=ROUND_HALF_UP
            )
    ),
})

validator.check({"name": "Bob", "salary": "1000.0"})
# {'name': 'Bob', 'salary': Decimal('1000.0000')}

validator.check({"name": "Tom", "salary": 1000.0005})
# {'name': 'Tom', 'salary': Decimal('1000.0005')}

validator.check({"name": "Jay", "salary": 1000.00049})
# {'name': 'Jay', 'salary': Decimal('1000.0005')}

validator.check({"name": "Joe", "salary": -1000})
# DataError: {'salary': DataError('value should be greater than 0')}
```

2.1.14 Int

Similar to `Float`, but checking for `int`:

```
t.Int(gt=3).check(4)
# 4
```

2.1.15 ToInt

Similar to `Int`, but converting to `int`:

```
import trafaret as t
from yarl import URL

query_validator = t.Dict({
    t.Key('node', default=0): t.ToInt(gte=0),
})

url = URL('https://www.amazon.com/b?node=18637575011')
query_validator.check(url.query)
# {'node': 18637575011}

url = URL('https://www.amazon.com/b')
query_validator.check(url.query)
# {'node': 0}

url = URL('https://www.amazon.com/b?node=-10')
query_validator.check(url.query)
# DataError: {'node': DataError('value is less than 0')}
```

2.1.16 Null

This checker test that a received value is `None`. This checker is very useful together with other checkers when you need to test that receive value has some type or `None`.

```
(t.Int | t.Null).check(5)
# 5

(t.Int | t.Null).check(None)
# None
```

2.1.17 Any

This checker doesn't check anything. This is very often use in `Dict` to test that some key exists in the dictionary, but doesn't care what type it is.

```
t.Dict({"value": t.Any}).check({"value": "123"})
# {'value': '123'}
```

This is the same with `allow_extra` method in `Dict`.

2.1.18 Type

Checks that data is instance of given class. Just instantiate it with any class, like int, float, str. For instance:

```
t.Type(int).check(4)
# 4
```

2.2 Atom

This checker test that a received value is equal with first argument.

```
t.Atom('this_key_must_be_this').check('this_key_must_be_this')
# 'this_key_must_be_this'
```

This may be useful in `Dict` with `Or` statements to create enumerations.

2.2.1 Date

Check that argument is an instance of `datetime.date` object:

```
>>> t.Date().check("2019-07-25")
'2019-07-25'
>>> t.Date().check(date.today())
datetime.date('2019-07-25')
```

You can easily specify the format for `Date` trafaret:

```
>>> t.Date(format='%y-%m-%d')
'<Date %y-%m-%d>'
>>> t.Date(format='%y-%m-%d').check('00-01-01')
'00-01-01'
```

2.2.2 ToDate

Behave like `Date`, but also returns `datetime.date` object:

```
>>> t.ToDate().check("2019-07-25")
datetime.date('2019-07-25')
>>> t.ToDate().check(datetime.now())
datetime.date('2019-07-25')
```

2.2.3 DateTime

Similar to `Date`, but checking for `datetime.datetime` object:

```
>>> DateTime('%Y-%m-%d %H:%M').check("2019-07-25 21:45")
'2019-07-25 21:45'
>>> t.extract_error(t.DateTime(), date.today())
'value cannot be converted to datetime'
```

2.2.4 ToDateTime

Behave like `DateTime`, but also returns `datetime.datetime` object:

```
>>> DateTime('%Y-%m-%d %H:%M').check("2019-07-25 21:45")
datetime.datetime(2019, 7, 25, 21, 45)
```

2.2.5 List

This checker test that a received value is a list of items with some type.

```
t.List(t.Int).check(range(100))
# [0, 1, 2, ... 99]

t.extract_error(t.List(t.Int).check(['a']))
# {0: DataError("value can't be converted to int")}
```

Also if an item has possible two or three types you can use `Or`.

```
t.List(t.ToInt | t.String).check(['1', 'test'])
# [1, 'test']
```

Options:

- **min_length** *(integer)* - validation for minimum length of receive list
- **max_length** *(integer)* - validation for maximum length of receive list

The simple examples of usage:

```
t.List(t.Int, min_length=1, max_length=2).check(['1', '2'])
# ['1', '2']
```

2.2.6 Iterable

This checker is the same with `List` but it don't raise error if received value isn't instance of a `list`.

```
my_data = (1, 2)

try:
    t.List(t.Int, min_length=1, max_length=2).check(my_data)
except t.DataError as e:
    print(e)
# value is not a list

t.Iterable(t.Int, max_length=2).check(my_data)
# [1, 2]
```

2.2.7 Tuple

This checker test that a received value is a tuple of items with some type.

```
t.Tuple(t.ToInt, t.ToInt, t.String).check([3, 4, u'5'])
# (3, 4, u'5')
```

2.2.8 Enum

This checker tests that given value is in the list of arguments passed to Enum. List of arguments can contain values of different types.

Example:

```
t.Enum(1, 2, 'error').check(2)
# 2
```

This checker can be used to validate user choice/input with predefined variants, for example defect severity in the bug tracking system.

Example:

```
user_choice = 'critical'
severities = ('trivial', 'minor', 'major', 'critical')

t.Enum(*severities).check(user_choice)
# 'critical'
```

2.2.9 Callable

This checker test that a received value is callable.

```
t.Callable().check(lambda: 1)
```

2.2.10 Call

This checker receive custom function for validation and convert value. If value is valid then function return converted value else raise `DataError`.

```
def validator(value):
    """The custom validation function."""
    if value != "foo":
        return t.DataError("I want only foo!", code='i_wanna_foo')
    return 'foo'

t.Call(validator).check('foo')
# 'foo'
```

2.3 Operations

2.3.1 Or

You can combine checkers and for that you need to use `Or`. `Or` takes other converters as arguments. The input is considered valid if one of the converters succeed:

```
Or(t.Int, t.String).check('1')
# 1
```

but the more popular way it is using `|`

```
(t.Int | t.String).check('five')
# 'five'
```

2.3.2 fold

We already talked about `fold` but let's see all features of this utils.

The parameters:

- *prefix* - the prefix which need to remove
- *delimeter* - the parameter which use for split to keys

The full example:

```
new_fold = lambda x: fold(x, 'data', '.')

book_validator = t.Dict({
    "data.author.first_name": t.String,
    "data.author.last_name": t.String,
}) >> new_fold

book_validator.check({
   "data.author.first_name": 'Irvine',
   "data.author.last_name": 'Welsh',
})
# {'author': {'first_name': 'Irvine', 'last_name': 'Welsh'}}
```

2.3.3 subdict

Very often when we do validation of the form we need to validate values which depend on each other. As example it can be *password* and *second_password*. For cases like this a trafaret has `subdict`.

```
from trafaret.keys import subdict

def check_passwords_equal(data):
    if data['password'] != data['password_confirm']:
        return t.DataError('Passwords are not equal')
    return data['password']

passwords_key = subdict(
    'password',
    t.Key('password', trafaret=t.String(max_length=10)),
    t.Key('password_confirm', trafaret=t.String(max_length=10)),
    trafaret=check_passwords_equal,
)

signup_trafaret = t.Dict(
    t.Key('email', trafaret=t.Email),
    passwords_key,
)

signup_trafaret.check({
    "email": "m@gmail.com",
    "password": "111",
    "password_confirm": "111",
})
# {'email': 'm@gmail.com', 'password': '111'}
```

As you can see, *password* and *password_confirm* replaced to just password with value that `check_passwords_equal` return.

2.4 Other

2.4.1 Forward

This checker is container for any checker, that you can provide later. To provide container use `provide` method or `&` operation:

```
node = t.Forward()
node & t.Dict(name=t.String, children=t.List[node])
```

2.4.2 guard

This is decorator for functions. You can validate and convert receive arguments.

```
@t.guard(user_name=t.String(max_length=10), age=t.ToInt, is_superuser=t.Bool)
def create_user(user_name, age, is_superuser=False):
   # do some stuff
   ...
   return (user_name, age, is_superuser)
```

(continues on next page)

(continued from previous page)

```
create_user('Misha', '12')
# ('Misha', 12, False)
# convert age to integer
```

GuardError

The *guard* raise `GuardError` error that base by `DataError`.

CHAPTER 3

Errors

3.1 DataError

Exception class that is used in the library. Exception hold errors in error attribute. For simple checkers it will be just a string. For nested structures it will be dict instance.

`DataError` instance has four important properties:

- *error* - error message describing what happened
- *code* - error code (this code you can use for replace an error message)
- *value* - raw value
- *trafaret* - checker instance which raised an error

`DataError` instance has two methods for represent full information about errors

- *as_dict* - the simple representation of errors as dictionary
- *to_struct* - more information than in *as_dict*. Here we can see code of error and other helpful information.

```
login_validator = t.Dict({'username': t.String(max_length=10), 'email': t.Email})

try:
    login_validator.check({'username': 'So loooong name', 'email': 'misha'})
except t.DataError as e:
    print(e.as_dict())
    print(e.to_struct())

# {
#    'username': 'String is longer than 10 characters',
#    'email': 'value is not a valid email address'
# }

# {
#    'code': 'some_elements_did_not_match',
```

(continues on next page)

(continued from previous page)

```
#     'nested': {
#         'username': {
#             'code': 'long_string',
#             'message': 'String is longer than 10 characters'
#         },
#         'email': {
#             'code': 'is_not_valid_email',
#             'message': 'value is not a valid email address'
#         }
#     }
# }
```

Also, as_dict and to_struct have optional parameter `value` that set to False as default. If set it to `True` trafaret will show bad value in error message.

CHAPTER 4

Web use

4.1 Introduction

Let's see a simple way to use `trafaret` for handle endpoints of rest api in `aiohttp`. We'll create a simple rest api for create / update books. The first thing which we need to do it's describe our schemes of input data in trafaret format.

When we work with form in the web we receive all data in a string format. So when you want to send a boolean type or list of integers you send some like this.

```
"True" # True
"1, 2, 3" # [1, 2, 3]
```

Trafaret designed for solve these problems.

```
import trafaret as t

def comma_to_list(text):
    """Convert string with words separated by comma to list."""
    return [
        s.strip() for s in text.split(',')
    ]

create_book_chacker = t.Dict({
   'title': t.String,
   'authors': comma_to_list,
   'sold': t.StrBool,
})
```

So, when you recive data from form it's not problem for you, because `StrBool` and `comma_to_list` prepare data for you in correct format.

```
create_book_chacker.check({"title": 'Glue', 'authors': 'Welsh,', 'sold': 'True'})
# {'title': 'Glue', 'authors': ['Welsh', ''], 'sold': True}
```

But if you receive data as `json` it's not very useful for you, because you can receive data in correct format from the client.

The second problem which trafaret solved it's a *camel/snake case war*. People why write in python prefe use *snake_case* unlike people why write in *ES* and use *CamelCase*. Trafaret give an approach for rename key of dictionary for solve this problem.

```
t.Dict({t.Key('userNameFirst') >> 'first_name': t.String})

# or

t.Dict({t.Key('userNameFirst', to_name='first_name'): t.String})
```

4.2 Schemes

So, now we are ready to write our schemes with *trafaret*. We can put this to the `utils.py`.

```
import trafaret as t

create_book_chacker = t.Dict({
   t.Key('bookTitle', to_name='title'): t.String,
   t.Key('bookPageCount', to_name='pages'): t.Int,
   t.Key('bookDescription', to_name='description'): t.String(min_length=20),
   t.Key('bookPrice', to_name='price', default=100): t.Int >= 100,
   t.Key('bookIsFree', optional=True, to_name='is_free'): t.Bool,
   t.Key('bookFirstAuthor', to_name='first_author'): t.String(max_length=10),
   t.Key('bookAuthors', to_name='authors'): t.List(t.String(max_length=10)),
})


update_user_chacker = create_book_chacker + {"id": t.Int}
```

Here we created a two schemes. For validate data which need to create a book and for update. This two schemes differing only by `id` field.

After that we can use this checkers for validation data in our web handlers. But for allocation all logic which connected with trafaret let's create functions which do it.

```
def prepare_data_for_create_book(data):
    valid_data = create_book_chacker.check(data)

    # do something else
    ...

    return valid_data

def prepare_data_for_update_book(data):
    valid_data = update_user_chacker.check(data)

    # do something else
    ...

    return valid_data
```

4.3 Handlers

Let's use these function in our handlers.

```
from aiohttp import web


# handlers

async def create_book(req):
    """Hadler for create book"""
    raw_data = await req.json()
    data = prepare_data_for_create_book(raw_data)

    # do something
    ...

    return web.json_response({"created": True})


async def update_book(req):
    """Handler for update book by id"""
    raw_data = await req.json()
    data = prepare_data_for_update_book(raw_data)

    # do something
    ...

    return web.json_response({"updated": True})


# setup an application

app = web.Application()
app.add_routes([
    web.post('/', create_book),
    web.put('/', update_book)
])

web.run_app(app, port=8000)
```

After that we can send request to the our server.

```
import requests as r


data = {
    "bookTitle": "Glue",
    "bookPageCount": 436,
    "bookDescription": "Glue tells the stories of four Scottish boys over four␣
↪decades...",
    "bookPrice": 423,
    "bookFirstAuthor": "Welsh",
    "bookAuthors": ["Welsh"]
}
r.post("http://0.0.0.0:8000/", json=data).text
```

(continues on next page)

(continued from previous page)

```
# '{"created": true}'
```

4.4 Errors

We made validation for input data but also we want eazy show errors if we have problem with it.

If input data is not valid then *trafaret* after call check method raise error (`t.DataError`) connected with that. Let's see easy way to handle all errors connected with `trafater`.

```
from functools import wraps


def with_error(fn):
    """
    This is decorator for wrapping web handlers which need to represent
    errors connected with validation if they exist.
    """

    @wraps(fn)
    async def inner(*args, **kwargs):
        try:
            return await fn(*args, **kwargs)
        except t.DataError as e:
            return web.json_response({
                'errors': e.as_dict(value=True)
            })

    return inner
```

After that we need to wrap all our handlers.

```
@with_error
async def create_book(req):
    """Hadler for create book"""
    ...

@with_error
async def update_book(req):
    """Handler for update book by id"""
    ...
```

That is it. Now, we receive pretty error messages when our input data is not valid.

```
import requests as r


data = {
    "bookTitle": "Glue",
    "bookPageCount": 436,
    "bookDescription": "Glue tells the stories of four Scottish boys over four␣
↪decades...",
    "bookPrice": 423,
    "bookFirstAuthor": "Welsh",
    "bookAuthors": ["Welsh"]
```

(continues on next page)

(continued from previous page)

```
}
r.put("http://0.0.0.0:8000/", json=data).text

# '{"errors": {"id": "is required"}}'
```

CHAPTER 5

API

5.1 Trafaret

Base class for checkers. Use it to create new checkers. In derived classes you need to implement *_check* or *_check_val* methods. *_check_val* must return a value, *_check* must return *None* on success.

You can implement *converter* method if you want to convert value somehow, that said you prolly want to make it possible for the developer to apply their own converters to raw data. This used to return strings instead of *re.Match* object in *String* trafaret.

5.2 Subclassing

For your own trafaret creation you need to subclass `Trafaret` class and implement `check_value` or `check_and_return` methods. `check_value` can return nothing on success, `check_and_return` must return value. In case of failure you need to raise `DataError`. You can use `self._failure` shortcut function to do this. Check library code for samples.

CHAPTER 6

API docs

6.1 `trafaret` — Validation atoms definition

exception trafaret.**DataError**(*error=None*, *name=None*, *value=<object object>*, *trafaret=None*, *code=None*)

Error with data preserve error can be a message or None if error raised in childs data can be anything

as_dict(*value=False*)

Use *to_struct* if need consistency

class trafaret.**Trafaret**

Base class for trafarets, provides only one method for trafaret validation failure reporting

check(*value*, *context=None*)

Common logic. In subclasses you need to implement check_value or check_and_return.

is_valid(*value*)

Allows to check value and get bool, is value valid or not.

class trafaret.**Call**(*fn*)

```
>>> def validator(value):
...     if value != "foo":
...         return DataError("I want only foo!")
...     return 'foo'
...
>>> trafaret = Call(validator)
>>> trafaret
<Call(validator)>
>>> trafaret.check("foo")
'foo'
>>> extract_error(trafaret, "bar")
'I want only foo!'
```

class trafaret.**Or**(**trafarets*)

```
>>> nullString = Or(String, Null)
>>> nullString
<Or(<String>, <Null>)>
>>> nullString.check(None)
>>> nullString.check("test")
'test'
>>> extract_error(nullString, 1)
{0: 'value is not a string', 1: 'value should be None'}
>>> nullString.is_valid(None)
True
>>> nullString.is_valid("test")
True
>>> nullString.is_valid(1)
False
```

class trafaret.**And**(*trafaret*, *other*)

Will work over trafarets sequentially

class trafaret.**Forward**

```
>>> node = Forward()
>>> node << Dict(name=String, children=List[node])
>>> node
<Forward(<Dict(children=<List(<recur>)>, name=<String>)>)>
>>> node.check({"name": "foo", "children": []}) == {'children': [], 'name': 'foo'}
True
>>> extract_error(node, {"name": "foo", "children": [1]})
{'children': {0: 'value is not a dict'}}
>>> node.check({"name": "foo", "children": [                          {"name": "bar
↪", "children": []}                        ]}) == {'children': [{'children': [],
↪'name': 'bar'}], 'name': 'foo'}
True
>>> empty_node = Forward()
>>> empty_node
<Forward(None)>
>>> extract_error(empty_node, 'something')
'trafaret not set yet'
```

class trafaret.**Any**

```
>>> Any()
<Any>
>>> (Any() >> ignore).check(object())
```

class trafaret.**Null**

```
>>> Null()
<Null>
>>> Null().check(None)
>>> extract_error(Null(), 1)
'value should be None'
>>> Null().is_valid(None)
```

(continues on next page)

(continued from previous page)

```
True
>>> Null().is_valid(1)
False
```

class trafaret.**Iterable**(*trafaret*, *min_length=0*, *max_length=None*)

```
>>> List(Int)
<List(<Int>)>
>>> List(Int, min_length=1)
<List(min_length=1 | <Int>)>
>>> List(Int, min_length=1, max_length=10)
<List(min_length=1, max_length=10 | <Int>)>
>>> extract_error(List(Int), 1)
'value is not a list'
>>> List(Int).check([1, 2, 3])
[1, 2, 3]
>>> List(String).check(["foo", "bar", "spam"])
['foo', 'bar', 'spam']
>>> extract_error(List(Int), [1, 2, 1 + 3j])
{2: 'value is not int'}
>>> List(Int, min_length=1).check([1, 2, 3])
[1, 2, 3]
>>> extract_error(List(Int, min_length=1), [])
'list length is less than 1'
>>> List(Int, max_length=2).check([1, 2])
[1, 2]
>>> extract_error(List(Int, max_length=2), [1, 2, 3])
'list length is greater than 2'
>>> extract_error(List(Int), ["a"])
{0: "value can't be converted to int"}
>>> List(Int).is_valid([1, 2, 3])
True
>>> List(Int).is_valid(1)
False
```

class trafaret.**List**(*trafaret*, *min_length=0*, *max_length=None*)

class trafaret.**Key**(*name*, *default=<object object>*, *optional=False*, *to_name=None*, *trafaret=None*)

Helper class for Dict.

It gets `name`, and provides method `extract(data)` that extract key value from data through mapping `get` method. Key *__call__* method yields `(key name, result or DataError, [touched keys])` triples.

You can redefine `get_data(data, default)` method in subclassed `Key` if you want to use something other then `.get(...)` method.

Like this for the aiohttp MultiDict:

```
class MDKey(t.Key):
    def get_data(self, data, default):
        return data.getall(self.name, default)
```

class trafaret.**Dict**(**args*, ***trafarets*)

Dict is a most complex trafaret that going with this library. The main difference from other common validation libs is that *Dict* trafaret does not know anything about actual keys. Every key that Dict works with knows itself how to get value from given mapping and if this will be one value or two or multi values.

So *Dict* cannot make any assumptions about whats going on other then contract with a *key* implementation. And we need to note, that any callable can be *key*, not only *Key* subclasses. So we need to look at the *key* contract:

```
key_instance(data: Mapping) -> Sequence[
    name_to_store,
    result or DataError,
    Sequence[touched keys],
]
```

It is a bit complex, so let me explain it to you. Every key instance get this data that *Dict* trying to check. Then *key* will return one or multiple results and it is common for *key* to be generator.

Every result is three component tuple 1. Key for the result dict. For standard *Key* it is result key in case of successful check or original key name if there was an error 2. Result if keys trafaret check was successful or DataError instance otherwise 3. An iterable with all keys of original mapping that this *key* touched

With this tricky interface *key* in our lib can do anything you can imagine. Like work with MultiDicts, compare keys, get subdicts and check them independently from main one.

Why we need this third extra iterable with touched names? Because our Dict can check that all keys were consumed and what to do with extras.

Arguments:

- Dict accepts keys as **args*
- if first argument to Dict is a *dict* then its keys will be merged with args keys and this *dict* values must be trafarets. If key of this *dict* is a str, then Dict will create *Key* instance with this key as Key name and value as its trafaret. If *key* is a *Key* instance then Dict will call this key *set_trafaret* method.

allow_extra argument can be a list of keys, or '*' for any, that will be checked against *allow_extra_trafaret* or *Any*.

ignore_extra argument can be a list of keys, or '*' for any, that will be ignored.

allow_extra(**names*, ***kw*)
multi arguments that represents attribute names or *. Will allow unconsumed by other keys attributes for given names or all if includes *. Also you can pass *trafaret* keyword argument to set *Trafaret* instance for this extra args, or it will be *Any*. Method creates *Dict* clone.

ignore_extra(**names*)
multi arguments that represents attribute names or *. Will ignore unconsumed by other keys attribute names for given names or all if includes *. Method creates *Dict* clone.

merge(*other*)
Extends one Dict with other Dict Key's or Key's list, or dict instance supposed for Dict

class trafaret.**Enum**(**variants*)

```
>>> trafaret = Enum("foo", "bar", 1) >> ignore
>>> trafaret
<Enum('foo', 'bar', 1)>
>>> trafaret.check("foo")
>>> trafaret.check(1)
>>> extract_error(trafaret, 2)
"value doesn't match any variant"
>>> trafaret.is_valid(1)
True
>>> trafaret.is_valid(2)
False
```

class trafaret.**Tuple**(**args*)

Tuple checker can be used to check fixed tuples, like (Int, Int, String).

```
>>> t = Tuple(Int, Int, String)
>>> t.check([3, 4, '5'])
(3, 4, '5')
>>> extract_error(t, [3, 4, 5])
{2: 'value is not a string'}
>>> t
<Tuple(<Int>, <Int>, <String>)>
```

class trafaret.**Atom**(*value*)

```
>>> Atom('atom').check('atom')
'atom'
>>> extract_error(Atom('atom'), 'molecule')
"value is not exactly 'atom'"
>>> Atom('atom').is_valid('atom')
True
>>> Atom('atom').is_valid('molecule')
False
```

class trafaret.**String**(*allow_blank=False*, *min_length=None*, *max_length=None*)

```
>>> String()
<String>
>>> String(allow_blank=True)
<String(blank)>
>>> String().check("foo")
'foo'
>>> extract_error(String(), "")
'blank value is not allowed'
>>> String(allow_blank=True).check("")
''
>>> extract_error(String(), 1)
'value is not a string'
>>> String(min_length=2, max_length=3).check('123')
'123'
>>> extract_error(String(min_length=2, max_length=6), '1')
'String is shorter than 2 characters'
>>> extract_error(String(min_length=2, max_length=6), '1234567')
'String is longer than 6 characters'
>>> String(min_length=2, max_length=6, allow_blank=True)
Traceback (most recent call last):
...
AssertionError: Either allow_blank or min_length should be specified, not both
>>> String(min_length=0, max_length=6, allow_blank=True).check('123')
'123'
>>> String().is_valid("foo")
True
>>> String().is_valid("")
False
>>> String().is_valid(1)
False
```

str_type

alias of __builtin__.unicode

class trafaret.**Date**(*format='%Y-%m-%d'*)

Checks that value is a *datetime.date* & *datetime.datetime* instances or a string that is convertable to *datetime.date* object.

```
>>> Date()
<Date %Y-%m-%d>
>>> Date('%y-%m-%d')
<Date %y-%m-%d>
>>> Date().check(date.today())
datetime.date(2019, 7, 25)
>>> Date().check(datetime.now())
datetime.datetime(2019, 10, 6, 14, 42, 52, 431348)
>>> Date().check("2019-07-25")
'2019, 7, 25'
>>> Date(format='%y-%m-%d').check('00-01-01')
'00-01-01'
>>> extract_error(Date(), "25-07-2019")
'value does not match format %Y-%m-%d'
>>> extract_error(Date(), 1564077758)
'value cannot be converted to date'
>>> Date().is_valid(date.today())
True
>>> Date().is_valid(1564077758)
False
```

class trafaret.**ToDate**(*format='%Y-%m-%d'*)

Returns instance of *datetime.date* object if value is a string or *datetime.date* & *datetime.datetime* instances.

```
>>> ToDate().check(datetime.now())
datetime.date(2019, 10, 6)
>>> ToDate().check("2019-07-25")
datetime.date(2019, 7, 25)
>>> ToDate(format='%y-%m-%d').check('00-01-01')
datetime.date(2000, 1, 1)
```

class trafaret.**DateTime**(*format='%Y-%m-%d %H:%M:%S'*)

Checks that value is a *datetime.datetime* instance or a string that is convertable to *datetime.datetime* object.

```
>>> DateTime()
<DateTime %Y-%m-%d %H:%M:%S>
>>> DateTime('%Y-%m-%d %H:%M')
<DateTime %Y-%m-%d %H:%M>
>>> DateTime().check(datetime.now())
datetime.datetime(2019, 7, 25, 21, 45, 37, 319284)
>>> DateTime('%Y-%m-%d %H:%M').check("2019-07-25 21:45")
'2019-07-25 21:45'
>>> extract_error(DateTime(), "2019-07-25")
'value does not match format %Y-%m-%d %H:%M:%S'
>>> extract_error(DateTime(), date.today())
'value cannot be converted to datetime'
>>> DateTime('%Y-%m-%d %H:%M').is_valid("2019-07-25 21:45")
True
>>> extract_error(DateTime(), "2019-07-25")
False
```

class trafaret.**ToDateTime**(*format='%Y-%m-%d %H:%M:%S'*)

Returns instance of *datetime.datetime* object if value is a string or *datetime.datetime* instance.

```
>>> DateTime('%Y-%m-%d %H:%M').check("2019-07-25 21:45")
datetime.datetime(2019, 7, 25, 21, 45)
```

class trafaret.**AnyString**(*allow_blank=False*, *min_length=None*, *max_length=None*)

class trafaret.**Bytes**(*allow_blank=False*, *min_length=None*, *max_length=None*)

class trafaret.**ToBytes**(*encoding='utf-8'*)
Get str and try to encode it with given encoding, utf-8 by default.

class trafaret.**FromBytes**(*encoding='utf-8'*)
Get bytes and try to decode it with given encoding, utf-8 by default. It can be used like `unicode_or_koi8r = String | FromBytes(encoding='koi8r')`

class trafaret.**Float**(*gte=None*, *lte=None*, *gt=None*, *lt=None*)
Tests that value is a float or a string that is convertable to float.

```
>>> Float()
<Float>
>>> Float(gte=1)
<Float(gte=1)>
>>> Float(lte=10)
<Float(lte=10)>
>>> Float(gte=1, lte=10)
<Float(gte=1, lte=10)>
>>> Float().check(1.0)
1.0
>>> extract_error(Float(), 1 + 3j)
'value is not float'
>>> extract_error(Float(), 1)
1.0
>>> Float(gte=2).check(3.0)
3.0
>>> extract_error(Float(gte=2), 1.0)
'value is less than 2'
>>> Float(lte=10).check(5.0)
5.0
>>> extract_error(Float(lte=3), 5.0)
'value is greater than 3'
>>> Float().check("5.0")
5.0
```

value_type
alias of `__builtin__.float`

class trafaret.**ToFloat**(*gte=None*, *lte=None*, *gt=None*, *lt=None*)
Checks that value is a float. Or if value is a string converts this string to float

class trafaret.**Int**(*gte=None*, *lte=None*, *gt=None*, *lt=None*)

```
>>> Int()
<Int>
>>> Int().check(5)
5
>>> extract_error(Int(), 1.1)
'value is not int'
>>> extract_error(Int(), 1 + 1j)
'value is not int'
```

value_type
alias of __builtin__.int

class trafaret.**ToInt**(*gte=None*, *lte=None*, *gt=None*, *lt=None*)

class trafaret.**ToDecimal**(*gte=None*, *lte=None*, *gt=None*, *lt=None*)

value_type
alias of decimal.Decimal

class trafaret.**Callable**

```
>>> (Callable() >> ignore).check(lambda: 1)
>>> extract_error(Callable(), 1)
'value is not callable'
>>> (Callable() >> ignore).is_valid(lambda: 1)
True
>>> Callable().is_valid(1)
False
```

class trafaret.**Bool**

```
>>> Bool()
<Bool>
>>> Bool().check(True)
True
>>> Bool().check(False)
False
>>> extract_error(Bool(), 1)
'value should be True or False'
>>> Null().is_valid(True)
True
>>> Null().is_valid(False)
True
>>> Null().is_valid(1)
False
```

class trafaret.**Type**(*type_*)

```
>>> Type(int)
<Type(int)>
>>> Type[int]
<Type(int)>
>>> c = Type[int]
>>> c.check(1)
1
>>> extract_error(c, "foo")
'value is not int'
>>> c.is_valid("foo")
False
```

typing_checker()
isinstance(object, class-or-type-or-tuple) -> bool

Return whether an object is an instance of a class or of a subclass thereof. With a type as second argument, return whether that is the object's type. The form using a tuple, isinstance(x, (A, B, ...)), is a shortcut for

isinstance(x, A) or isinstance(x, B) or ... (etc.).

class trafaret.**Subclass**(*type_*)

```
>>> Subclass(type)
<Subclass(type)>
>>> Subclass[type]
<Subclass(type)>
>>> s = Subclass[type]
>>> s.check(type)
<type 'type'>
>>> extract_error(s, object)
'value is not subclass of type'
```

typing_checker()

issubclass(C, B) -> bool

Return whether class C is a subclass (i.e., a derived class) of class B. When using a tuple as the second argument issubclass(X, (A, B, ...)), is a shortcut for issubclass(X, A) or issubclass(X, B) or ... (etc.).

class trafaret.**Mapping**(*key*, *value*)

Mapping gets two trafarets as arguments, one for key and one for value, like *Mapping(t.Int, t.List(t.Str))*.

class trafaret.**ToBool**

```
>>> extract_error(ToBool(), 'aloha')
"value can't be converted to Bool"
>>> ToBool().check(1)
True
>>> ToBool().check(0)
False
>>> ToBool().check('y')
True
>>> ToBool().check('n')
False
>>> ToBool().check(None)
False
>>> ToBool().check('1')
True
>>> ToBool().check('0')
False
>>> ToBool().check('YeS')
True
>>> ToBool().check('No')
False
>>> ToBool().check(True)
True
>>> ToBool().check(False)
False
```

trafaret.**DictKeys**(*keys*)

Checks if dict has all given keys

Parameters **keys** –

trafaret.**guard**(*trafaret=None*, ***kwargs*)

Decorator for protecting function with trafarets

```
>>> @guard(a=String, b=Int, c=String)
... def fn(a, b, c="default"):
...     '''docstring'''
...     return (a, b, c)
...
>>> fn.__module__ = None
>>> help(fn)
Help on function fn:
<BLANKLINE>
fn(*args, **kwargs)
    guarded with <Dict(a=<String>, b=<Int>, c=<String>)>
<BLANKLINE>
    docstring
<BLANKLINE>
>>> fn("foo", 1)
('foo', 1, 'default')
>>> extract_error(fn, "foo", 1, 2)
{'c': 'value is not a string'}
>>> extract_error(fn, "foo")
{'b': 'is required'}
>>> g = guard(Dict())
>>> c = Forward()
>>> c << Dict(name=str, children=List[c])
>>> g = guard(c)
>>> g = guard(Int())
Traceback (most recent call last):
...
RuntimeError: trafaret should be instance of Dict or Forward
```

class trafaret.**Regexp**(*regexp*, *re_flags=0*)

class trafaret.**RegexpRaw**(*regexp*, *re_flags=0*)

Check if given string match given regexp

class trafaret.**RegexpString**(**args*, ***kwargs*)

class trafaret.**URLSafe**(**args*, ***kwargs*)

class trafaret.**Hex**(**args*, ***kwargs*)

class trafaret.**OnError**(*trafaret*, *message*, *code=None*)

class trafaret.**WithRepr**(*trafaret*, *representation*)

trafaret.**ensure_trafaret**(*trafaret*)

Helper for complex trafarets, takes trafaret instance or class and returns trafaret instance

trafaret.**extract_error**(*checker*, **a*, ***kw*)

Helper for tests - catch error and return it as dict

trafaret.**ignore**(*val*)

Stub to ignore value from trafaret Use it like:

```
>>> a = Int >> ignore
>>> a.check(7)
```

trafaret.**catch**(*checker*, **a*, ***kw*)

Helper for tests - catch error and return it as dict

trafaret.**catch_error**(*checker*, **a*, ***kw*)

Helper for tests - catch error and return it as dict

6.2 trafaret.keys — custom Dict keys implementations

class trafaret.keys.**KeysSubset**(**keys*)

From checkers and converters dict must be returned. Some for errors.

```
>>> from . import extract_error, Mapping, String
>>> cmp_pwds = lambda x: {'pwd': x['pwd'] if x.get('pwd') == x.get('pwd1') else␣
↪DataError('Not equal')}
>>> d = Dict({KeysSubset('pwd', 'pwd1'): cmp_pwds, 'key1': String})
>>> sorted(d.check({'pwd': 'a', 'pwd1': 'a', 'key1': 'b'}).keys())
['key1', 'pwd']
>>> extract_error(d.check, {'pwd': 'a', 'pwd1': 'c', 'key1': 'b'})
{'pwd': 'Not equal'}
>>> extract_error(d.check, {'pwd': 'a', 'pwd1': None, 'key1': 'b'})
{'pwd': 'Not equal'}
>>> get_values = (lambda d, keys: [d[k] for k in keys if k in d])
>>> join = (lambda d: {'name': ' '.join(get_values(d, ['name', 'last']))})
>>> Dict({KeysSubset('name', 'last'): join}).check({'name': 'Adam', 'last': 'Smith
↪'})
{'name': 'Adam Smith'}
```

trafaret.keys.**confirm_key**(*name*, *confirm_name*, *trafaret*)

confirm_key - takes *name*, *confirm_name* and *trafaret*.

Checks if data[‘name’] equals data[‘confirm_name’] and both are valid against *trafaret*.

trafaret.keys.**subdict**(*name*, **keys*, ***kw*)

Subdict key.

Takes a *name*, any number of keys as args and keyword argument *trafaret*. Use it like:

```
def check_passwords_equal(data):
    if data['password'] != data['password_confirm']:
        return t.DataError('Passwords are not equal')
    return data['password']

passwords_key = subdict(
    'password',
    t.Key('password', trafaret=check_password),
    t.Key('password_confirm', trafaret=check_password),
    trafaret=check_passwords_equal,
)

signup_trafaret = t.Dict(
    t.Key('email', trafaret=t.Email),
    passwords_key,
)
```

trafaret.keys.**xor_key**(*first*, *second*, *trafaret*)

xor_key - takes *first* and *second* key names and *trafaret*.

Checks if we have only *first* or only *second* in data, not both, and at least one.

Then checks key value against trafaret.

6.3 trafaret.utils — utils for unfolding netsted dict syntax

There will be small helpers to render forms with exist trafarets for DRY.

trafaret.utils.**fold**(*data*, *prefix=''*, *delimeter='__'*)

```
>>> _dd(fold({'a__a': 4}))
"{'a': {'a': 4}}"
>>> _dd(fold({'a__a': 4, 'a__b': 5}))
"{'a': {'a': 4, 'b': 5}}"
>>> _dd(fold({'a__1': 2, 'a__0': 1, 'a__2': 3}))
"{'a': [1, 2, 3]}"
>>> _dd(fold({'form__a__b': 5, 'form__a__a': 4}, 'form'))
"{'a': {'a': 4, 'b': 5}}"
>>> _dd(fold({'form__a__b': 5, 'form__a__a__0': 4, 'form__a__a__1': 7}, 'form'))
"{'a': {'a': [4, 7], 'b': 5}}"
>>> repr(fold({'form__1__b': 5, 'form__0__a__0': 4, 'form__0__a__1': 7}, 'form'))
"[{'a': [4, 7]}, {'b': 5}]"
```

trafaret.utils.**unfold**(*data*, *prefix=''*, *delimeter='__'*)

```
>>> _dd(unfold({'a': 4, 'b': 5}))
"{'a': 4, 'b': 5}"
>>> _dd(unfold({'a': [1, 2, 3]}))
"{'a__0': 1, 'a__1': 2, 'a__2': 3}"
>>> _dd(unfold({'a': {'a': 4, 'b': 5}}))
"{'a__a': 4, 'a__b': 5}"
>>> _dd(unfold({'a': {'a': 4, 'b': 5}}, 'form'))
"{'form__a__a': 4, 'form__a__b': 5}"
```

6.4 trafaret.constructor — methods to access object's attribute/netsted key by path

class trafaret.constructor.**C**

Start object. It has | and & operations defined that will use construct to it args

Use it like *C & int & check_less_500*

trafaret.constructor.**construct**(*arg*)

Shortcut syntax to define trafarets.

- int, str, float and bool will return t.Int, t.String, t.Float and t.Bool
- one element list will return t.List
- tuple or list with several args will return t.Tuple
- dict will return t.Dict. If key has '?' at the and it will be optional and '?' will be removed
- any callable will be t.Call
- otherwise it will be returned as is

construct is recursive and will try construct all lists, tuples and dicts args

CHAPTER 7

Changelog

CHAPTER 8

2.1.0

- fix for *Dict* merge
- new *is_valid* method

CHAPTER 9

2.0.2

- construct for *int* and *float* will use *ToInt* and *ToFloat*

9.1 2.0.0

- `WithRepr` – use it to return custom representation, like `<Email>`
- Strip a lot from dict, like `keys()`
- Trafarets are not mutable
- DataError has new `code` attribute, self.failure has `code` argument
- OnError has `code` argument too
- New `DataError.to_struct` method that returns errors in more consistent way
- `String`, `AnyString`, `Bytes`, `FromBytes(encoding=utf-8)`
- `Int`, `ToInt`, `Float`, `ToFloat`
- `ToDecimal`
- `Iterable` that acts like a `List`, but works with any iterable
- New `Date`, `ToDate` and `DateTime`, `ToDateTime` trafarets
- `StrBool` trafaret renamed to `ToBool`
- `Visitor` trafaret was deleted
- Test coverage

9.2 1.0.3

- new `trafaret.keys` dict key `subdict` from `trafaret_schema`

9.3 1.0.1

- `Date` catches `TypeError` in cases like `None`

9.4 1.0.0

- `Or` is immutable now
- fixes for `OnError`, `DeepKey`
- default `Key` implementations for `Dict` will return original key name in case of incorrect value

9.5 2017-08-04

- converters and `convert=False` are deleted in favor of `And` and `&`
- `String` parameter `regex` deleted in favor of `Regexp` and `RegexpRaw` usage
- new `OnError` to customize error message
- `context=something` argument for `__call__` and `check` Trafaret methods. Supported by `Or`, `And`, `Forward` etc.
- new customizable method `transform` like `change_and_return` but takes `context=` arg
- new `trafaret_instance.async_check` method that works with `await`

9.6 2017-05-12

- removed entrypoint magic
- 0.10.0

9.7 2017-03-25 0.9.0

- added *And* trafaret and *&* shortcut operation.
- change >> behaviour. From now on Trafaret does not use self.converters and use *And* trafaret instead
- added *RegxpRaw* and *Regexp* trafarets. *RegexpRaw* returns re.Match object and *Regexp* returns match string.
- deprecate *String regex* argument in favor to *Regexp* and *RegexpRaw* usage
- *Dict* now takes *allow_extra*, *allow_extra_trafaret* and *ignore_extra* keyword arguments as preferred alternative to methods

9.8 0.8.1

- added trafaret.constructor. Now you can use *construct* and *C* from this package.

9.9 2016-09-25

Added *trafaret* argument to *DataError* constructor and made *_failure* a method (rather than static method)

9.10 2016-08-03

Added *Subclass* trafaret.

9.11 2016-03-31

Fixed loading contrib modules, so now original contrib module loading exception will be raised on contrib Trafaret access. Added *value* option to internal _failure interface, and option *value* to *DataError.as_dict* method.

9.12 2016-03-18

Fixed Key default behaviour for Dict with allowed extra when names are the same in Key and in data source

9.13 2014-09-17

Fixed Email validator

9.14 2012-05-30

Renamed methods to *check_value* and *check_and_return*. Added *Tuple* trafaret.

9.15 2012-05-28

Fixed *Dict(. . .).make_optional(. . .)* method for a chaining support

9.16 2012-05-21

Updated *KeysSubSet* errors propagation - now you can return error either *{'a': DataError('message')}*, or *DataError({'a': 'message'})*

9.17 2012-05-16

Added *__call__* alias to *check*.

9.18 2012-05-11

Added *visitor* module.

9.19 2012-05-10

Fixed *Dict.allow_extra* behaviour.

9.20 2012-04-12

Int will not convert not-rounded floats like 2.2

Dict have *.ignore_extra* method, similar to *.allow_extra*, but given keys will not included to result dict. If you will provide *, any extra will be ignored.

CHAPTER 10

Indices and tables

- genindex
- modindex * *Introducing* * *API docs*
- search

Python Module Index

t

trafaret, 31
trafaret.constructor, 42
trafaret.keys, 41
trafaret.utils, 42

Index

A

allow_extra() (*trafaret.Dict method*), 34
And (*class in trafaret*), 32
Any (*class in trafaret*), 32
AnyString (*class in trafaret*), 37
as_dict() (*trafaret.DataError method*), 31
Atom (*class in trafaret*), 35

B

Bool (*class in trafaret*), 38
Bytes (*class in trafaret*), 37

C

C (*class in trafaret.constructor*), 42
Call (*class in trafaret*), 31
Callable (*class in trafaret*), 38
catch() (*in module trafaret*), 40
catch_error() (*in module trafaret*), 40
check() (*trafaret.Trafaret method*), 31
confirm_key() (*in module trafaret.keys*), 41
construct() (*in module trafaret.constructor*), 42

D

DataError, 31
Date (*class in trafaret*), 36
DateTime (*class in trafaret*), 36
Dict (*class in trafaret*), 33
DictKeys() (*in module trafaret*), 39

E

ensure_trafaret() (*in module trafaret*), 40
Enum (*class in trafaret*), 34
extract_error() (*in module trafaret*), 40

F

Float (*class in trafaret*), 37
fold() (*in module trafaret.utils*), 42
Forward (*class in trafaret*), 32
FromBytes (*class in trafaret*), 37

G

guard() (*in module trafaret*), 39

H

Hex (*class in trafaret*), 40

I

ignore() (*in module trafaret*), 40
ignore_extra() (*trafaret.Dict method*), 34
Int (*class in trafaret*), 37
is_valid() (*trafaret.Trafaret method*), 31
Iterable (*class in trafaret*), 33

K

Key (*class in trafaret*), 33
KeysSubset (*class in trafaret.keys*), 41

L

List (*class in trafaret*), 33

M

Mapping (*class in trafaret*), 39
merge() (*trafaret.Dict method*), 34

N

Null (*class in trafaret*), 32

O

OnError (*class in trafaret*), 40
Or (*class in trafaret*), 31

R

Regexp (*class in trafaret*), 40
RegexpRaw (*class in trafaret*), 40
RegexpString (*class in trafaret*), 40

S

str_type (*trafaret.String attribute*), 35

String (*class in trafaret*), 35
Subclass (*class in trafaret*), 39
subdict() (*in module trafaret.keys*), 41

T

ToBool (*class in trafaret*), 39
ToBytes (*class in trafaret*), 37
ToDate (*class in trafaret*), 36
ToDateTime (*class in trafaret*), 36
ToDecimal (*class in trafaret*), 38
ToFloat (*class in trafaret*), 37
ToInt (*class in trafaret*), 38
Trafaret (*class in trafaret*), 31
trafaret (*module*), 31
trafaret.constructor (*module*), 42
trafaret.keys (*module*), 41
trafaret.utils (*module*), 42
Tuple (*class in trafaret*), 34
Type (*class in trafaret*), 38
typing_checker() (*trafaret.Subclass method*), 39
typing_checker() (*trafaret.Type method*), 38

U

unfold() (*in module trafaret.utils*), 42
URLSafe (*class in trafaret*), 40

V

value_type (*trafaret.Float attribute*), 37
value_type (*trafaret.Int attribute*), 37
value_type (*trafaret.ToDecimal attribute*), 38

W

WithRepr (*class in trafaret*), 40

X

xor_key() (*in module trafaret.keys*), 41