
Trafaret Documentation

Release 0.5.4

Mikhail Krivushin

November 13, 2015

1	Introducing trafaret	3
1.1	Features	3
1.2	DataError	4
1.3	Trafaret	4
1.4	Subclassing	4
1.5	Type	4
1.6	Any	4
1.7	Or	5
1.8	Null	5
1.9	Bool	5
1.10	Float	5
1.11	Int	5
1.12	Atom	5
1.13	String, Email, URL	6
1.14	List	6
1.15	Dict	6
1.16	Mapping	7
1.17	Enum	7
1.18	Callable	8
1.19	Call	8
1.20	Forward	8
1.21	guard	8
2	API docs	9
2.1	trafaret — Validation atoms definition	9
2.2	trafaret.extras — structs for trafaret structures extended definition	15
2.3	trafaret.utils — utils for unfolding netsted dict syntax	16
2.4	trafaret.visitor — methods to access object's attribute/netsted key by path	16
3	Indices and tables	19
	Python Module Index	21

Contents:

Introducing trafaret

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))
assert date.check({'year': 2012, 'month': 1, 'day': 12}) == datetime.datetime(2012, 1, 12)
```

`t.Dict` creates new dict structure validator with three `t.Int` elements. `>>` operation adds lambda function to the converters of given checker. Some checkers have default converter, but when you use `>>` or `.append`, you disable default converter with your own.

This does not mean that `Int` will not convert numbers to int, this mean that some checkers, like `String` with regular expression, have special converters applied to that can be overridden by your own.

Converters can be chained. You can raise `DataError` in converters.

1.1 Features

Trafaret has very handy features, read below some samples.

1.1.1 Regex String

`String` can work with regular expressions, and this gives you real power:

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

Some way you can use all `re.Match` power to extract from strings dicts and so on.

1.1.2 Dict and Key

`Dict` get dict with keys and checkers, like `{ 'a' : t.Int }`. But instead of string key you can use `Key` class. And `Key` instance can rename given key name to something else:

```
>>> c = t.Dict(t.Key('uNJ') >> 'user_name': t.String)
>>> c.check({'uNJ': 'Adam'})
{'user_name': 'Adam'}
```

And we can do more with right converter:

```
>>> from trafaret.utils import fold
>>> c = t.Dict({t.Key('uNJ') >> 'user__name': t.String}) >> fold
>>> c.check({'uNJ': 'Adam'})
{'user': {'name': 'Adam'}}
```

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'}
```

1.2 DataError

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

1.3 Trafaret

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

You can implement *converter* method if you want to convert value somehow, but want to make free for developer to apply his own converters to raw data. This used to return strings instead of *Match* object in *String* trafaret.

1.4 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.

1.5 Type

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

```
>>> Type(int).check(4)
4
```

1.6 Any

Will match any element.

1.7 Or

Get other converters as args. This samples are equivalent:

```
>>> Or(t.Int, t.Null).check(None)
None
>>> (t.Int | t.Null).check(5)
5
```

1.8 Null

Value must be *None*.

1.9 Bool

Check if value is boolean:

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

1.10 Float

Check if value is float or can be converted to. Supports *lte*, *gte*, *lt*, *gt* parameters:

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

1.11 Int

Similar to *Float*, but checking for int:

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

1.12 Atom

Value must be exactly equal to *Atom* first arg:

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

This may be useful in *Dict* in pair with *Or* statements.

1.13 String, Email, URL

Basically just check that arg is string. Argument `allow_blank` indicates if string can be blank or not. If you will provide `regex` param - will return `re.Match` object. Default converter will return `match.group()` result. You will get `re.Match` object in converter.

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` in converter.

So, some examples to make things clear:

```
>>> t.String().check('werwerwer')
'werwerwer'
>>> t.String(regex='^\s+$').check('   ')
'   '
>>> t.String(regex='^name=(\w+)$').check('name=Jeff')
'Jeff'
```

And one wild sample:

```
>>> todt = lambda m: datetime(*[int(i) for i in m.groups()])
>>> (t.String(regex='^year=(\d+),month=(\d+),day=(\d+)$') >> todt).check('year=2011,month=07,day=23')
datetime.datetime(2011, 7, 23, 0, 0)
```

1.14 List

Just List of elements of one type. In converter you will get list of converted elements. Sample:

```
>>> t.List(t.Int).check(range(100))
[0, 1, 2, ... 99]
>>> t.extract_error(t.List(t.Int).check(['a']))
{0: 'value cant be converted to int'}
```

1.15 Dict

Dict include named params. You can use for keys plain strings and `Key` instances. In case you provide just string keys, they will converted to `Key` instances. Actual checking proceeded in `Key` instance.

Methods:

`allow_extra(*names)` : where names can be key names or `*` to allow any additional keys.

`make_optional(*names)` : where names can be key names or `*` to make all options optional.

`ignore_extra(*names)`: where names are the names of the keys or `*` to exclude listed key names or all unspecified ones from the validation process and final result

`merge(Dict|dict|[t.Key...])` : 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`.

1.15.1 Key

Special class to create dict keys. Parameters are:

- name - key name

- default - default if key is not present
- optional - if True allow to not provide arg
- to_name - instead of key name will be returned this key

You can provide `to_name` with `>>` operation:: `Key('javaStyleData') >> 'plain_cool_data'`

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 MultiDict:

```
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 function:

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

check_args = t.Dict(simple_key)
```

1.15.2 KeysSubset

Experimental feature, not stable API. Sometimes you need to make something with part of dict keys. So you can:

```
>>> join = (lambda d: {'name': ' '.join(d.values())})
>>> Dict({KeysSubset('name', 'last'): join}).check({'name': 'Adam', 'last': 'Smith'})
{'name': 'Smith Adam'}
```

As you can see you need to return dict from checker.

1.15.3 Error raise

In Dict you can just return error from checkers or converters, there is need not to raise them.

1.16 Mapping

Check both keys and values:

```
>>> trafaret = Mapping(String, Int)
>>> trafaret
<Mapping(<String> => <Int>)>
>>> trafaret.check({"foo": 1, "bar": 2})
{'foo': 1, 'bar': 2}
```

1.17 Enum

This checker check that value one from provided. Like::

```
>>> Enum(1, 2, 'error').check('2')
2
```

1.18 Callable

Check if data is callable.

1.19 Call

Take a function that will be called in `check`. Function must return value or `DataError`.

1.20 Forward

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

```
>> node = Forward()
>> node << Dict(name=String, children=List[node])
```

1.21 guard

Decorator for function:

```
>>> @guard(a=String, b=Int, c=String)
... def fn(a, b, c="default"):
...     '''docstring'''
...     return (a, b, c)
```

1.21.1 GuardError

Derived from `DataError`.

2.1 trafaret — Validation atoms definition

exception `trafaret.DataError` (*error=None, name=None*)

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

class `trafaret.Trafaret`

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

Check that converters can be stacked `>>> (Int() >> (lambda x: x * 2) >> (lambda x: x * 3)).check(1) 6`

Check order `>>> (Int() >> float >> str).check(4) '4.0'`

append (*converter*)

Appends new converter to list.

check (*value*)

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

converter (*value*)

You can change converter with `>>` operator or `append` method

class `trafaret.Any`

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

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 `int`

class trafaret.**String** (*allow_blank=False, regex=None, 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(regex='\w+').check('wqerwqer')
'wqerwqer'
>>> extract_error(String(regex='^\w+$'), 'wqe rwqer')
'value does not match pattern: '^\w+$'
>>> 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'
```

class trafaret.**List** (*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 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'}
```

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

```
>>> trafaret = Dict(foo=Int, bar=String) >> ignore
>>> trafaret.check({"foo": 1, "bar": "spam"})
>>> extract_error(trafaret, {"foo": 1, "bar": 2})
{'bar': 'value is not a string'}
>>> extract_error(trafaret, {"foo": 1})
{'bar': 'is required'}
>>> extract_error(trafaret, {"foo": 1, "bar": "spam", "eggs": None})
{'eggs': 'eggs is not allowed key'}
>>> trafaret.allow_extra("eggs")
<Dict(extras=(eggs) | bar=<String>, foo=<Int>)>
>>> trafaret.check({"foo": 1, "bar": "spam", "eggs": None})
>>> trafaret.check({"foo": 1, "bar": "spam"})
>>> extract_error(trafaret, {"foo": 1, "bar": "spam", "ham": 100})
{'ham': 'ham is not allowed key'}
>>> trafaret.allow_extra("*")
<Dict(any, extras=(eggs) | bar=<String>, foo=<Int>)>
>>> trafaret.check({"foo": 1, "bar": "spam", "ham": 100})
>>> trafaret.check({"foo": 1, "bar": "spam", "ham": 100, "baz": None})
>>> extract_error(trafaret, {"foo": 1, "ham": 100, "baz": None})
{'bar': 'is required'}
>>> trafaret = Dict({Key('bar', optional=True): String}, foo=Int)
>>> trafaret.allow_extra("*")
<Dict(any | bar=<String>, foo=<Int>)>
>>> _dd(trafaret.check({"foo": 1, "ham": 100, "baz": None}))
"{'baz': None, 'foo': 1, 'ham': 100}"
>>> _dd(extract_error(trafaret, {"bar": 1, "ham": 100, "baz": None}))
"{'bar': 'value is not a string', 'foo': 'is required'}"
>>> extract_error(trafaret, {"foo": 1, "bar": 1, "ham": 100, "baz": None})
{'bar': 'value is not a string'}
>>> trafaret = Dict({Key('bar', default='nyanya') >> 'baz': String}, foo=Int)
>>> _dd(trafaret.check({'foo': 4}))
"{'baz': 'nyanya', 'foo': 4}"
>>> _ = trafaret.ignore_extra('fooz')
>>> _dd(trafaret.check({'foo': 4, 'fooz': 5}))
"{'baz': 'nyanya', 'foo': 4}"
>>> _ = trafaret.ignore_extra('*')
>>> _dd(trafaret.check({'foo': 4, 'foor': 5}))
"{'baz': 'nyanya', 'foo': 4}"
```

merge (*other*)

Extends one Dict with other Dict Key's or Key's list, or dict instance supposed for Dict

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'}
```

class trafaret.**Null**

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

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

```
>>> 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 float

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'
```

class trafaret.**Callable**

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

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.**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 dict'}}
>>> node.check({"name": "foo", "children": [{"name": "bar", "children":
True
>>> empty_node = Forward()
>>> empty_node
<Forward(None)>
>>> extract_error(empty_node, 'something')
'trafaret not set yet'

```

class trafaret.**Bool**

```

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

```

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))*.

`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:

fn(a, b, c='default')
    guarded with <Dict(a=<String>, b=<Int>, c=<String>)>

    docstring

>>> 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.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, Maybe(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(data, default):
        return data.get_all(self.name, default)
```

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'"
```

class trafaret.**Email** (*allow_blank=False*)

```
>>> Email().check('someone@example.net')
'someone@example.net'
>>> extract_error(Email(), 'someone@example') # try without domain-part
'value is not a valid email address'
>>> str(Email().check('someone@.')) # try with `idna` encoding
'someone@xn--elafmkfd.xn--plai'
>>> (Email() >> (lambda m: m.groupdict()['domain'])).check('someone@example.net')
'example.net'
>>> extract_error(Email(), 'foo')
'value is not a valid email address'
>>> extract_error(Email(), 'f' * 10000 + '@correct.domain.edu')
'value is not a valid email address'
>>> extract_error(Email(), 'f' * 248 + '@x.edu') == 'f' * 248 + '@x.edu'
True
```

class trafaret.**URL** (*allow_blank=False*)

```
>>> URL().check('http://example.net/resource/?param=value#anchor')
'http://example.net/resource/?param=value#anchor'
>>> str(URL().check('http://./resource/?param=value#anchor'))
'http://xn--elafmkfd.xn--plai/resource/?param=value#anchor'
```

2.2 trafaret.extras — structs for trafaret structures extended definition

class trafaret.extras.**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 e
>>> 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'}
>>> Dict({KeysSubset(): Dict({'a': Any})}).check({'a': 3})
{'a': 3}
```

2.3 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='', delimiter='__')`

```
>>> _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='', delimiter='__')`

```
>>> _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}"
```

2.4 trafaret.visitor — methods to access object's attribute/netsted key by path

This module is expirement. API and implementation are unstable. Supposed to use with Request object or something like that.

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

Lookup for attributes and items Path in name must be delimited by ..

```
>>> from trafaret import Int
>>> class A(object):
...     class B(object):
...         d = {'a': 'word'}
>>> dict((DeepKey('B.d.a') >> 'B_a').pop(A))
{'B_a': 'word'}
>>> dict((DeepKey('c.B.d.a') >> 'B_a').pop({'c': A}))
{'B_a': 'word'}
>>> dict((DeepKey('B.a') >> 'B_a').pop(A))
{'B.a': DataError(Unexistent key)}
>>> dict(DeepKey('c.B.d.a', to_name='B_a', trafaret=Int()).pop({'c': A}))
{'B_a': DataError(value can't be converted to int)}
```

class `trafaret.visitor.Visitor` (*keys*)

Check any object or mapping with `DeepKey` instances. This means that counts only existance and correctness of given paths. Visitor will not check for additional attributes etc.

`trafaret.visitor.get_deep_attr` (*obj, keys*)

Helper for `DeepKey`

Indices and tables

- [genindex](#)
- [modindex](#) * [Introducing trafaret](#) * [API docs](#)
- [search](#)

t

trafaret, [9](#)
trafaret.extras, [15](#)
trafaret.utils, [16](#)
trafaret.visitor, [16](#)

A

Any (class in trafaret), 9
append() (trafaret.Trafaret method), 9
Atom (class in trafaret), 14

B

Bool (class in trafaret), 13

C

Call (class in trafaret), 12
Callable (class in trafaret), 12
check() (trafaret.Trafaret method), 9
converter() (trafaret.Trafaret method), 9

D

DataError, 9
DeepKey (class in trafaret.visitor), 16
Dict (class in trafaret), 10

E

Email (class in trafaret), 15
Enum (class in trafaret), 12

F

Float (class in trafaret), 12
fold() (in module trafaret.utils), 16
Forward (class in trafaret), 13

G

get_deep_attr() (in module trafaret.visitor), 17
guard() (in module trafaret), 13

I

Int (class in trafaret), 9

K

Key (class in trafaret), 14
KeysSubset (class in trafaret.extras), 15

L

List (class in trafaret), 10

M

Mapping (class in trafaret), 13
merge() (trafaret.Dict method), 11

N

Null (class in trafaret), 11

O

Or (class in trafaret), 11

S

String (class in trafaret), 9

T

Trafaret (class in trafaret), 9
trafaret (module), 9
trafaret.extras (module), 15
trafaret.utils (module), 16
trafaret.visitor (module), 16
Tuple (class in trafaret), 14
Type (class in trafaret), 13

U

unfold() (in module trafaret.utils), 16
URL (class in trafaret), 15

V

value_type (trafaret.Float attribute), 12
value_type (trafaret.Int attribute), 9
Visitor (class in trafaret.visitor), 16