Trafaret Documentation

Release 1.1.2

Mikhail Krivushin

July 04, 2018

Table of Contents

Contents:

CHAPTER 1

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 integers, this mean that some checkers, like `String` with regular expression, have special converters applied them and can be overridden.

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 Regexp

`RegexpRaw` can work with regular expressions:

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

You can use all `re.match` power to extract from strings dicts and other higher level datastructures.

1.1.2 Dict and Key

`Dict` take as argument dictionaries with string keys and checkers as value, like `{'a': t.Int}`. But instead of a string key you can use the `Key` class. A `Key` instance can rename the 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 the 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 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.

1.3 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.

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*. For instance:

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

1.6 Any

Will match any element.

1.7 Or

Or takes other converters as arguments. The input is considered valid if one of the converters succeed:

```
>>> 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 a boolean:

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

1.10 Float

Check if value is a float or can be converted to a float. 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` with `Or` statements to create enumerations.

1.13 String, Email, URL

Basically just check that argument is a string.

Argument `allow_blank` indicates if string can be blank or not.

If you provide a `regex` parameter - it will return `re` match object. Default converter will return `match.group()` result.

`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.

Here is 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 parameters. 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 with `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* the key is optional
- *to_name* - allows to rename the 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'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)
```

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 a *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

Example:

```
>>> 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`.

CHAPTER 2

Changelog

2.1 1.0.3

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

2.2 1.0.1

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

2.3 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

2.4 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`

2.5 2017-05-12

- removed entrypoint magic
- 0.10.0

2.6 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

2.7 0.8.1

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

2.8 2016-09-25

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

2.9 2016-08-03

Added *Subclass* trafaret.

2.10 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.

2.11 2016-03-18

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

2.12 2014-09-17

Fixed Email validator

2.13 2012-05-30

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

2.14 2012-05-28

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

2.15 2012-05-21

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

2.16 2012-05-16

Added *__call__* alias to *check*.

2.17 2012-05-11

Added *visitor* module.

2.18 2012-05-10

Fixed *Dict.allow_extra* behaviour.

2.19 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 3

API docs

3.1 `trafaret` — Validation atoms definition

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

append(*other*)
Appends new converter to list.

check(*value*, *context=None*)
Common logic. In subclasses you need to implement check_value or check_and_return.

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'}
```

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'
```

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)
```

(continues on next page)

(continued from previous page)

```
<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"}
```

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.

make_optional(**args*)
Deprecated. will change in-place keys for given args or all keys if * in args.

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"
```

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

```
>>> String()
<String>
>>> String(allow_blank=True)
<String(blank)>
```

(continues on next page)

(continued from previous page)

```
>>> 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'
```

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

class trafaret.**IntRaw**(*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.**Callable**

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

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'
```

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)>
```

(continues on next page)

(continued from previous page)

```
>>> 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.**StrBool**

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

trafaret.**DictKeys**(*keys*)
Checks if dict has all given keys

Parameters **keys** –

```
>>> _dd(DictKeys(['a','b']).check({'a':1,'b':2,}))
"{'a': 1, 'b': 2}"
>>> extract_error(DictKeys(['a','b']), {'a':1,'b':2,'c':3,})
{'c': 'c is not allowed key'}
>>> extract_error(DictKeys(['key','key2']), {'key':'val'})
{'key2': 'is required'}
```

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(*args, **kwargs)
    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.**Regexp**(*regexp*, *re_flags=0*)

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

Check if given string match given regexp

class trafaret.**OnError**(*trafaret*, *message*)

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

3.2 trafaret.keys — custom Dict keys implementations

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.

3.3 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 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'}
```

3.4 `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}"
```

3.5 `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

3.6 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 4

Indices and tables

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

Python Module Index

t

Index

U

V

X