datatyping Documentation

Release 0.6.0

Carl Bordum Hansen

Oct 23, 2018

Contents

1 Installation **3**

2 Contents **5**
- 2.1 Getting Started 5
- 2.2 Defining Custom Types 6
- 2.3 Structure Generation 7

Python Module Index **9**

datatyping is a Python library, that can verify whether data is well-formed.

This makes datatyping useful for stuff that is currently hard in Python such as documenting incoming data or testing outgoing data.

```
>>> import datatyping
>>> structure = {'status_code': int, 'content': [str]}
>>> data = {'status_code': 400, 'content': ['Gouda', 'Cheddar']}
>>> datatyping.validate(structure, data)
```

This approach ensures early failure in a specific spot if data is malformed or has changed format unexpectedly which results in a more explicit codebase that is easier to maintain.

For more, check out Jeff Knupp's "How Python Makes Working With Data More Difficult in the Long Run" which inspired this library.

CHAPTER 1

Installation

```
$ pip install datatyping
```

CHAPTER 2

Contents

genindex, search

2.1 Getting Started

The bread and butter of *datatyping* is the *validate* function.

`datatyping.`**`validate`**(*structure*, *data*, *, *strict=True*)

Verify that values in a dataset are of correct types.

Example

```
>>> validate([str], ['a', 'b', 'c'])
>>> validate(
...     {'id': int, 'lucky_numbers': [int]},
...     {'id': 700, 'lucky_numbers': [1, 3, 7, 13]}
... )
>>> validate([int], [1, 2, 3, 4.5])
TypeError: 4.5 is of type float, expected type int.
```

Parameters

- **`structure`** (*type or collection of types*) – The data structure that *data* should follow.
- **`data`** (*anything*) – The data you want type checked.
- **`strict`** (*bool*) – Dicts in *data* must have the **exact** keys specified in *structure*. No more.

Raises

- `TypeError` – If an elements in *data* has wrong type.
- `ValueError` – If the length of *structure* doesn't make sense for validating *data*.

- `KeyError` – If a dict in *data* misses a key or *strict* is True and a dict has keys not in *structure*.

2.1.1 Examples

The following short examples are meant to clarify. If these are not sufficient, let me know (or maybe check out the unit tests :).

```
>>> validate([int, str], [1, 'a'])
>>> validate([[int], [str]], [[1, 2, 3], ['a', 'b', 'c']])

>>> validate([dict], [{'can have': 1}, {'any keys': 2}])
>>> validate({'a': int, 'b': str}, {'a': 4, 'b': 'c'})
>>> validate({'a': int}, {'a': 2, 'b': 'oops'})
KeyError: {'b'}
>>> validate({'a': int}, {'a': 2, 'b': 'yay'}, strict=False)
```

2.2 Defining Custom Types

Special constraints can be imposed with this handy decorater.

2.2.1 datatyping.customtype

`datatyping.`**`customtype`**(*check_function*)

Decorate a function, so it can be used for type checking.

Example

```
>>> @customtype
... def two_item_list(lst):
...     if len(lst) != 2:
...         raise TypeError('length %d!!!' % len(lst))
...
>>> validate([two_item_list], [[1, 2], [3, 4]])
>>> validate([two_item_list], [[1, 2], [3, 4, 5]])
TypeError: length 3!!!
```

Note: Sets the *check_function.__datatyping_validate* attribute.

Parameters **`check_function`** (*function*) – Function that should be used to type check.

2.2.2 Example

The following example defines a "custom type", that can only be positive integers.

```
from datatyping import validate, customtype
@customtype
def positive_int(i):
    if i < 1:
        raise TypeError('%d is not positive' % i)

validate([positive_int], [1, 2, 3, 4])
validate([positive_int], [1, 2, 3, -4])
TypeError: -4 is not positive
```

2.3 Structure Generation

It can be tedious to type out a structure. Luckily *datatyping* comes with a useful submodule for that called *printer*.

```
>>> from datatyping.printer import pprint
>>> import requests
>>> r = requests.get('http://httpbin.org/anything')
>>> pprint(r.json())
{
    'args': dict,
    'data': str,
    'files': dict,
    'form': dict,
    'headers': {
        'Accept': str,
        'Accept-Encoding': str,
        'Connection': str,
        'Host': str,
        'User-Agent': str,
    },
    'json': NoneType,
    'method': str,
    'origin': str,
    'url': str,
}
```

2.3.1 Printer module

Like pprint, but with types except for dictionary keys.

datatyping.printer.**pprint**(*object*, *stream=None*, *indent=4*, *width=80*, *depth=None*, *compact=False*)

Pretty-prints the data structure.

datatyping.printer.**pformat**(*object*, *indent=4*, *width=80*, *depth=None*, *compact=False*)

Return the pretty printed data structure of *object*.

Python Module Index

d

datatyping.printer, 7

Index

C

customtype() (in module datatyping), 6

D

datatyping.printer (module), 7

P

pformat() (in module datatyping.printer), 7
pprint() (in module datatyping.printer), 7

V

validate() (in module datatyping), 5