pyvaru Documentation

Release 0.3.0

Davide Zanotti

Oct 30, 2017

Contents

CHAPTER 1

What is pyvaru?

Pyvaru is a simple, flexible and unobtrusive data validation library for Python 3 (3.4+), based on the concept of validation rules.

From the software design point of view, a rule is a class implementing the strategy pattern, by encapsulating the validation logic in an interface method called `apply()`.

The library already offers a series of common validation rules like:

- `TypeRule` (it checks that the target value is an instance of the expected type)
- `FullStringRule` (it checks the the target value is a string with content)
- `ChoiceRule` (it checks that the target value is contained in a list of available options)
- `MinValueRule` (it checks that the target value is >= x) *
- `MaxValueRule` (it checks that the target value is <= x) *
- `MinLengthRule` (it checks that the target value length is >= x) *
- `MaxLengthRule` (it checks that the target value length is <= x) *
- `RangeRule` (it checks that the target value is contained in a given `range`)
- `IntervalRule` (it checks that the target value is contained in a given interval)
- `PatternRule` (it checks that the target value matches a given regular expression)
- `PastDateRule` (it checks that the target value is a date in the past)
- `FutureDateRule` (it checks that the target value is a date in the future)
- `UniqueItemsRule` (it checks that the target iterable does not contain duplicated items)

* where "x" is a provided reference value

The developer is then free to create his custom rules by extending the abstract `ValidationRule` and implementing the logic in the `apply()` method. For example:

```
class ContainsHelloRule(ValidationRule):
    def apply(self) -> bool:
        return 'hello' in self.apply_to
```

These rules are then executed by a `Validator`, which basically executes them in the provided order and eventually returns a `ValidationResult` containing the validation response.

CHAPTER 2

Installation

```
pip install pyvaru
```

CHAPTER 3

Usage

Given an existing model to validate, like the one below (but it could be a simple dictionary or any data structure since *pyvaru* does not make any assumption on the data format):

```
class User:
    def __init__(self, first_name: str, last_name: str, date_of_birth: datetime, sex:␣
↪str):
        self.first_name = first_name
        self.last_name = last_name
        self.date_of_birth = date_of_birth
        self.sex = sex
```

We have to define a validator, by implementing the `get_rules()` method and for each field we want to validate we have to provide one or more proper rule(s).

```
from pyvaru import Validator
from pyvaru.rules import TypeRule, FullStringRule, ChoiceRule, PastDateRule

class UserValidator(Validator):
    def get_rules(self) -> list:
        user = self.data # type: User
        return [
            TypeRule(apply_to=user,
                     label='User',
                     valid_type=User,
                     error_message='User must be an instance of user model.',
                     stop_if_invalid=True),
            FullStringRule(lambda: user.first_name, 'First name'),
            FullStringRule(lambda: user.last_name, 'Last name'),
            ChoiceRule(lambda: user.sex, 'Sex', choices=('M', 'F')),
            PastDateRule(lambda: user.date_of_birth, 'Date of birth')
        ]
```

It's also possible to create groups of rules by using `RuleGroup` and avoid code duplication if multiple rules should be applied to the same field. So this code:

```
def get_rules(self) -> list:
    return [
        TypeRule(lambda: self.data.countries, 'Countries', valid_type=list),
        MinLengthRule(lambda: self.data.countries, 'Countries', min_length=1),
        UniqueItemsRule(lambda: self.data.countries, 'Countries')
    ]
```

can be replaced by:

```
def get_rules(self) -> list:
    return [
        RuleGroup(lambda: self.data.countries,
                  'Countries',
                  rules=[(TypeRule, {'valid_type': list}),
                         (MinLengthRule, {'min_length': 1}),
                         UniqueItemsRule])
    ]
```

Finally we have two choices regarding how to use our custom validator:

1. As a context processor:

```
with UserValidator(user):
    # do whatever you want with your valid model
```

In this case the code inside `with` will be executed only if the validation succeed, otherwise a `ValidationException` (containing a `validation_result` property with the appropriate report) is raised.

2. By invoking the `validate()` method (which returns a `ValidationResult`)

```
validation = UserValidator(user).validate()
if validation.is_successful():
    # do whatever you want with your valid model
else:
    # you can take a proper action and access validation.errors
    # in order to provide a useful message to the application user,
    # write logs or whatever
```

Assuming we have a instance of an User configured as the one below:

```
user = User(first_name=' ',
            last_name=None,
            date_of_birth=datetime(2020, 1, 1),
            sex='unknown')
```

By running a validation with the previous defined rules we will obtain a `ValidationResult` with the following errors:

```
{
    'First name': ['String is empty.'],
    'Last name': ['Not a string.'],
    'Sex': ['Value not found in available choices.'],
    'Date of birth': ['Not a past date.']
}
```

CHAPTER 4

Full API Documentation

Go to: http://pyvaru.readthedocs.io/en/latest/contents.html

CHAPTER 5

Credits

Pyvaru is developed and maintained by Davide Zanotti.

Blog: http://www.daveoncode.com

Twitter: https://twitter.com/daveoncode

CHAPTER 6

Indices and tables

- genindex
- modindex
- search