VALIDATING FUNCTION ARGUMENTS IN PYTHON SIGNAL PROCESSING APPLICATIONS

Patrick Steffen Pedersen, Christian Schou Oxvig, Jan Østergaard, and Torben Larsen
Faculty of Engineering and Science, Department of Electronic Systems, Signal and Information Processing Section, Aalborg University, Denmark

DET FRIE FORSKNINGSRÅD
DANISH COUNCIL FOR INDEPENDENT RESEARCH

Proposal

Validation in Python

Python is dynamically typed and employs duck typing, relying on documentation, clear code and testing. As such, it does not have a built-in mechanism to validate the value of a function argument. This can lead to nonsensical exceptions, unexpected behaviour, erroneous results and the like.

The capabilities of existing argument validation solutions (such as PyContracts or Traits) vary from type, attribute, and value checks to fully customisable checks, and they rely on different interfaces and language constructs. Generally, there are a number of minor issues in using these solutions in signal processing applications:

- Some lack generality.
- Some do not promote readability.
- Some are inconvenient to use.

Most important, though, is that the validation scheme of function arguments is expressed in terms of Python objects rather than in terms of the data they hold. This is particularly problematic in signal processing applications and scientific computing in general.

A Motivating Example

Consider a very simple Python function which should accept only sequences with a non-negative, numerical first item:

```
def func(seq):
    return seq[0]**0.5
```

If an integer is passed, a somewhat non-sensible exception is raised, and this will typically be accompanied by a long traceback:

```
>>> func(42)
TypeError: 'int' object has no attribute '__getitem__'
```

However, if duck typing allows an invalid argument such as a mapping, no exception is raised, and an unintended behaviour is observed:

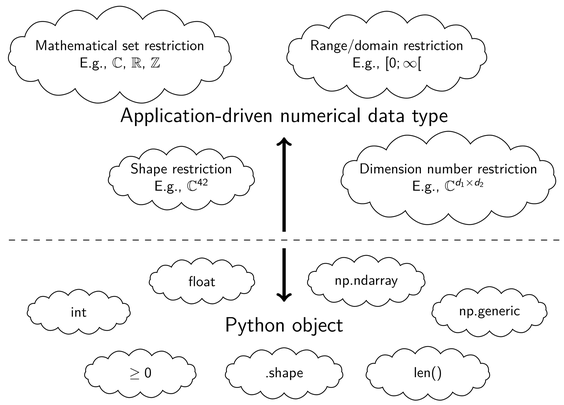
```
>>> func({-1: 0, 0: 1})
1.0
```

This example showcases how the lack of function argument validation can lead to hard-to-debug exceptions or even worse to unexpected or erroneous results. In this work we focus on preventing this by use of application-driven data type validation.

Implementation

Suggested Validation Schemes

We propose to specify the validation scheme in terms of application-driven data types as abstractions of Python data types.



The restrictions on the application-driven numerical data type translate to restrictions on the Python object, and vice versa. This translation is tasked to the implementation.

Interface Type and Python Constructs

We suggest allowing exactly one validation call for each argument of a function. The interface of the validation function should allow specifying the validation scheme in terms of the restrictions put on the general application-driven numerical data type. Consequently, the validation function should use named arguments and named keyword arguments to specify the allowed mathematical set(s), range/domain, dimension number, and shape.

As for Python constructs, we propose to define and explicitly call a nested function for validating the arguments of a function:

```
def func(*args, **kwargs):
    def validate_input():
        pass  # validate first arg
        pass  # validate second arg
        pass  # validate first kwarg
        pass  # validate second kwarg

    validate_input()

    pass  # the body of func
```

Perspective

`magni.utils.validation` Interface

A reference implementation is available as part of the open source Magni Python package.

The functions of `magni.utils.validation` most relevant to application-driven numerical data type validation are:

- `decorate_validation` and `disable_validation` are used to possibly disable validation globally.
- `validate_numeric` is used to validate numeric objects based on application-driven numerical data type validation schemes.

Usage Examples

Example of validation of a positive real scalar argument:

```
from magni.utils.validation import *

def func(var):
    @decorate_validation
    def validate_input():
        real = ('integer', 'floating')
        validate_numeric('var', real, range_=']0;inf[')

    validate_input()

func(1)  # will pass
func(3.14)  # will pass
func(0)  # will fail
func(-3.14)  # will fail
func(1j)  # will fail
```

Example of validation of a real matrix with its first dimension equal to 5, i.e., $\mathbb{R}^{5\times n}$ for any non-negative integer n:

```
from magni.utils.validation import *
import numpy as np

def func(var):
    @decorate_validation
    def validate_input():
        real = ('integer', 'floating')
        validate_numeric('var', real, shape=(5, -1))

    validate_input()

func(np.empty((5, 5)))  # will pass
func(np.empty((5, 10)))  # will pass
func(np.empty((10, 5)))  # will fail
func(np.empty((5, 5, 5)))  # will fail
```

Conclusions

We propose to

- Specify the validation schemes in terms of application-driven data types.
- Define a validation interface with named (keyword) arguments that relate to the validation scheme.
- Define and explicitly call a nested function to validate function arguments.
- Use this validation strategy to raise the abstraction level of function argument validation from Python to signal processing, to make validation easy to write, and to enhance readability of validation.

Details on Magni

The Magni Python package is fully documented and comes with an extensive test suite. It has been developed using best practices for developing scientific software and all code has been reviewed by at least one other person than its author prior to its inclusion in Magni. All code adheres to the PEP8 style guide and no function or class has a cyclomatic complexity exceeding 10. The source code is under version control using Git and a continuous integration system based on Travis CI is in use for the git repository.

Resources

- Official releases: `doi:10.5278/VBN/MISC/Magni`
- Online documentation: `http://magni.readthedocs.io`
- GitHub repo: `https://github.com/SIP-AAU/`
- Software Metapaper: `doi:10.5334/jors.bk`

Acknowledgements

This work was supported in part by the Danish Council for Independent Research (DFF/FTP) under Project 1335-00278B/12-134971 and in part by the Danish e-Infrastructure Cooperation (DeIC) under Project DeIC2013.12.23.