Validating Function Arguments in Python Signal Processing Applications

Patrick Steffen Pedersen‡*, Christian Schou Oxvig‡, Jan Østergaard‡, Torben Larsen‡

Abstract—Python does not have a built-in mechanism to validate the value of function arguments. This can lead to nonsensical exceptions, unexpected behaviour, erroneous results and the like. In the present paper, we define the concept of so-called application-driven data types which place a layer of abstraction on top of Python data types. With this concept in mind, we discuss the current argument validation solutions of PyDBC, Traitlets and Numtraits, MyPy, PyValid, and PyContracts. We find that they share the issue of expressing the validation scheme in terms of Python objects rather than in terms of the data they hold. Consequently, we lay out a suggestion for a validation strategy including what qualifies as a validation scheme, how to create an interface which promotes both usability and readability, and which Python constructs to encourage using for validation encapsulation. A reference implementation of the suggested validation strategy is part of the open-source Python package, Magni which is thus presented along with a number of examples of the usages of this package.

Index Terms—Function Argument Validation, Application-driven Data Types, Signal Processing, Computational Science

Introduction

Python is a dynamically typed language that does not have a built-in mechanism to ensure that the value of an argument passed to a function conforms to the intentions of that particular argument. This can lead to nonsensical exceptions, unexpected behaviour, erroneous results and the like. In signal processing applications and scientific computing in general, large amounts of numerical data are passed to any number of functions that inherently impose limitations upon that data. If such functions do not validate their arguments, these limitations may be violated without raising exceptions leading to potentially erroneous results. Thus, although impairing the performance, explicit validation may not only spare the user a lot of frustration by providing useful exceptions but may also prevent erroneous results and thereby ensure the credibility of works in scientific computing.

The usage of explicit function argument validation could be considered "unpythonic"[1] as it goes against dynamic typing [CVS13] and duck typing [CVS13] by not relying on documentation, clear code and testing to ensure correct usage. Even so, there exist a number of solutions for validating function arguments in Python relying on a wide range of language constructs and interfaces. The validation capabilities of these solutions vary greatly from type, attribute, and value checks to fully customisable checks. Among these solutions are PyDBC, Traitlets and Numtraits, MyPy, PyValid, and PyContracts which are all discussed later. Most of the solutions do, however, seem to have an interface which relates to the data model used by Python and therefore translates to Python check in a straightforward way.

Unfortunately, there are a number of shortcomings with the existing validation strategies as implemented in the existing Python packages with validation capabilities; in particular in signal processing applications. Some of the existing solutions lack generality, some do not promote readability, and some are inconvenient to use. However, the primary issue is that the validation scheme of function arguments is expressed in terms of Python objects rather than in terms of the data they hold, and this poses a number of problems. Even with these shortcomings, the existing solutions represent a large variety of validation strategies which are an obvious source of inspiration.

In the present effort, we suggest the concept of so-called application-driven data types as a signal processing data model for programming. These data types are intended for expressing the validation scheme of function arguments. Furthermore, based on existing solutions, we lay out a new strategy for validating function arguments in Python signal processing applications. Finally, we present the open-source Python package, Magni which includes a reference implementation of the suggested validation strategy, and we show a number of examples of the usage of this package.

The remainder of the present paper is organised as follows. We first take a look at validation in Python at a glance before presenting the concept of application-driven data types. Next, we discuss some of the existing solutions with an emphasis on the validation strategies they represent. Drawing on the observations made, we then present the suggested Python validation strategy. Following this specification, we detail a reference implementation of it and give examples of its usage. Finally, we conclude on what is achieved by the presented validation strategy and reference implementation as well as when to use them. All code examples have been run with Python 2.7 unless otherwise noted, all tracebacks have been removed to save space, and exception messages and the like have been broken across multiple lines using trailing backslashes where necessary.

* *Corresponding author: psp@es.aau.dk*
‡ *Faculty of Engineering and Science, Department of Electronic Systems, Section of Signal and Information Processing, Aalborg University, 9220 Aalborg, Denmark*

1. For an informal yet fitting definition, see http://stackoverflow.com/questions/25011078/what-does-pythonic-mean

Validation in Python at a glance

For the purpose of exemplifying the concepts discussed in this section, we define a simple Python function for returning the square root of the first item of a sequence. Obviously, only a sequence with a non-negative, numerical first item, $a_0 \in \mathbb{R}_{\geq 0}$, should be a valid argument of this function.

```
def do_something(a):
    print(a[0]**0.5)
```

To quote the Zen of Python[2], "there should be one-- and preferably only one --obvious way to do it" when faced with solving a task in Python, and the obvious ways to solve common tasks are oftentimes referred to as pythonic idioms. When it comes to function argument validation in Python, the most pythonic idiom is to clearly document what a function expects and then just try to use whatever gets passed to the function and either let exceptions propagate or catch attribute errors and raise other exceptions instead. This approach is well-suited for Python because it is a dynamically typed language. Basically, this means that variables, such as the function argument in the example, are not limited to hold values of a certain type. Instead, we can pass a number, a sequence, a mapping, or any other type to the example function. Regardless of the type, Python tries to use whatever value gets passed to the function which is a consequence of duck typing. The basic principle is that if a bird looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck. That is, if a value exhibits the desired behaviour, then that value probably is valid. Translated to our example, if the value of the function argument, `a`, has the `__getitem__` attribute which Python uses internally for retrieving the first item, then `a` probably is valid. Thus, the most pythonic idiom would rely on documentation, clear code, and testing to ensure correct usage rather than explicitly testing function arguments to ensure conformity to the intentions of the function.

What happens, then, if the value of a function argument is invalid by the reckoning of duck typing? This is the case with the following call as the built-in `int` type does not define `__getitem__`:

```
>>> integer = 42
>>> do_something(integer)
TypeError: 'int' object has no attribute \
'__getitem__'
```

With the following call, a `TypeError` exception is raised with a message that "`'int'` object has no attribute `'__getitem__'`". Even with this simple example, such an exception message is less sensible than desired. Furthermore, such an exception is as likely to occur in some obscure function call and, thus, be accompanied by a traceback with more levels than anyone would want. However, at least the presence of an exception indicates that something did not go as expected. What happens, however, if the value of a function argument is valid by the reckoning of duck typing but does not conform to the intentions of the function? This is the case with the following call as the built-in `dict` type defines `__getitem__` but with a different purpose than the `__getitem__` of sequences:

```
>>> dictionary = {-1: 0, 0: 1}
>>> do_something(dictionary)
1.0
```

The intention of the function is to operate on the first item of the function argument, but `dictionary` is unordered meaning that there is no such thing as a first item. However, the call does not raise an exception because of duck typing. This is an example of unexpected or erroneous behaviour.

The two examples of calls presented showcase how the lack of function argument validation can lead to hard-to-debug exceptions or even worse to unexpected or erroneous behaviour. The benefit of explicit function argument validation is that the mentioned problems should be avoided. Furthermore, by having such validation for functions that are part of a public API of released packages, the package is made more trustworthy and user-friendly.

How to Test for Validity

One way to test for validity would be to check if the value of a variable has a certain type. That is, to determine the validity based on what a value *is*. For example, we could rewrite the `do_something` example in the following way:

```
def do_something(a):
    if not isinstance(a, list):
        raise TypeError('Descriptive message.')

    if not isinstance(a[0], int):
        raise TypeError('Descriptive message.')

    print(a[0]**0.5)
```

Obviously, this approach to validation goes against dynamical typing as it restricts variables to only hold values of certain types. In the example, `a` may hold values of the type `list` or of a derived type, and the first item of `a` may hold values of the type `int` or of a derived type. Clearly, the validation in the above example is too restrictive: as the intention of the function is to allow a sequence with a non-negative, numerical first item, the following call should pass but instead fails the validation checks:

```
>>> sequence = (0., 1.)
>>> do_something(sequence)
TypeError: Descriptive message.
```

The issue is that a number of Python types represent sequences, and a number of Python types represent numbers. This could be accounted for in the example, but the point to stress is that the programmer should not have to know about every single Python type, nor should he or she have to explicitly list a large number of Python types for each validation check.

Another way to test for validity would be to check if the value of a variable displays a certain behaviour. That is, to determine the validity based on what a value *can do*. For example, we could rewrite the `do_something` example in the following way:

```
def do_something(a):
    if not hasattr(a, '__getitem__'):
        raise TypeError('Descriptive message.')

    if not hasattr(a[0], '__pow__'):
        raise TypeError('Descriptive message.')

    print(a[0]**0.5)
```

Clearly, this approach to validation is along the lines of duck typing as it explicitly checks for the presence of the required attribute. In the example, `a` may hold values of any type that defines the `__getitem__` attribute, and `a[0]` may hold values of any type that defines the `__pow__` attribute. Unlike with the first way to test for validity, the validation in the above example is not restrictive enough as already explained using the example with the dictionary. The same check could be achieved in a cleaner and

2. See https://www.python.org/dev/peps/pep-0020/

more thorough way using abstract base classes[3], but this solution would essentially suffer from the same type of problem.

Neither of the two ways to test for validity mentioned, consider the fact that the square root operation is only defined for non-negative `a[0]` values if complex numbers are ignored. Thus, a third way to partially test for validity would be to check if the value of a variable is in a set of valid values. That is, to determine validity based on what a value *contains*. For example, we could rewrite the `do_something` example in the following way:

```
def do_something(a):
    if len(a) < 1:
        raise ValueError('Descriptive message.')

    if a[0] < 0:
        raise ValueError('Descriptive message.')

    print(a[0]**0.5)
```

Obviously, this approach would have to be combined with something else to ensure that `a` is indeed a sequence and `a[0]` is indeed a number as covered by the first two ways to test for validity.

The Concept of Application-Driven Data Types

The approaches presented in the previous section do not even consider less common although valid cases such as non-derived types that only implicitly define the required attributes. Even more so, it is apparent that there is no straightforward way to test for validity based solely on what a value *is*, *can do*, or *contains*. A possible explanation for this is that all three approaches express the validation scheme in terms of Python objects rather than in terms of the data they hold. Indeed, it was easy to identify and in plain writing express that the function argument of the `do_something` example must be a sequence with a non-negative, numerical first item. Expressing the validation scheme in this way does provide a layer of abstraction.

Instead of checking if the value of `a` is a certain Python type, it would be convenient to be able to check if the value of `a` is a sequence. Likewise, instead of checking if the value of `a[0]` is a certain Python type containing a non-negative value, it would be convenient to be able to check if the value of `a[0]` is a non-negative, numerical type. Both "sequence" and "non-negative, numerical type" are examples of data types at a higher abstraction level than actual Python types, and we will name these abstractions application-driven data types.

In the context of scientific computing and signal processing in particular, the most relevant and interesting application-driven data types are numerical types. Here, an application-driven data type is some "mental" intersection between math and computer science in scientific computing and signal processing in particular. For example, the set of real-valued matrices with dimensions m times n, $\mathbb{R}^{m\times n}$, is an example of an application-driven data type. If the user is able to test the validity of a function argument against this application-driven data type, there is no need for the user to consider the distinction between Python floats, numpy generics, numpy ndarrays, and so on.

Existing Solutions

As mentioned in the introduction, there exist a number of solutions to validating function arguments in Python relying on a wide range of language constructs and interfaces and thereby representing a large variety of validation strategies. As these strategies are a source of inspiration for any new validation strategy, this section is used to briefly discuss some existing solutions with a focus on the three aspects which make up the suggested validation strategy: 1) The validation schemes that can be expressed and through that the abstraction level of the application-driven data types. 2) The way the interface of the implementation allows the validation scheme to be specified. 3) The Python constructs used to allow Python to validate the function arguments against the validation specification. Additionally, the relevant versions of Python are mentioned as 4) under each solution. Thus, the emphasis of this section is not to give a complete review of all existing solutions.

PyDBC

Although the original PyDBC[4] is long outdated, it represents an approach worth mentioning. The package allows so-called contracts to be specified using method preconditions, method postconditions, and class invariants. Thus, function argument validation can be performed using method preconditions. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar in the range $[0;1]$:

```
import dbc
__metaclass__ = dbc.DBC

class Example:
    def exemplify(self, a):
        pass  # do something

    def exemplify__pre(self, a):
        assert isinstance(a, float)
        assert 0 <= a <= 1
```

When an invalid value is passed, the following assertion error occurs:

```
>>> example = Example()
>>> example.exemplify(-0.5)
AssertionError
```

As for validation strategy, the following observations are made:

1. As shown in the example above, the validation function, `exemplify__pre` contains custom validity checks, as PyDBC does not include any functionality for specifying a validation scheme.
2. Without any functionality for specifying a validation scheme, there is no fixed interface, and the user instead writes a number of `assert` statements to validate the function arguments.
3. The Python constructs used rely on object oriented Python by using metaclasses. When the metaclass creates the class, it rewrites the function `exemplify` to first invoke the function named `exemplify__pre` when `exemplify` is called following a fixed naming scheme.
4. PyDBC was intended for Python 2.2 and has not been changed since 2005, but the package does work with Python 2.7. It does, however, not work with Python 3, but the same functionality could indeed be implemented in Python 3.

3. See https://docs.python.org/2/glossary.html#term-abstract-base-class

4. See http://www.nongnu.org/pydbc/

Traits, Traitlets, and Numtraits

Traits[5] is an extensive package by Enthought which provides class attributes with the additional characteristics of customisable initialisation, validation, delegation, notification, and even visualisation. Traitlets[6] is a lightweight Traits-like module which provides customisable validation, default values, and notification. Finally, Numtraits[7] adds to Traitlets with a numerical trait with more versatility in validation than that of the numerical traits of Traitlets. Thus, although hardly as intended by the developers, function argument validation can be performed using an attribute for each function argument. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar in the range $[0;1]$:

```
from numtraits import NumericalTrait
from traitlets import HasTraits

class Example(HasTraits):
    _a = NumericalTrait(ndim=0, domain=(0, 1))

    def exemplify(self, a):
        self._a = a

        pass  # do something
```

When an invalid value is passed, the following assertion error occurs:

```
>>> example = Example()
>>> example.exemplify(-0.5)
traitlets.traitlets.TraitError: _a should be in \
the range [0:1]
```

As for validation strategy, the following observations are made:

1. The validation scheme of Traitlets requires specifying a static Python type, allows specifying a valid range of values for numerical types, and allows specifying relevant properties for other specific types. Furthermore, the validation scheme of the numerical trait of Numtraits does not require specifying a static Python type but allows specifying the number of dimensions and the shape of a value.
2. As shown in the example above, the interface of the implementation lets the user specify the validation scheme using a single call for each function argument with named arguments, named keyword arguments and in some cases unspecified keyword arguments using `**kwargs`.
3. The Python constructs used rely on object oriented Python by using descriptors which modify the retrieving and modification of attribute values of objects. Thus, when assigning a new value to an attribute, the relevant descriptor validates the new value.
4. Traitlets and Numtraits work with Python 2.7 and with Python 3.3 or above.

Annotations, Type Hints, and MyPy

PEP 3107[8] is a Python enhancement proposal on function annotations which is a feature which has recently been added to Python. This PEP allows arbitrary annotations without assigning any meaning to the particular annotations. PEP 484[9] is a PEP on type hints which attach a certain meaning to particular annotations to hint the type of argument values and return values of functions. The most important goal of this is static analysis, but runtime type checking is mentioned as a potential goal also. For more information, see PEP 483[10] on the theory of type hints and PEP 482[11] for a literature overview for type hints. MyPy[12] is a static type checker which, thus, does not enforce data type conformance at runtime. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar:

```
def exemplify(a: float):
    pass  # do something

exemplify('0')
```

When the script above is passed to MyPy using Python 3.5, the following message is produced:

```
$ mypy example.py
example.py:4: error: Argument 1 to "exemplify" has \
incompatible type "str"; expected "float"
```

As for validation strategy, the following observations are made:

1. The validation scheme of MyPy requires specifying a static Python type or a union of static Python types. This is hardly surprising for a static type checker.
2. As mentioned, the syntax of annotations is given by PEP 3107, and the format of the type hints is given by PEP 484 making the type hints explicit and readable although a less well-known feature of Python.
3. The Python constructs used rely only on annotations and runs offline and separately of normal execution of Python code.
4. PEP 484 was accepted for Python 3.5, but the syntax is compatible with that of PEP 3107 which was accepted for Python 3.0, and thus MyPy works with Python 3.2 or above. Furthermore, PEP 484 suggests a syntax for Python 2.7 using comments instead of annotations, and MyPy supports this and thus also works with Python 2.7.

PyValid

As the name suggests, PyValid[13] is a Python validation package, and it allows validation of function arguments and function return values. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar:

```
from pyvalid import accepts

@accepts(float)
def exemplify(a):
    pass  # do something
```

When an invalid value is passed, the following assertion error occurs:

```
>>> exemplify(0)
pyvalid.__exceptions.ArgumentValidationError: The \
1st argument of exemplify() is not in a \
[<type 'float'>]
```

As for validation strategy, the following observations are made:

1. The validation scheme for PyValid requires specifying one or more static Python types and acts as a runtime

5. See http://docs.enthought.com/traits/
6. See http://traitlets.readthedocs.org/
7. See http://github.com/astrofrog/numtraits/
8. See https://www.python.org/dev/peps/pep-3107/
9. See https://www.python.org/dev/peps/pep-0484/
10. See https://www.python.org/dev/peps/pep-0483/
11. See https://www.python.org/dev/peps/pep-0482/
12. See http://mypy.readthedocs.org/
13. See http://uzumaxy.github.com/pyvalid/

type checker. Thus, in terms of validation scheme capabilities, this is equivalent to MyPy.
2. As shown in the example above, the interface of the implementation lets the user specify the validation scheme using a single call for an entire function with a single argument or keyword argument for each validated function argument.
3. The Python constructs used rely on decorators by including an `accept` decorator in order to precede function execution by function argument validation.
4. PyValid works with Python 2.6 or above and with Python 3.

PyContracts

PyContracts[14] is a Python package that allows declaring constraints on function arguments and return values. In the following example, the function argument, `a`, of the function, `exemplify` is validated to be a real scalar in the range $[0; 1]$:

```
from contracts import contract

@contract(a='float,>=0,<=1')
def exemplify(a):
    pass  # do something
```

When an invalid value is passed, the following assertion error occurs:

```
>>> exemplify(-0.5)
contracts.interface.ContractNotRespected: Breach \
for argument 'a' to exemplify().
Condition -0.5 >= 0 not respected
checking: >=0              for value: Instance of \
<type 'float'>: -0.5
checking: float,>=0,<=1    for value: Instance of \
<type 'float'>: -0.5
Variables bound in inner context:
```

As for validation strategy, the following observations are made:

1. The capabilities of PyContracts allows specifying any conceivable validation scheme. This is achieved in part through built-in capabilities including specifying one or more static types in a flexible way, specifying value ranges, and specifying flexible length/shape constraints. And in part through custom specifications by using so-called custom contracts.
2. As shown in the example above, the interface of the implementation lets the user specify the validation scheme using a single call for an entire function with a single keyword argument for each validated function argument. The validation schemes for the individual arguments are specified using a custom string format. As the validation scheme becomes more advanced, the specification becomes less Python-like and less readable. For example, the following was taken from an official presentation and allows an argument to be a list containing a maximum of two types of objects: `list(type(t)|type(u))`.
3. The Python constructs used rely on decorators by including a `contract` decorator in order to precede function execution by function argument validation. Depending on the preference of the user, the validation scheme is either specified through arguments of the decorator, through annotations in the form of type hints or custom annotations, or through docstrings following a specific format.
4. PyContracts works with Python 2 and with Python 3.

14. See http://andreacensi.github.com/contracts/

The Suggested Python Validation Strategy

This section lays out a suggestion for a Python validation strategy for validating function arguments in signal processing applications. This strategy uses the introduced concept of application-driven data types and the observations made on the strategies of existing solutions. As mentioned in the previous section, the suggested validation strategy is made up of three aspects which are discussed separately in the following.

The Suggested Validation Schemes

As described in a previous section, we want to specify validation schemes in terms of application-driven data types rather than in terms of what a valid Python object *is*, *can do*, or *contains*. Needless to say, a translation must still be made from application-driven data types to Python data types, but this task is left for the validation package according to the suggested validation strategy. For an early implementation, any application-driven data type will allow only a limited set of Python data types. This does, however, not mean that the application-driven data type is limited to a few Python data types. Rather, more Python data types may be added along the way as long as they provide the necessary attributes with the desired interpretation. Thus, effectively, the suggested validation strategy can be considered less strict than static type checking but more strict than duck type checking.

The numerical trait of the Numtraits package has an interesting approach which is not too different from the concept of application-driven data types. The numerical trait does not distinguish between Python data types as long as they are numerical, and this corresponds to the most general numerical application-driven data type able to assume any numerical value of any shape. Furthermore, the numerical trait allows restricting the data type to more restrictive data types by specifying a number of dimensions, a specific shape, and/or a range of valid values. Indeed, signal processing applications could benefit from having such an application-driven data type. However, in some applications it may be necessary to work with boolean values, integral values, real values, or complex values only. Therefore, it should be possible to restrict the data type to suit these cases in addition to the other possible restrictions allowed by numerical traits.

To summarise, in Python signal processing applications, there should be an application-driven data type representing the most general numerical value being able to assume any numerical value of any shape. This data type should be able to be restricted to less general data types by specifying the mathematical set, the range or domain of valid values, the number of dimensions, and/or the specific shape of the data type. The suggested validation schemes should be expressed in terms of the desired application-driven data type.

The Suggested Interface Type

Most of the existing solutions which were mentioned in the previous section specify the validation scheme of all function arguments of a function in a single call to the validation package in question. This is not the case with the traits of the Trailets and Numtraits packages which only specify the validation scheme of a single function argument in each call to the validation package.

From the perspective of the authors, the latter approach yields the better readability. Therefore, the suggested interface type should only let the user specify the validation scheme of a single function argument in each call.

As for the specifics of the interface, the validation scheme must be easy both for the programmer to state and for users to read. The PyContracts details its own format where the validation scheme is given by a string. However, it would be desirable to use a more standard Python interface to ease the usages even if it means having to be more verbose. On the other hand, the numerical trait of the Numtraits package uses named named arguments and keyword arguments which relate to the possible restrictions of the application-driven data types. From the perspective of the authors, the latter approach works well with application-driven data types and result in logical, easy to use interfaces. Therefore, the suggested interface should use named arguments and keyword arguments related to the possible restrictions of the general numerical application-driven data type to specify the validation scheme of function arguments.

The Suggested Python Constructs to Use

There are a lot of Python constructs which could potentially be used as showcased by the existing solutions. PyContracts allows the user to specify the validation scheme through the docstring of a function. However, most users would not expect docstrings to be parsed to yield the validation scheme, and furthermore the format used to specify the validation scheme would not be obvious because of the lack of restrictions put on docstrings. Therefore, docstrings are not suggested as a Python construct to use here. Annotations, as used by MyPy, are relatively new to Python, but that should not disqualify them from being used. However, the format used would not be obvious because there are few restrictions put on annotations so with the exception of type hints which are insufficient for this purpose. Therefore, annotations are not suggested as a Python construct to use here.

Next, there are the object oriented Python constructs. Metaclasses, as used by, PyDBC, have existed for a long time. However, these have changed over time, and so the metaclass attribute feature of Python 2 no longer works in Python 3, and only one metaclass is allowed per class in the more recent Python versions. Furthermore, the behaviour of metaclasses makes them impair the readability, especially to users that are unfamiliar with the construct. Therefore, metaclasses are not suggested as a Python construct to use here. Descriptors, as used by Traits, Traitlets, and Numtraits, are another feature applicable to object oriented Python, and these can provide flexibility and readability. However, they are limited to object oriented Python, and furthermore it seems unpythonic to validate function arguments by invoking descriptors through class instance attribute assignment. Therefore, descriptors are not suggested as a Python construct to use here.

Decorators, as used by PyValid and PyContracts, are a well-known and general Python construct. However, it is not immediately apparent if something goes on "under the hood", and the pythonic approach is to specify the validation scheme of all function arguments in a single decorator call, both of which affect readability. Therefore, decorators are not suggested as a Python construct to use here.

The suggested Python construct values explicit over implicit and promotes readability. The suggestion is to define and explicitly call a nested validation function with no arguments. There are a number of obvious alternatives which are not suggested for different reasons:

- It is not suggested to precede the function code by calls directly to a validation package because this does not clearly separate validation from the rest of the code.
- It is not suggested to use arguments for the validation function because this could potentially lead to error-prone validation if the validation function arguments are wrongly named or ordered, or the function arguments are renamed or reordered.
- It is not suggested to use a global rather than nested validation function because this could potentially separate the validation from the function and thus reduce readability.

Magni Reference Implementation

A reference implementation of the **suggested validation strategy** is made available by the open source Magni Python package [OPA+14] through the subpackage `magni.utils.validation`. The subpackage contains the following functions:

```
decorate_validation(func)
disable_validation()
validate_generic(
    name, type_, value_in=None, len_=None,
    keys_in=None, has_keys=None, ignore_none=False,
    var=None)
validate_levels(name, levels)
validate_numeric(
    name, type_, range_='[-inf;inf]', shape=(),
    precision=None, ignore_none=False, var=None)
```

Of these, `validate_generic` and `validate_levels` are concerned with validating objects outside the scope of the present paper. The function, `disable_validation` can be used to disable validation globally. Although discouraged, this can be done to remove the overhead of validating function arguments. As the name suggests, `decorate_validation` is a decorator, and this should be used to decorate every validation function with the sole purpose of being able to disable validation. Using the suggested validation strategy with Magni, the following structure is used for all validation adhering to **the suggested Python constructs to use**:

```
from magni.utils.validation import decorate_validation

def func(*args, **kwargs):
    @decorate_validation
    def validate_input():
        pass  # validation calls

    validate_input()

    pass  # the body of func
```

The remaining function, `validate_numeric`, is used to validate numeric objects based on application-driven data types as proposed by **the suggested validation scheme** of the validation strategy. This is done using the interface as proposed by **the suggested interface type** of the validation strategy: The `type_` argument is used for specifying one or more of the `boolean`, `integer`, `floating`, and `complex` subtype specifiers. The `range_` argument is used for specifying the set of valid values with a minimum value and a maximum value both of which may be included or excluded. The `shape` argument is used for specifying the shape with the entry, -1 allowing an arbitrary shape

for a given dimension and any non-negative entry giving a fixed shape for a given dimension.

The remaining arguments of `validate_numeric` are not directly related to the validation scheme but rather to the surrounding Python code. The `precision` argument is used for specifying one or more allowed precisions in terms of bits per value. The `name` argument is used for specifying which argument of the function to validate with the particular validation call. The `ignore_none` argument is a flag indicating if the validation call should ignore `None` objects and thereby accept them as valid. The `var` argument is irrelevant to the scope of the present paper and the reader is referred to the documentation for more information.

Additional resources for `magni` are:

- Official releases: doi:10.5278/VBN/MISC/Magni
- Online documentation: http://magni.readthedocs.io
- GitHub repository: https://github.com/SIP-AAU/Magni

Examples

As mentioned in relation to the suggested validation schemes, there should be an application-driven data type representing the most general numerical value being able to assume any numerical value of any shape. The following example validates a variable against exactly this application-driven data type. The validation only fails when a non-numerical object is passed as argument to `func`.

```
from magni.utils.validation import decorate_validation
from magni.utils.validation import validate_numeric
import numpy as np

def func(var):
    @decorate_validation
    def validate_input():
        all_types = ('boolean', 'integer',
                     'floating', 'complex')
        validate_numeric(
            'var', all_types, shape=None)

    validate_input()

    pass  # the body of the func
```

When valid values are passed, nothing happens:

```
>>> func(42)
>>> func(3.14)
>>> func(np.empty((5, 5), dtype=np.complex_))
```

However, when a non-numerical object is passed, the following exception occurs:

```
>>> func('string')
TypeError: The value(s) of >>var<<, 'string', must \
be numeric.
```

In the next example, the application-driven data type is any non-negative real scalar, i.e., $\mathbb{R}_{\geq 0}$.

```
from magni.utils.validation import decorate_validation
from magni.utils.validation import validate_numeric

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

    validate_input()

    pass  # the body of the func
```

When valid values are passed, nothing happens:

```
>>> func(0)
>>> func(3.14)
```

However, when a complex object or a negative float is passed, the following exception occurs:

```
>>> func(1j)
TypeError: The value(s) of >>var.dtype<<, \
<type 'complex'>, must be in ('integer', 'floating').

>>> func(-3.14)
ValueError: The value(s) of >>min(real(var))<<, \
-3.14, must be >= 0.
```

Notice, that the `range_` argument in the validation call of the previous includes the values zero and infinity using `[...]`. One or both of these values could be excluded using `(...)` or `]...[` as is the case in the next example, i.e., $\mathbb{R}_{>0}$.

```
from magni.utils.validation import decorate_validation
from magni.utils.validation import validate_numeric

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

    validate_input()

    pass  # the body of the func
```

When a valid value is passed, nothing happens:

```
>>> func(3.14)
```

However, when a zero-valued object is passed, the following exception occurs:

```
>>> func(0.)
ValueError: The value(s) of >>min(real(var))<<, \
0.0, must be > 0.
```

In the final example, the application-driven data type is any 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 decorate_validation
from magni.utils.validation import validate_numeric
import numpy as np

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

    validate_input()

    pass  # the body of the func
```

When a valid value is passed, nothing happens:

```
>>> func(np.empty((5, 5)))
>>> func(np.empty((5, 10)))
```

However, when an $\mathbb{R}^{10\times 5}$ object or an $\mathbb{R}^{5\times 5\times 5}$ object is passed, the following exception occurs:

```
>>> func(np.empty((10, 5)))
ValueError: The value(s) of>>var.shape[0]<<, 10, \
must be 5.

>>> func(np.empty((5, 5, 5)))
ValueError: The value(s) of >>len(var.shape)<<, 3, \
must be 2.
```

Requirements

The required dependencies for `magni` (as of version 1.4.0) are:

- Python >= 2.7 / 3.3
- Matplotlib [Hun07] (Tested on version >= 1.3)
- NumPy [vdWCV11] (Tested on version >= 1.8)
- PyTables[15] (Tested on version >= 3.1)
- SciPy [Oli07] (Tested on version >= 0.14)

It should be noted that the requirements other than Python and NumPy are due to `magni` rather than `magni.utils.validation`. In addition to the above requirements, `magni` has a number of optional dependencies but none of these are relevant to the usage of `magni.utils.validation`.

Quality Assurance

The Magni Python package has been developed according to best practices for developing scientific software [WAB+14], and every included piece of code has been reviewed by at least one person other than its author. Furthermore, the PEP 8[16] style guide is adhered to, no function has a cyclomatic complexity [McC76] exceeding 10, the code is fully documented, and an extensive test suite accompanies the package. More details about the quality assurance of `magni` is given in [OPA+14].

Conclusions

We have argued that function arguments should be validated according to data types at a higher abstraction level than actual Python types, and we have named these application-driven data types. Based on a discussion of existing validation solutions, we have suggested a Python validation strategy including three aspects: 1) The validation schemes that can be expressed. 2) The way the interface of the implementation allows the validation scheme to be specified. 3) The Python constructs used to allow Python to validate the function arguments. A reference implementation of this strategy is available in the open source Magni Python package which we have presented along with a number of examples. In short, `magni` and more generally the validation strategy should be used to abstract function argument validation from Python to signal processing, to make validation ease to write, and to enhance readability of validation.

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.

References

[CVS13] José Cordeiro, Maria Virvou, and Boris Shishkov. *Software and Data Technologies: 5th International Conference, ICSOFT 2010, Athens, Greece, July 22-24, 2010. Revised Selected Papers*. Springer, 2013.

[Hun07] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, May 2007. doi:10.1109/MCSE.2007.55.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976. doi:10.1109/TSE.1976.233837.

[Oli07] Travis E. Oliphant. Python for Scientific Computing. *Computing in Science & Engineering*, 9(3):10–20, May 2007. doi:10.1109/MCSE.2007.58.

[OPA+14] Christian Schou Oxvig, Patrick Steffen Pedersen, Thomas Arildsen, Jan Østergaard, and Torben Larsen. Magni: A Python Package for Compressive Sampling and Reconstruction of Atomic Force Microscopy Images. *Journal of Open Research Software*, 2(1):e29, October 2014. doi:10.5334/jors.bk.

[vdWCV11] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, March 2011. doi:10.1109/MCSE.2011.37.

[WAB+14] Greg Wilson, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, Ben Waugh, Ethan P. White, and Paul Wilson. Best Practices for Scientific Computing. *PloS Biology*, 12(1):e1001745, January 2014. doi:10.1371/journal.pbio.1001745.

15. See http://www.pytables.org/
16. See https://www.python.org/dev/peps/pep-0008/
17. See https://travis-ci.org/