Pankoff

Release 8.0

Yaroslav Pankovych

Apr 11, 2022

CONTENTS:

```
pip install --user pankoff
```

Pankoff is easy to use, lightweight, pure python library to validate, mutate, serialize/deserialize python classes.

FAQ:

```
> Q: How is that better then the others?
> A: It's not, it has it's unique features, and built using lots of tweaks and hacks.

> Q: So if it uses lots of tweaks and hacks, is it safe to use in prod?
> A: Yes, but read the docs carefully.
```

CHAPTER

ONE

USAGE EXAMPLES

1.1 Basic usage

Lets create `data.json` as following:

```
{
  "name": "Yaroslav",
  "age": 22,
  "salary": 100,
  "office": "Central Office",
  "position": "Manager",
  "greeting_template": "Hello, {}"
}
```

Now lets load it:

```
from pankoff.base import Container
from pankoff.combinator import combine
from pankoff.exceptions import ValidationError
from pankoff.magic import autoinit, Alias
from pankoff.validators import String, Number, BaseValidator, Predicate, LazyLoad


class Salary(BaseValidator):

    def __setup__(self, amount, currency):
        self.amount = amount
        self.currency = currency

    def mutate(self, instance, value):
        return f"{instance.name} salary is: {value}"

    def validate(self, instance, value):
        amount, currency = value.split()
        if int(amount) != self.amount or currency != self.currency:
            raise ValidationError(f"Wrong data in field: `{self.field_name}`")


@autoinit(merge=True)
class Person(Container):
    name = String()
```

(continues on next page)

(continued from previous page)

```
    age = Number(min_value=18)
    salary = combine(
        Predicate, Salary,
        # Predicate
        predicate=lambda instance, value: value == "100 USD",
        default=lambda instance, value: str(value) + " USD",
        # Salary
        amount=100, currency="USD"
    )
    office = Predicate(
        predicate=lambda instance, value: value in ["Central Office"],
        error_message="Invalid value for field {field_name}, got {value}"
    )
    position = Predicate(
        # NOTE: we can use `salary` field at this point
        predicate=lambda instance, value: value == "Manager" and "100 USD" in instance.
↪salary,
        error_message="Invalid value for {field_name}, person got into wrong position:
↪{value}"
    )
    payment = Alias("salary")
    job_desc = LazyLoad(factory=lambda instance: f"{instance.position} at {instance.
↪office}")

    def  __init__(self, greeting_template):
        self.greeting = greeting_template.format(self.name)

person = Person.from_path("data.json")
print(person)  # Person(name=Yaroslav, age=22, salary=Yaroslav salary is: 100 USD,␣
↪office=Central Office, position=Manager, job_desc=Manager at Central Office)
print(person.greeting)  # Hello, Yaroslav
```

1.2 Trying invalid data

Change your `data.json`

```
{
  "name": "Yaroslav",
  "age": 22,
  "salary": 100,
  "office": "Central Office",
  "position": "HR",
  "greeting_template": "Hello, {}"
}
```

Now load it:

```
>>> person = Person.from_path("data.json")
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
pankoff.exceptions.ValidationError: ['Invalid value for position, person got into wrong␣
↪position: HR']
```

1.3 Lets do some transformations

Here's our `data.json`:

```
{
    "name": "Yaroslav",
    "salary": "100 USD",
    "kind": 1
}
```

```
kinds = {
    1: "Good person",
    2: "Bad person"
}

class KindMutator(BaseValidator):

    def validate(self, instance, value):
        if value not in kinds:
            raise ValidationError(f"Person kind should be in {kinds.keys()}")

    def mutate(self, instance, value):
        return kinds[value]


@autoinit
class Person(Container):
    name = String()
    salary = Salary(amount=100, currency="USD")
    kind = KindMutator()

Person.from_path("data.json").to_path("mutated_data.json", indent=4)
```

And here's what we get in `mutated_data.json`:

```
{
    "name": "Yaroslav",
    "salary": "Yaroslav salary is: 100 USD",
    "kind": "Good person"
}
```

1.4 Making object factories

It is possible to make object factory based on the same Container class.

```
class Multiplication(BaseValidator):

    def validate(self, instance, value):
        if not isinstance(value, (int, float)):
            raise ValidationError(f"`{self.field_name}` should be a number")

    def mutate(self, instance, value):
        return value * instance.get_extra("multiplicator", default=1)

@autoinit
class Person(Container):
    name = String()
    age = Multiplication()


young_person = Person.extra(multiplicator=2)
old_person = Person.extra(multiplicator=5)

john = young_person(name="John", age=10)
yaroslav = old_person(name="yaroslav", age=10)

print(john)  # Person(name=John, age=20)
print(yaroslav)  # Person(name=yaroslav, age=50)
```

As you can see, `young_person` and `old_person` acting like completely different things, but in fact they're not.

Also, you can access underlying `extra` structure by doing `yaroslav._extra`, which returns `MappingProxyType` view.

1.5 Magic mixins

Lets say you want to create a mixin, normally you'd do:

```
class HelloMixin:

    def say(self):
        value = super().say()
        return f"My name is {value} !!!"

class Hello:

    def say(self):
        return self.name

class Person(HelloMixin, Hello):
    def __init__(self, name):
        self.name = name
```

(continues on next page)

(continued from previous page)

```
person = Person("Yaroslav")
print(person.say())  # My name is Yaroslav !!!
```

As you can see, we're using `super()` here. Magic mixins allows you to avoid that, e.g:

```
from pankoff.magic import MagicMixin

class HelloMixin(MagicMixin):

    def say(self, value):
        return f"My name is {value} !!!"

class Hello:

    def say(self):
        return self.name

class Person(HelloMixin, Hello):
    def __init__(self, name):
        self.name = name

person = Person("Yaroslav")
print(person.say())  # My name is Yaroslav !!!
```

So the idea is, when you have same method names in both mixin and its parent, mixin will consume parents' value implicitly.

In the example above, `value` parameter for `HelloMixin` is the result `say` method on `Hello` class.

Note that it'll chain through all the mixins in MRO.

CHAPTER

TWO

PANKOFF'S MAGIC

pankoff.magic.**autoinit**(*klass*, *verbose=False*)

Auto generates `__init__` method for your class based on its validators.

Parameters

- **merge** (*bool*) – in case you have existing `__init__` in your class, you can merge them
- **klass** (*type*) – Class to decorate
- **verbose** (*bool*) – In case its `True`, it'll print out the generated source for `__init__` method, defaults to `False`

Returns Same class but with newly created `__init__`

Raises **RuntimeError** – raised in case class already has `__init__` defined

```
@autoinit(verbose=True)
class Person:
    name = String()
```

Prints:

```
Generated __init__ method for <class '__main__.Person'>
def __init__(self, name):
    self.name = name
```

You can merge existing `__init__` with generated one by using `merge=True`, e.g:

```
@autoinit(verbose=True, merge=True)
class Person:
    name = String()

    def __init__(self, surname):
        self.full_name = self.name + " " + surname
```

Which prints:

```
Generated __init__ method for <class '__main__.Person'>
def __init__(self, name, *args, **kwargs):
    self.name = name
    user_defined_init(self, *args, **kwargs)
```

As you can see, you can use `self.name` straight away.

```
person = Person(name="Yaroslav", surname=Pankovych)
print(person.full_name)  # Yaroslav Pankovych
```

class pankoff.magic.**Alias**(*source*)

Create and alias for your fields.

Parameters **source** (*str*) – Attribute name to reffer to

```
>>> class Person:
...     full_person_name = String()
...     name = Alias("full_person_name")
```

```
>>> obj = Person("Yaroslav Pankovych")
>>> obj.name
"Yaroslav Pankovych"
```

class pankoff.magic.**MagicMixin**

Create `MagicMixin` by inheriting from it.

See examples: *Magic mixins*

CHAPTER

THREE

USING VALIDATORS

Pankoff defines a small preset of validator, and it allow you to define your own.

3.1 Default validators

By default, Pankoff defines a few validators, `String`, `Number`, `Type`, `Sized`, `Predicate`, `LazyLoad`. We'll go over each one below.

class pankoff.validators.**String**

Validate whether field is instance of type `str`.

```
>>> @autoinit
>>> class Person:
...     name  = String()
```

```
>>> person = Person(name="Guido")
```

class pankoff.validators.**List**

Validate whether field is instance of type `list`.

```
>>> @autoinit
>>> class Person:
...     items  = List()
```

```
>>> person = Person(items=[1, 2, 3])
```

class pankoff.validators.**Dict**(*required_keys=None*)

Validate whether field is instance of type `dict`.

```
>>> @autoinit
>>> class Person:
...     mapping  = Dict(required_keys=["name"])
```

```
>>> person = Person(mapping={"name": "Guido"})
```

class pankoff.validators.**Tuple**

Validate whether field is instance of type `tuple`.

```
>>> @autoinit
>>> class Person:
...     items  = Tuple()
```

```
>>> person = Person(items=(1, 2, 3))
```

class pankoff.validators.**Iterable**

Check whether field supports `collections.abc.Iterable` interface.

class pankoff.validators.**Container**

Check whether field supports `collections.abc.Container` interface.

class pankoff.validators.**Hashable**

Check whether field supports `collections.abc.Hashable` interface.

class pankoff.validators.**Iterator**

Check whether field supports `collections.abc.Iterator` interface.

class pankoff.validators.**Reversible**

Check whether field supports `collections.abc.Reversible` interface.

class pankoff.validators.**Generator**

Check whether field supports `collections.abc.Generator` interface.

class pankoff.validators.**Callable**

Check whether field supports `collections.abc.Callable` interface.

class pankoff.validators.**Collection**

Check whether field supports `collections.abc.Collection` interface.

class pankoff.validators.**Sequence**

Check whether field supports `collections.abc.Sequence` interface.

class pankoff.validators.**MutableSequence**

Check whether field supports `collections.abc.MutableSequence` interface.

class pankoff.validators.**ByteString**

Check whether field supports `collections.abc.ByteString` interface.

class pankoff.validators.**Set**

Check whether field supports `collections.abc.Set` interface.

class pankoff.validators.**MutableSet**

Check whether field supports `collections.abc.MutableSet` interface.

class pankoff.validators.**Mapping**

Check whether field supports `collections.abc.Mapping` interface.

class pankoff.validators.**MutableMapping**

Check whether field supports `collections.abc.MutableMapping` interface.

class pankoff.validators.**Awaitable**

Check whether field supports `collections.abc.Awaitable` interface.

class pankoff.validators.**Coroutine**

Check whether field supports `collections.abc.Coroutine` interface.

class pankoff.validators.**AsyncIterable**

Check whether field supports `collections.abc.AsyncIterable` interface.

class pankoff.validators.**AsyncIterator**

Check whether field supports `collections.abc.AsyncIterator` interface.

class pankoff.validators.**AsyncGenerator**

Check whether field supports `collections.abc.AsyncGenerator` interface.

class pankoff.validators.**Number**(*mix_value=None*, *max_value=None*)

Validate whether field is an instance of type `int` and within specified range.

Parameters

- **min_value** (*int, optional*) – minimum value for a field
- **max_value** (*int, optional*) – maximum value for a field

```
>>> @autoinit
>>> class Person:
...     age = Number(min_value=18, max_value=100)
```

```
>>> person = Person(age=25)
```

class pankoff.validators.**Type**(*types=(int, str, ...)*)

Validate whether field is instance of at least one type in `types`.

```
>>> @autoinit
>>> class Car:
...     speed = Type(types=(int, float))
```

```
>>> car = Car(speed=500.4)
```

class pankoff.validators.**Sized**(*min_size=None*, *max_size=None*)

Validate whether field length is within specified range.

Parameters

- **min_size** (*int, optional*) – minimum length for a field
- **max_size** (*int, optional*) – maximum length for a field

```
>>> @autoinit
>>> class Box:
...     size = Sized(min_size=20, max_size=50)
```

```
>>> box = Box(size=40)
```

class pankoff.validators.**LazyLoad**(*factory*)

Calculate an attribute based on other fields.

Parameters **factory** – callable to calculate value for current field, accepts current instance

```
>>> @autoinit
>>> class Person(Container):
...     name = String()
```

(continues on next page)

(continued from previous page)

```
...     surname = String()
...     full_name = LazyLoad(factory=lambda instance: f"{instance.name} {instance.
↪surname}")
```

```
>>> person = Person(name="Yaroslav", surname="Pankovych")
>>> print(person)
Person(name=Yaroslav, surname=Pankovych, full_name=Yaroslav Pankovych)
```

class pankoff.validators.**Predicate**(*predicate*, *default=None*, *error_message=None*)

Predicate is a bit more complex validator. Basically, it checks your field against specified condition in `predicate`.

`predicate` is a simple callable which should return either `True` or `False`. It'll be called with current `instance` and `value` to validate: `predicate(instance, value)`.

If `predicate` returned `False`, and `default` is set, `instance` and `value` will be propagated to `default` if `default` is callable, if it's not, `default` will be returned straight away.

THe key feature of `default` is that it can "normalize" your value if it's invalid. See example below.

Parameters

- **predicate** (*callable*) – function to call in order to validate value
- **default** (*callable, any, optional*) – default value to use if `predicate` returned `False`

```
>>> @autoinit
>>> class Person:
...     salary = Predicate(
...         predicate=lambda instance, value: value == "100 USD",
...         default=lambda instance, value: str(value) + " USD"
...     )
```

```
>>> person = Person(salary=100)
>>> person.salary
"100 USD"
```

As you can see, we just turned `100` into `"100 USD"`. You can also chain (see *Chaining validators*) `Predicate` with other validators, and normalized value will be propagated to further validators.

`Predicate` supports rich error messages:

```
>>> @autoinit
>>> class Car:
...     speed = Predicate(
...         predicate=lambda instance, value: value == 100,
...         error_message="Invalid value in field: {field_name}, got {value} for
↪{predicate}"
...     )
```

```
>>> car = Car(speed=50)
Traceback (most recent call last):
...
pankoff.exceptions.ValidationError: ['Invalid value in field: speed, got 50 for
↪<lambda>']
```

(continues on next page)

(continued from previous page)

3.2 Custom validators

You can define you ows validator by subclassing `BaseValidator`.

```
>>> from pankoff.base import BaseValidator
>>> from pankoff.exceptions import ValidationError
```

```
>>> class EnumValidator(BaseValidator):
...     def __setup__(self, allowed_values):
...         self.allowed_values = allowed_values
...     def mutate(self, instance, value):
...         return f"Mutated value: {value}"
...     def validate(self, instance, value):
...         if value not in self.allowed_values:
...             raise ValidationError(
...                 f"Invalid value in field {self.field_name}, value should be in {self.
↪allowed_values} "
...                 f"got {value}"
...             )
```

It is required for validators to define `validate`, but `__setup__` and *mutate* is optional.

You can use `mutate` to modify returned value when its being accessed. It won't be cached, `mutate` is re-calculated on every attribute access.

CHAPTER

FOUR

VALIDATING DATA

You can validate data using different methods for different use cases. Obvious case is to just create an instance of your class with validators defined in it, which we covered in previous sections.

But there's a few more options to it.

You can inherit some capabilities by subclassing `pankoff.base.Container`.

class pankoff.base.**Container**

asdict(*dump_aliases=False*)

Dump your object do a dict. Dumps only validator fields and aliases.

Parameters

- **obj** – object to dump
- **dump_aliases** (*bool*) – if `True`, dump alias fields as well, defaults to `False`

Returns dict

```
>>> Person(...).asdict(dump_aliases=True)
```

asjson(*dump_aliases=False*, ***kwargs*)

Same as `asdict`, but returns JSON string.

Parameters

- **dump_aliases** – if `True`, dump alias fields as well, defaults to `False`
- **kwargs** – keyword arguments will be propagated to `dumps`

Returns JSON string

```
>>> Person(...).asjson(dump_aliases=True, indent=4)
```

asyaml(*dump_aliases=False*, ***kwargs*)

Dump object to YAML. Works only if PyYAML is installed.

Parameters

- **dump_aliases** – if `True`, dump alias fields as well, defaults to `False`
- **kwargs** – keyword arguments will be propagated to `dumps`

Returns YAML string

```
>>> Person.asyaml(dump_aliases=True)
```

dumps(*dumps*, *dump_aliases=False*, ***kwargs*)

Dump current object using provided dumper, e.g `yaml.dump` or `json.dumps`.

Parameters

- **dumps** – callable to use on dump, defaults to `json.dumps`
- **dump_aliases** – if `True`, dump alias fields as well, defaults to `False`
- **kwargs** – keyword arguments will be propagated to `dumps`

```
>>> Person.dumps(yaml.dump, dump_aliases=True)
```

classmethod extra(***kwargs*)

Add extra setup for your class future instances. This way you can create "temporary" objects, a.k.a factories.

```
>>> fast_person = Person.extra(walk_speed=150)
>>> slow_person = Person.extra(walk_speed=10)
```

```
>>> yaroslav = fast_person(...)
>>> john = slow_person(...)
```

```
>>> print(yaroslav.get_extra("walk_speed"))  # 150
>>> print(john.get_extra("walk_speed"))  # 10
```

NOTE: extra args set at very beginning of instance setup, before any `__init__`/etc

See example: *Making object factories*

Parameters **kwargs** – arguments to set on instancee before `__init__` call

classmethod from_dict(*data*)

Make on object from dictionary.

Parameters **data** (*dict*) – dictionary to load

classmethod from_file(*fp*, *loader=<function load>*)

Loads content from file using `loader` and returns validated instance of it.

Parameters

- **fp** – file object
- **loader** – defaults to `json.load`

classmethod from_json(*data*, *loader=<function loads>*)

Loads JSON and returns validated instance of it.

classmethod from_path(*path*, *loader=<function load>*)

Reads file at given path and returns validates instance of it. Uses `loader` to load it.

Parameters

- **path** – file path
- **loader** – defaults to `json.load`

classmethod is_valid(*data*)

Validate data :param data: data to validate :type data: dict

Returns `True/False`

to_path(*path*, *dump_aliases=False*, *dumps=<function dumps>*, ***kwargs*)

Dump current instance to a file. :param path: path to dump to :type path: str

Parameters

- **dump_aliases** – if `True`, dump alias fields as well, defaults to `False`
- **dumps** – callable to use on dump, defaults to `json.dumps`
- **kwargs** – keyword arguments will be propagated to `dumps`

```
>>> Person(...).to_path("path/to/data.json", dump_aliases=True, indent=4)
```

classmethod validate(*data*)

Validate data and raise if its invalid.

Parameters **data** (*dict*) – data to validate

Raises ValidationError

Lets validate JSON directly:

```
>>> from pankoff.base import Container
```

```
>>> @autoinit
>>> class Person(Container):
...     name = String()
...     age = Number(min_value=18)
```

```
>>> data = """{"name": "John", "age": 18}"""
>>> obj = Person.from_json(data)
>>> obj.name
"John"
>>> obj.age
18
```

Besides that, there's 2 more method, `validate` and `is_valid`.

```
>>> Person.validate({"name": "Carl", "age": 17})
Traceback (most recent call last):
...
pankoff.exceptions.ValidationError: ['Attribute `age` should be >= 18']
```

```
>>> Person.is_valid({"name": "Carl", "age": 17})
False
```

There's also `from_dict` method, e,g `Person.from_dict({...})`.

4.1 Validation errors

class pankoff.exceptions.**ValidationError**(*errors*)

Parameters **errors** – a list of errors

CHAPTER

FIVE

CHAINING VALIDATORS

It is possible to combine validators together. There's a few ways to do this, I'll go over them below.

pankoff.combinator.**combine**(**validators*, ***kwargs*)

Returns either "raw" combined validator or an instance of it.

Parameters

- **validators** – Validators to combine
- **kwargs** – If specified, `kwargs` will be unpacked to newly created combined validator

Returns Either "raw" combined validator or its instance

5.1 Using `combine()` function

`combine()` allows you to do exactly 2 things, create "raw" combined validator, or create combined "instance".

Lets create "raw" combined validator first:

```
from pankoff.magic import autoinit
from pankoff.combinator import combine
from pankoff.validators import String, Sized

sized_name = combine(String, Sized)
```

Now, we can actually use it to create instances:

```
@autoinit
class Person:
    name = sized_name(min_size=5)

guido = Person(name="Guido")
print(guido.name)  # Guido
```

Alternatively, you can create instance straight away:

```
@autoinit
class Person:
    name = combine(String, Sized, min_size=5)

guido = Person(name="Guido")
print(guido.name)  # Guido
```

5.2 Using “and” operator

Besides `combine()` function, you can chain validators using & operator, it’ll create a “raw” validator for you:

```
>>> raw_validator = Sized & String & Number & Type
>>> raw_validator
Combination of (Sized, String, Number, Type) validators
>>> validator = raw_validator(...)
>>> validator
CombinedValidator(Sized, String, Number, Type)
```

5.3 Using inheritance

Chaining uses inheritance mechanism under the hood, so you can do the following:

```
class MyCombinedValidator(String, Sized, ...):
    pass
```

CHAPTER

SIX

LIMITATIONS

As you may already know, Pankoff uses MRO a lot, because of that, you're not allowed to use `super().validate()` and `super().__setup__()`, instead, you should still subclass validators, but call directly to `__setup__` and *validate*, e.g: `Type.validate(...)`.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX