metaparams Documentation

Release 2.0.3

Daniel Rodriguez

Apr 09, 2019

Contents

Contents:

CHAPTER 1

Background

When working with classes in Python, class attributes can be used to have default values, which can be customized during initialization, such as in:

```
class A:
    value1 = 'value1'
    value2 = 'value2'

    def __init__(self, value1='value1', value2='value2'):
        self.value1 = self.value1
        self.value2 = self.value2
```

Which can be generalized with the use of `**kwargs`:

```
class A:
    value1 = 'value1'
    value2 = 'value2'

    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            if hasattr(self, k):
                setattr(self, k, v)
```

Case in which we have added a restriction to only consider `**kwargs` which have already been defined in the class. But even with this pattern, some things could have room for improvement:

- Avoid the manual coding in `__init__` (with or without `**kwargs`)
- Control the types which are passed
- Transform the values if needed
- Document the types when they are declared and not in the *docstring*

One can of course add type hints in the latest Python versions and document the parameters in the docstring, but:

- The type hints are just that . . . hints

- It seems better to document the parameter when it's defined.
- There is no way to know (except reading the docs, which sometimes do not exist) if the class needs that the caller provides a value

And one final caveat:

- Those attributes which are supposed to be configured via `__init__` are not separated from any other attributes in the instances

CHAPTER 2

The *params* pattern

The `metaparams` library offers therefore the following pattern:

```
import metaparams

class A(metaparams.ParamsBase):
    params = {
        # The option will be automatically prefixed with ``--`` in argparse
        'value1': {
            'value': 'value1',  # default value (can be skipped if required)
            'doc': 'This is value1',  # documentation (goes to docstring)
            'required': False,  # if required or not (also for argparse)
            'type': str,  # for type checking
            'transform': None,  # callable which transforms a str
            'argparse': True,  # if the option shall be added to argparse
            'group': None,  # a group name for argparse options grouping
            'choices': None,  # list of "choices" for argparse integration
            'alias': None,  # list of "alias" for argparse integration
                            # automatically prefixed with ``-``
        },
        'value2': {
            'required': True,
        },
        'value3': 'value3',
    }
```

We have provided a full definition for `value1`, a reduced one for `value2` and just the actual default value is provided for `value3`

- `value1` (complete example) gets documented, gets a default value, is marked as not *required*, must be of type `str` and will undergo no *transform* (`None` is given rather than a transformation function)

  Because `argparse` is `True` it will included in the `argparse` integration (if used)

  And it will be added to no `argpare` parsing group because `group` is `None`

- `value2` on the other hand is just marked as **required**. If not provided when the host class is instantiated an *Exception* will be raised

  Note: Notice that there is no need to provide a default value, because the caller has to actually provide a value.

- `value3` gets a default value. It will not be *required*, has no documentation, no specific type definition and no transform function.

One can now do the following:

```
a = A(value2=22, value3='this is my value')
print(a.params.value1)  # shorthand a.p.value1
print(a.params.value2)  # shorthand a.p.value2
print(a.params.value3)  # shorthand a.p.value3
```

which prints:

```
value1
22
this is my value
```

Notice that we **haven't defined ''__init__''** and yet `value2` and `value3` have received the values passed to the class instance. This because behind the scenes the following has happened:

- The `params` definition (a `dict`) has been turned dynamically into a subclass of `metaparams.Params`
- When `A` is instantiated into `a`, the `Params` subclass is also instantiated, intercepts the `**kwargs` and uses the values and is installed in the class instance.
- There is therefore a Class-Class and Instance-Instance duality in that:
  - `A`, a class, has a `params` attribute which is a subclass of `metaparams.Params`
  - `a`, an instance, has a `params` attribute which is an instance of `A.params`

  This is possible because in Python, attributes at instance level obscure the definition at class level (without overwriting it)

One can still define `__init__` and even have extra `**kwargs` passed to it:

```
import metaparams

class A(metaparams.ParamsBase):
    params = {
        # The option will be automatically prefixed with ``--`` in argparse
        'value1': {
            'value': 'value1',  # default value (can be skipped if required)
            'doc': 'This is value1',  # documentation (goes to docstring)
            'required': False,  # if required or not (also for argparse)
            'type': str,  # for type checking
            'transform': None,  # callable which transforms a str
            'argparse': True,  # if the option shall be added to argparse
            'group': None,  # a group name for argparse options grouping
            'choices': None,  # list of "choices" for argparse integration
            'alias': None,  # list of "alias" for argparse integration
                            # automatically prefixed with ``-``
        },
        'value2': {
            'required': True,
```

(continues on next page)

(continued from previous page)

```
        },
        'value3': 'value3',
    }

    def __init__(self, **kwargs):
        print('Extra **kwargs:', kwargs)
```

And then do:

```
a = A(value2=22, some_extra_kw='hello')
```

which prints:

```
Extra **kwargs: {'some_extra_kw': 'hello'}
```

2.1 Required parameters

Let's see what happens when a *required* parameter (`value2` in our examples) is not provided during instantiation:

```
a = A(value1='only value1')
```

And the error is:

```
...
    a = A(value1='only value1')
...
    raise ValueError(errmsg)
ValueError: Missing value for required parameter "value2" in parameters "__main___A_
↪params"
```

The raised exception is `ValueError`, because no value has been provided, is raised to let the caller know that `value2` has to be supplied.

Note: The name auto-magically assigned to the dynamically created parameters class tries to be descriptive and let us know where things are. In this case the name is `__main___A_params`, i.e.:

- Module `__main__`
- Inside Class `A`

A complete *traceback* will of course also point out in which file and line the error has kicked in

---

2.2 Type Checking

We already have a *type* specified for `value1` which is `str`. Let's see what happens if we pass a `float`:

```
a = A(value2=45, value1=22.0)
```

The result:

```
...
    a = A(value2=45, value1=22.0)
...
    raise TypeError(errmsg)
TypeError: Wrong type "<class 'float'>" for param "value1" with type <class 'str'> in␣
↪parameters "__main___A_params"
```

A `TypeError` (obviously) is raised if the passed value is not of the type defined for the parameter.

2.3 Transformation

In the examples above we have only shown the definition with:

```
transform=None
```

as one of the components of a parameter. `None` is there to indicate that nothing has to be done. Let's change that to see how things work:

```
import metaparams

class A(metaparams.ParamsBase):
    params = {
        'value1': {
            'value': 'value1',
            'doc': 'This is value1',
            'required': False,
            'type': str,
            'transform': lambda x: x.upper(),
        },
        'value2': {
            'required': True,
        },
        'value3': 'value3',
    }

a = A(value1='hello', value2='no value 2')  # supply required value2
print('a.params.value1:', a.params.value1)
```

In the `transform` we can be sure that we can apply `x.upper()` because we are requiring that the type be `str`.

The outcome:

```
a.params.value1: HELLO
```

which shows our input value `hello` in uppercase form.

2.4 Auto-Documentation

One of the reasons to go into this, is to document the parameter when it is being defined. In the above examples this is being done for `value1`. And the magic behind the scenes makes it possible that the following is true:

```
print(A.__doc__)  # print the docstring
```

which results in the following output:

```
Args

  - value1: (default: value1) (required: False) (type: <class 'str'>) (transform:␣
↪None)
    This is value1

  - value2: (default: None) (required: True) (type: None) (transform: None)

  - value3: (default: value3) (required: False) (type: None) (transform: None)
```

The parameters have auto-documented themselves in the host class, which means that they will for example be part of auto-generated documentation when using, for example, *Sphinx*

Where the presence of a `bool` or a `str` will determine if the third value is the doc string or the `required` indication.

CHAPTER 3

argparse integration

The *params* pattern can be used to dynamically generate command line options with the `argparse` module, i.e.: adding new definitions to the `params` of a class will add new command line switches to match those definitions.

Generation of the command line switches

```
import argparse
from metaparams import ParamsBase

parser = argparse.ArgumentParser(
    formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    description=(
        'Some script with auto-generated command line switches '
    )
)

class A(ParamsBase):
  params = {
      'value1': {
          'value': 'value1',
          'doc': 'This is value1',
          'required': False,
          'type': str,
          'transform': None,
          'argparse': True,
          'group': None,
          'choices': None,
      },
      'value2': {
          'required': True,
      },
      'value3': 'value3',
  }
```

(continues on next page)

(continued from previous page)

```
# The integration of the params in the command line switches
A.params._argparse(parser)
```

Use of the paramters for instantiation

```
args = parser.parse_args()

# The integration of command line switches values for instantiation
a = A(**A.params._parseargs(args))
```

Or even simpler:

```
args = parser.parse_args()

# The integration of command line switches values for instantiation
a = A.params._create(args)
```

In the example above for `value1` three entries are shown which specifically influence the `argparse` integration

- `argparse`: if `True` (default), the parameter is included in the integration
- `group`: if not `None`, the passed name is used to create a parsing group. In this ways, several parameters can be logically grouped.
- `choices`: if not `None`, it must be an iterable of options from which it can be chosen and will be passed to `argparse`

CHAPTER 4

The API

The parameter values, as shown above, can be accessed with `.` (dot) notation, but there is a lot more that can be done. All methods have been prefixed with a leading underscore (_) to avoid collision with parameter names the end user could choose.

Notice the following relationship *class-class* and *instance-instance*

- `A.params` - Here `A` is the host class holding parameters, and `A.params` is a parameter class (dynamically generated)
- `a.params` - Here `a` is an instance of `A` and `a.params` is an instance of `A.params`

4.1 Customization

Per default parameters are defined with the name `params` in the host class:

```
class A(Paramsbase):

    params = {
        ...
    }
```

And are reachable in the instance of the host class as either:

```
a = A()

a.params

# 1st letter of the name params. If the name had a leading underscore
# such as _params, the shortcut would be _p
a.p
```

The name `params` and the creation of the shorthand `p` can be customized when `Paramsbase` is subclassed using keyword arguments for `Python >= 3.6`:

```
from metaparams import MetaParams

class A_poroms(metaclass=MetaParams, _pname='poroms', _pshort=False)
    poroms = {
        ...
    }
```

Note: Notice how instead of subclassing from `ParamsBase`, when changing the *name* of the params, this has to be specified using `metaclass=MetaParams`

This is because `ParamsBase` has already defined a fixed name `params` for the declaration and this is already set for any subclass. The reason being that class attributes (not to be confused with instance attributes) cannot be deleted. Overriding the name for the params declaration would lead to multiplicity of params class attributes in the host class

If using `Python < 3.6`, use the decorator, because no keyword arguments are supported durint class creation:

```
from metaparams import metaparams

@metaparams(_pname='poroms', _pshort=False)
class A_poroms:
    poroms = {
        ...
    }
```

In this case:

- The parameters are defined and are reachable under the name `poroms`
- No shortcut `p` is created

Another example:

```
class A_poroms(metaclass=MetaParams, _pname='_xarams')
    _xarams = {
        ...
    }
```

or:

```
from metaparams import metaparams

@metaparams(_pname='_xarams')
class A_poroms:
    _xarams = {
        ...
    }
```

And now

- Parameters are reachable under the name `_xarams`
- A shortcut will be created with `_x`

4.2 The features

A parameter can be canonically defined (as already seen above) in 3 different ways.

- Using a `name: value` entry in the `params` dictionary. Such as:

```
params = {
    'myparam1': 'myvalue1',
}
```

  This will be internally translated to a full `dict` entry as specified below

- Using a complete `dict` entry for the param:

```
params = {
    'myparam1': {
        # Default value for the parameter (default: None)
        'value': 'myvalue1',
        # if param is required for host instantiation (default: False)
        'required': False,
        # Document the param (default: '')
        'doc': 'my documentation',
         # Check if given type is passed (default: None)
        'type': str,
        # Transform given parameter with function (default: None)
        'transform': lambda x: x.upper(),
        # If params should be part of argparse integration (default: True)
        'argparse': True,
}
```

Note: If the name of a parameter ends with _ it will be automatically excluded from `argparse` integration

4.2.1 Using iterables

The *params* can also be specified as iterables (*list/tuple*) of iterables (*list/tuple*) with the following notation (elements in between square brackets are optional:

```
params = (
    (name, value, [doc, [required, [type, [transform, [argparse]]]]]),
    (name1, value1, [doc1, [required1, [type1, [transform1, [argparse1]]]]]),
    ...
)
```

Or:

```
params = (
    (name, value, [required, [doc, [type, [transform, [argparse]]]]]),
    (name1, value1, [required1, [doc1, [type1, [transform1, [argparse1]]]]]),
    ...
)
```

Note: This is provided as a backwards compatibility to the original supported declaration in the previous versions of `metaparams`. It is actually recommended **not** to use it.

4.3 Customization

The following keyword arguments are accepted by a class definition (Python >= 3.6) or by the decorator.

- `_pname` (default: `params`)

  This defines the main name for the declaration and attribute for accessing the declared parameters.

  Note: If one of the base classes (such as `ParamsBase`) has already set this name, it cannot be overridden by subclasses.

- `_pshort` (default: `True`)

  Provide a 1-letter shorthand of the name defined in `_pname` in the instance of the host class holding the params. For example: `params` will also be installed as `p`.

  If the defined name has a leading _ (underscore) it will respected and the next character will be also taken. For example: `_myparams` will be shortened to `_m`

- `_pinst` (default: `False`)

  Only valid in combination with `_pshort = True`. Install an instance attribute using the shortened notation, an _ (underscore) and the name of the parameter.

  If a params declaration looks like this:

```
class A(ParamsBase, _pinst=True):
    params = {
        'myparam': True,
    }
```

  The following will be true in an instance of `A`:

```
a = A()

assert(a.params.myparam == a.p_myparam)
```

4.4 The methods

This is a list of the supported methods and features:

- Operator `[name]` - To access the current parameter value applied to the class or instance of the parameters
- `len(self.params)` gives the number of defined parameters
- Iteration is supported: `[x for x in self.params]` or `iter(self.params)` will give you access to the parameter names

  The pattern can be applied to the class or the instance of the parameters.

Defaults (can be applied to the parameters class or instance)

- `def _defkwargs()` - returns a `dict` with *name/value* pairs where the values are the default values and not the current ones
- `def _defitems()` - returns an iterable with *name/value* pairs where the values are the default values and not the current ones
- `def _defkeys()` - returns an iterable with the parameter *names* This is really an oxymoron because the names cannot be changed.
- **`def _defvalues()` - returns an iterable with the default** parameter *values*
- `def _defvalue(name)` - returns the default value for *name*

- `def _isrequired(name)` - returns `True` if the parameter name has to be specified during the instantiation of host class instances
- `def _doc(name=None)` - returns the doc string for *name* if given or else return the autogenerated docstring for all parameters which is automatically added to the host class
- `def _get(name, prop)` - returns a specific property `prop` for the param `name`. Example: to get the doc string use:

```
``_get(param_name, 'doc')``
```

Current values (can be applied to the parameters instance)

- `def _update(x)` - Update the value of the parameters with a dict-like object or an iterable of pairs *name/value*
- `def _update(**kwargs)` - Update the value of the parameters with the given keyword arguments
- `def _reset(name=None)` - Reset either an individual parameter if *name* to its default value is given or reset all parameters to the default values if no *name* is provided
- `def _kwargs()` - returns a `dict` with *name/value* pairs where the values are the current ones
- `def _items()` - returns an iterable with *name/value* pairs where the values are the current ones
- `def _keys()` - returns an iterable with the parameter *names*
- `def _values()` - returns an iterable with the parameter *values*
- `def _value(name)` - returns the current value for *name*
- `def _isdefault(name)` - returns `True` if the value is the default one

Argparse integration (intended to be used as classmethod)

- `def _argparse(parser, group=None, skip=True, minus=True)`

  Integrate params in the given `parser`

  - `group`: If a string is passed, the params will be put inside a group with that name
  - `skip`: If `True`, any param with a name ending in _ will be ignored
  - `minus`: If `True`, underscores will be translated to - (minus) signs for the options in `argparse` (the module does automatically translate them backwards to _ in member attributes)

- `def _parseargs(args, skip=True)`

  Use the already parsed `args` to assign value to the params

  - `skip`: If `True`, any param with a name ending in _ will be ignored

- `def _create(args, skip=True)`

  Using the given *argparse* `args` object create an instance of the host class holding this params

  - `skip`: If `True`, any param with a name ending in _ will be ignored

CHAPTER 5

Indices and tables

- genindex
- modindex
- search