kwargshelper

Release 2.7.1

:Barry-Thomas-Paul: Moss

Nov 21, 2021

CONTENTS

CHAPTER

ONE

KWARGSHELPER

A python package for working with **kwargs**.

Allows for validation of args passed in via `**kwargs` in various ways. Such as type checking, rules checking, error handeling.

Many built in rules that make validation of input simple and effective. Easily create and add new rules.

Various callbacks to hook each **kwarg** with rich set of options for fine control.

1.1 Docs

Read the docs here

1.2 Installation

You can install the Version Class from PyPI

```
pip install kwargshelper
```

1.3 KwargsHelper Class

Helper class for working with python `**kwargs` in a class constructor

Assigns values of `**kwargs` to an exising class with type checking and rules

Parse kwargs with suport for rules that can be extended that validate any arg of kwargs. Type checking of any type.

Callback function for before update that includes a Cancel Option.

Many other options avaliable for more complex usage.

1.4 KwArg Class

Helper class for working with python `**kwargs` in a method/function Wrapper for `KwargsHelper` Class.

Assigns values of `**kwargs` to itself with validation

Parse kwargs with suport for rules that can be extended that validate any arg of kwargs. Type checking of any type.

Callback function for before update that includes a Cancel Option.

Many other options avaliable for more complex usage.

1.5 Decorator Classes

Decorators that can applied to function that validate arguments.

Decorators for Type checking and Rule testing is built in.

The following example ensures all function args are a positive `int` or a positive `float`.

```
from kwhelp.decorator import RuleCheckAny
import kwhelp.rules as rules

@RuleCheckAny(rules.RuleIntPositive, rules.RuleFloatPositive)
def speed_msg(speed, limit, **kwargs) -> str:
    if limit > speed:
        msg = f"Current speed is '{speed}'. You may go faster as the limit is '{limit}'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. You are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and you are currenlty going '{speed}
↪'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'"
    return msg
```

CHAPTER

TWO

TOC

2.1 Usage

2.1.1 KwargsHelper

Assign Default Value

Default value can be assigned by adding `default` args to *`assign()`* method.

```
from kwhelp import KwargsHelper
import kwhelp.rules as rules

class MyClass:
    def __init__(self, **kwargs):
        self._duration = "Long"
        kw = KwargsHelper(self, {**kwargs})
        kw.assign(key="speed", require=True, rules_all=[rules.RuleIntPositive])
        kw.assign(key="unit", default="MPH", rules_all=[rules.RuleStrNotNullOrEmpty])
        kw.assign(key="duration", default=self._duration,
            rules=[rules.RuleStrNotNullOrEmpty])
```

```
>>> myclass = MyClass(speed=123, unit="KPH")
>>> print(myclass._speed)
123
>>> print(myclass._unit)
KPH
>>> print(myclass._duration)
Long
```

```
>>> myclass = MyClass(speed=123, duration="Short")
>>> print(myclass._speed)
123
>>> print(myclass._unit)
MPH
>>> print(myclass._duration)
Short
```

Assign Field Value

Field value can be assigned by adding `field` arg to *assign()* method. By default field values are key values with _ prepended. If `key="speed"` then `field` defaults to `_speed` and thus attribute with the name of `_speed` is a assigned to class instance. Setting `field` arg overrides attribute name that is assigned to current class instance.

```
from kwhelp import KwargsHelper
import kwhelp.rules as rules

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(self, {**kwargs})
        kw.assign(key="speed", field="race_speed", require=True,
            rules_all=[rules.RuleIntPositive])
        kw.assign(key="unit", field="unit", default="MPH",
            rules_all=[rules.RuleStrNotNullOrEmpty])
```

```
>>> myclass = MyClass(speed=123)
>>> print(myclass.race_speed)
123
>>> print(myclass.unit)
MPH
```

```
>>> myclass = MyClass(speed=123, unit="KPH")
>>> print(myclass._speed)
123
>>> print(myclass._unit)
KPH
```

Note: Default prefix for all field can be set by setting `field_prefix` arg of *KwargsHelper* constructor.

Assign Require Arg

Required args can be done by adding `require` args to *assign()* method. If a required args is missing then a `ValueError` will be raised.

```
from kwhelp import KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(self, {**kwargs})
        kw.assign(key="speed", require=True)
        kw.assign(key="unit")
```

```
>>> myclass = MyClass(speed=123, unit="KPH")
>>> print(myclass._speed)
123
>>> print(myclass._unit)
KPH
```

```
>>> myclass = MyClass(unit="KPH")
ValueError: MyClass arg 'speed' is required
```

Assign Rule Checking

Rule checking can be done by adding `rules_all` and/or `rules_any` to *`assign()`* method. Rule checking ensures a value of `**kwargs` values matches all rules before assign to current instance of class.

All rules

In the following example attribute `speed` can be a positive `float` or `int` zero. All other values will result in an error. Arg `rules_all` of *`assign()`* method validates as `True` only if all *`IRule`* match. Trying to assign any other type or value results in an error.

Custom Rule for a maximum `int` value of 100

```
import kwhelp.rules as rules

class RuleIntMax100(rules.IRule):
    def validate(self) -> bool:
        if self.field_value > 100:
            if self.raise_errors:
                raise ValueError(
                    f"Arg error: '{self.key}' must be equal or less than 100")
            return False
        return True
```

```
from kwhelp import rules, KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix="")
        kw.assign(key="speed", rules_all=[
            rules.RuleIntPositive,
            RuleIntMax100
        ])
```

Assign `int`.

```
>>> myclass = MyClass(speed = 12)
>>> print(myclass.speed)
12
```

Assign `int` greater then 100.

```
>>> myclass = MyClass(speed = 126)
kwhelp.exceptions.RuleError: RuleError: Argument: 'speed' failed validation.
Rule 'RuleIntMax100' Failed validation.
Expected all of the following rules to match: RuleIntPositive, RuleIntMax100.
Inner Error Message: ValueError: Arg error: 'speed' must be equal or less than 100
```

Assign `int` negative.

```
>>> myclass = MyClass(speed = -3)
kwhelp.exceptions.RuleError: RuleError: Argument: 'speed' failed validation.
Rule 'RuleIntPositive' Failed validation.
Expected all of the following rules to match: RuleIntPositive, RuleIntMax100.
Inner Error Message: ValueError: Arg error: 'speed' must be a positive int value
```

Assign `float`.

```
>>> myclass = MyClass(speed = 22.4)
kwhelp.exceptions.RuleError: RuleError: Argument: 'speed' failed validation.
Rule 'RuleIntPositive' Failed validation.
Expected all of the following rules to match: RuleIntPositive, RuleIntMax100.
Inner Error Message: TypeError: Argument Error: 'speed' is expecting type of 'int'. Got␣
↪type of 'float'
```

Any rules

In the following example attribute `speed` can be a positive `float` or `int` zero. All other values will result in an error. Arg `rules_all` of *assign()* method validates as `True` only if all *IRule* match. Trying to assign any other type or value results in an error.

```
from kwhelp import rules, KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix="")
        kw.assign(key="speed", rules_any=[
            rules.RuleFloatPositive,
            rules.RuleIntZero
        ])
```

Assign `int`.

```
>>> myclass = MyClass(speed = 123.55)
>>> print(myclass.speed)
123.55
```

Assign `int` zero.

```
>>> myclass = MyClass(speed = 0)
>>> print(myclass.speed)
0
```

Assign `float` zero.

```
>>> myclass = MyClass(speed = 0.0)
>>> print(myclass.speed)
0.0
```

Assign `int` negative.

```
>>> myclass = MyClass(speed = -123)
kwhelp.exceptions.RuleError: RuleError: Argument: 'speed' failed validation.
Rule 'RuleFloatPositive' Failed validation.
Expected at least one of the following rules to match: RuleFloatPositive, RuleIntZero.
Inner Error Message: TypeError: Argument Error: 'speed' is expecting type of 'float'.␣
↪Got type of 'int'
```

Assign `str`.

```
>>> myclass = MyClass(speed="a")
kwhelp.exceptions.RuleError: RuleError: Argument: 'speed' failed validation.
Rule 'RuleFloatPositive' Failed validation.
Expected at least one of the following rules to match: RuleFloatPositive, RuleIntZero.
Inner Error Message: TypeError: Argument Error: 'speed' is expecting type of 'float'.␣
↪Got type of 'str'
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*
- *RuleIntZero*
- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*

- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

Assign Type Checking

Type checking can be done by adding `types` to *assign()* method. Type checking ensures the type of `**kwargs` values that are assigned to attributes of current instance of class.

In the following example attribute `speed` can be of type `float` or of type `int`. All other type will result in an error.

```
from kwhelp import KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(self, {**kwargs})
        kw.assign(key="speed", types=[int, float])
```

```
>>> myclass = MyClass(speed=123)
>>> print(myclass._speed)
123
```

```
>>> myclass = MyClass(speed=19.8)
>>> print(myclass._speed)
19.8
```

```
>>> myclass = MyClass(speed="a")
TypeError: MyClass arg 'speed' is expected to be of '<class 'float'> | <class 'int'>'␣
↪but got 'str'
```

Auto_assign Usage

KwargsHelper.auto_assign() method automatically assigns all key-value pairs to class.

Assign with no args

Assigns all of the key, value pairs passed into constructor to class instance, unless the event is canceled in *BeforeAssignAutoEventArgs* via *KwargsHelper.add_handler_before_assign_auto()* callback.

```
from kwhelp import KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix='')
        kw.auto_assign()
```

```
>>> myclass = MyClass(speed=123, msg="Hello World")
>>> print(myclass.speed)
123
>>> print(myclass.msg)
Hello World
```

Assign with type checking

By setting the optional arg `types` of *KwargsHelper.auto_assign()* it is possible to restrict what types can be assigned.

Assigns all of the key, value pairs passed into constructor to class instance if the value is of type `int` or `float`. Any other type results in an error.

```
from kwhelp import KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix='')
        kw.auto_assign(types=[int, float])
```

Any arg assigned with a value of type `int` or `float` is automatically assigned.

```
>>> myclass = MyClass(speed = 123, distance = 557.46)
>>> print(myclass.speed)
123
>>> print(myclass.distance)
557.46
```

Assigning an arg with a value that is not of type `int` or `float` result is an error.

```
>>> myclass = MyClass(speed = "Fast", distance = 557.46)
TypeError: MyClass arg 'speed' is expected to be of '<class 'int'> | <class 'float'>'␣
↪but got 'str'
```

Assign with Rule checking

All Rules

By setting the optional arg `rules_all` of *KwargsHelper.auto_assign()* it is possible to set rules that must all be met for key, values to be successfully assigned.

```
from kwhelp import KwargsHelper
import kwhelp.rules as rules

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix='')
        kw.auto_assign(rules_all=[rules.RuleNotNone, rules.RuleFloatPositive])
```

Any arg assigned with a value of `float` is automatically assigned.

```
>>> myclass = MyClass(speed = 99.999, distance = 557.46)
>>> print(myclass.speed)
99.999
>>> print(myclass.distance)
557.46
```

Assigning an arg with a value that is not `float` result is an error.

```
>>> myclass = MyClass(speed = 99.999, distance = 55)
kwhelp.exceptions.RuleError: RuleError: Argument: 'distance' failed validation.
Rule 'RuleFloatPositive' Failed validation.
Expected all of the following rules to match: RuleNotNone, RuleFloatPositive.
Inner Error Message: TypeError: Argument Error: 'distance' is expecting type of 'float'.␣
↪Got type of 'int'
```

Assigning an arg with a value that is a negative `float` result is an error.

```
>>> myclass = MyClass(speed = 99.999, distance = -128.09)
kwhelp.exceptions.RuleError: RuleError: Argument: 'distance' failed validation.
Rule 'RuleFloatPositive' Failed validation.
Expected all of the following rules to match: RuleNotNone, RuleFloatPositive.
Inner Error Message: ValueError: Arg error: 'distance' must be a positive float value
```

Any Rules

By setting the optional arg `rules_any` of *KwargsHelper.auto_assign()* it is possible to set rules that must have at least one match for key, values to be successfully assigned.

```
from kwhelp import KwargsHelper
import kwhelp.rules as rules

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix='')
        kw.auto_assign(rules_any=[rules.RuleIntPositive, rules.RuleFloatPositive])
```

Any arg assigned with a value of `int` or `float` is automatically assigned.

```
>>> myclass = MyClass(speed = 99.999, distance = 558)
>>> print(myclass.speed)
99.999
>>> print(myclass.distance)
558
```

Assigning an arg with a value that is not `int` or `float` result is an error.

```
>>> myclass = MyClass(speed = 'Fast', distance = 55)
kwhelp.exceptions.RuleError: RuleError: Argument: 'speed' failed validation.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
Inner Error Message: TypeError: Argument Error: 'speed' is expecting type of 'int'. Got␣
↪type of 'str'
```

Assigning an arg with a value that is a negative `int` result is an error.

```
>>> myclass = MyClass(speed = 99.999, distance = -35)
kwhelp.exceptions.RuleError: RuleError: Argument: 'distance' failed validation.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
Inner Error Message: ValueError: Arg error: 'distance' must be a positive int value
```

See also:

- *KwargsHelper.auto_assign()*
- *KwargsHelper.add_handler_before_assign_auto()*
- *KwargsHelper*
- *BeforeAssignAutoEventArgs*
- *Auto_assign Callback*
- *Assign Rule Checking*

Auto_assign Callback

KwargsHelper.auto_assign() method automatically assigns all key-value pairs to class.

Assigns all of the key, value pairs passed into constructor to class instance, unless the event is canceled in *BeforeAssignAutoEventArgs* via *KwargsHelper.add_handler_before_assign_auto()* callback.

`MyClass._arg_before_cb()` callback method reads conditions before attribue and value are assigned. If required conditions are not met then cancel can be set.

```
from kwhelp import KwargsHelper, AfterAssignAutoEventArgs, BeforeAssignAutoEventArgs

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix="")
        kw.add_handler_before_assign_auto(self._arg_before_cb)
        kw.add_handler_after_assign_auto(self._arg_after_cb)
        kw.auto_assign()

    def _arg_before_cb(self, helper: KwargsHelper,
                       args: BeforeAssignAutoEventArgs) -> None:
        # callback function before value assigned to attribute
        if args.key == 'loop_count' and args.field_value < 0:
            # cancel will raise CancelEventError unless
            # KwargsHelper constructor has cancel_error=False
            args.cancel = True
        if args.key == 'name' and args.field_value == 'unknown':
            args.field_value = 'None'

    def _arg_after_cb(self, helper: KwargsHelper,
                      args: AfterAssignAutoEventArgs) -> None:
        # callback function after value assigned to attribute
        if args.key == 'name' and args.field_value == 'unknown':
```

(continues on next page)

(continued from previous page)

```
            raise ValueError(
                f"{args.key} This should never happen. value was suppose to be reassigned
↪")
```

In the following case key, value args are automatically assigned to class.

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=3)
>>> print(my_class.exporter)
json
>>> print(my_class.file_name)
data.json
>>> print(my_class.loop_count)
3
```

In the following case `my_class.name` value is changed from `"unknown"` to `"None"` in callback `MyClass._arg_before_cb()` method.

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=3, name=
↪"unknown")
>>> print(my_class.exporter)
json
>>> print(my_class.file_name)
data.json
>>> print(my_class.loop_count)
3
>>> print(my_class.name)
None
```

In the following case `loop_count` is a negative number which triggers a *CancelEventError* in `MyClass._arg_before_cb()`.

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=-1)
kwhelp.CancelEventError: KwargsHelper.auto_assign() canceled in
↪'BeforeAssignBlindEventArgs'
```

Note: If *KwargsHelper constructor* has `cancel_error` set to `False` then no error will be raised when *BeforeAssignAutoEventArgs.cancel* is set to `True`.

See also:

- *KwargsHelper*
- *KwargsHelper.add_handler_after_assign()*
- *KwargsHelper.add_handler_after_assign_auto()*
- *KwargsHelper.add_handler_before_assign()*
- *KwargsHelper.add_handler_before_assign_auto()*
- *AfterAssignAutoEventArgs*
- *BeforeAssignAutoEventArgs*
- *AssignBuilder*

- *Auto_assign Usage*
- *Callback Usage*

Callback Usage

Callbacks can be used to hook some of *KwargsHelper* events.

This class as two callback methods. `MyClass._arg_before_cb()` method is called before attribute assignment. `MyClass._arg_after_cb()` method is called after attribute assignment.

```
from kwhelp import KwargsHelper, AfterAssignEventArgs, BeforeAssignEventArgs,␣
↪AssignBuilder

class MyClass:
    def __init__(self, **kwargs):
        self._loop_count = -1
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs})
        ab = AssignBuilder()
        kw.add_handler_before_assign(self._arg_before_cb)
        kw.add_handler_after_assign(self._arg_after_cb)
        ab.append(key='exporter', require=True, types=[str])
        ab.append(key='name', require=True, types=[str],
                default='unknown')
        ab.append(key='file_name', require=True, types=[str])
        ab.append(key='loop_count', types=[int],
                default=self._loop_count)
        result = True
        # by default assign will raise errors if conditions are not met.
        for arg in ab:
            result = kw.assign_helper(arg)
            if result == False:
                break
        if result == False:
            raise ValueError("Error parsing kwargs")

    def _arg_before_cb(self, helper: KwargsHelper,
                       args: BeforeAssignEventArgs) -> None:
        # callback function before value assigned to attribute
        if args.key == 'loop_count' and args.field_value < 0:
            # cancel will raise CancelEventError unless
            # KwargsHelper constructor has cancel_error=False
            args.cancel = True
        if args.key == 'name' and args.field_value == 'unknown':
            args.field_value = 'None'

    def _arg_after_cb(self, helper: KwargsHelper,
                      args: AfterAssignEventArgs) -> None:
        # callback function after value assigned to attribute
        if args.key == 'name' and args.field_value == 'unknown':
            raise ValueError(
                f"{args.key} This should never happen. value was suppose to be reassigned
↪")
```

(continues on next page)

(continued from previous page)

```
    @property
    def exporter(self) -> str:
        return self._exporter

    @property
    def file_name(self) -> str:
        return self._file_name

    @property
    def name(self) -> str:
        return self._name

    @property
    def loop_count(self) -> int:
        return self._loop_count
```

In the following case `my_class.name` value is changed from `"unknown"` to `"None"` in callback `MyClass._arg_before_cb()` method.

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=3)
>>> print(my_class.exporter)
json
>>> print(my_class.file_name)
data.json
>>> print(my_class.name)
None
>>> print(my_class.loop_count)
3
```

In the following case `loop_count` is a negative number which triggers a *CancelEventError* in `MyClass._arg_before_cb()`.

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=-1)
kwhelp.CancelEventError: KwargsHelper.assign() canceled in 'BeforeAssignEventArgs'
```

Note: If *KwargsHelper constructor* has `cancel_error` set to `False` or *KwargsHelper.cancel_error* property is set to `False` then no error will be raised when *BeforeAssignEventArgs.cancel* is set to `True`.

If no error is raised then *AfterAssignEventArgs.canceled* can be used to check if an event was canceled in *BeforeAssignEventArgs*

See also:

- *KwargsHelper*
- *KwargsHelper.add_handler_after_assign()*
- *KwargsHelper.add_handler_after_assign_auto()*
- *KwargsHelper.add_handler_before_assign()*
- *KwargsHelper.add_handler_before_assign_auto()*
- *AfterAssignEventArgs*
- *BeforeAssignEventArgs*

- *AssignBuilder*
- *Auto_assign Callback*

Basic Usage

Processing **kwargs in a class.

KwargsHelper class assigns attributes to existing class instance if they are missing. Each key in **kwargs is transformed into an attribute name and that attribute name is assigned to current instance of class if it does not already exist. Be default attribute names is the key name with _ appended. See *Figure 2*

**kwargs values are assigned to attribue that match keys.

Use *KwargsHelper* class to process **kwargs in a class.

Listing 1: Figure 1

```
from kwhelp import KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(self, {**kwargs})
        kw.auto_assign()

>>> myclass = MyClass(speed=123)
>>> print(myclass._speed)
123
```

In the following example **attribute** names are transformed to match the **key** name. This is done by setting field_prefix to empty string in constructor. See Also: *KwargsHelper.field_prefix*

Listing 2: Figure 2

```
from kwhelp import KwargsHelper

class MyClass:
    def __init__(self, **kwargs):
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs}, field_prefix='')
        kw.auto_assign()
```

```
>>> myclass = MyClass(speed=123)
>>> print(myclass.speed)
123
```

See also:

- *Example Simple Usage*
- *KwargsHelper Class*

2.1.2 KwArg

Auto assign

Auto assign of key, value parirs can be accomplish by calling *KwargsHelper.auto_assign()* of *KwArg.kwargs_helper* property.

```
from kwhelp import KwArg

def sum_of(**kwargs) -> str:
    kw = KwArg(**kwargs)
    kw.kwargs_helper.auto_assign(types=[int])
    result = 0
    for key in kw.kwargs_helper.kw_args:
        result = result + kw.__dict__[key]
    return result
```

Assing values of type `int`.

```
>>> result = sum_of(first_qtr=199, second_qtr=201)
>>> print(result)
400
```

Assigning value not of type `int` results in an error.

```
>>> result = sum_of(first_qtr=199.78, second_qtr=201)
TypeError: KwArg arg 'first_qtr' is expected to be of '<class 'int'>' but got 'float'
```

See also:

- *KwArg.kwargs_helper*
- *KwargsHelper auto_assign Usage*

Callback Usage

Callbacks can be used to hook some of *KwArg* events.

By using the *KwArg.kwargs_helper* property which is an instance of *KwargsHelper* it is possible to use callbacks.

This function has two nested callback functions. `arg_before_cb()` function is called before key, value is assigned to *KwArg* instance. `arg_after_cb()` function is called after key, value is assigned to *KwArg* instance.

```
from kwhelp import AfterAssignEventArgs, BeforeAssignEventArgs, KwargsHelper, KwArg

def my_method(**kwargs) -> str:
    def arg_before_cb(helper: KwargsHelper,
                      args: BeforeAssignEventArgs) -> None:
        # callback function before value assigned to KwArg instance
        if args.key == 'msg':
            if args.field_value == "cancel":
                # cancel will raise CancelEventError unless
                # kw.kwargs_helper.cancel_error = False
                args.cancel = True
            elif args.field_value == "":
```

(continues on next page)

(continued from previous page)

```
                args.field_value = "Value:"
        if args.key == 'name' and args.field_value == 'unknown':
            args.field_value = 'None'
    def arg_after_cb(helper: KwargsHelper,
                     args: AfterAssignEventArgs) -> None:
        # callback function after value assigned to KwArg instance
        if args.key == 'msg' and args.field_value == "":
            raise ValueError(
                f"{args.key} This should never happen. value was suppose to be reassigned
↪")
    # main function
    kw = KwArg(**kwargs)
    # assign Callbacks
    kw.kwargs_helper.add_handler_before_assign(arg_before_cb)
    kw.kwargs_helper.add_handler_after_assign(arg_after_cb)
    # assign args
    kw.kw_assign(key='first', require=True, types=[int])
    kw.kw_assign(key='second', require=True, types=[int])
    kw.kw_assign(key='msg', types=[str], default='Result:')
    kw.kw_assign(key='end', types=[str])
    _result = kw.first + kw.second
    if kw.is_attribute_exist('end'):
        return_msg = f'{kw.msg} {_result}{kw.end}'
    else:
        return_msg = f'{kw.msg} {_result}'
    return return_msg
```

Call back does not make changes to the following result.

```
>>> result = my_method(first = 10, second = 22)
>>> print(result)
Result: 32
```

Call back changes `msg` from empty string to `Value:`

```
>>> result = my_method(first = 10, second = 22, msg = "")
>>> print(result)
Value: 32
```

Call back raises `CancelEventError` when `msg` equals `cancel`.

```
>>> result = my_method(first = 10, second = 22, msg = "cancel")
kwhelp.CancelEventError: KwargsHelper.assign() canceled in 'BeforeAssignEventArgs'
```

Note: If *KwargsHelper.cancel_error* property is set to `False` then no error will be raised when *BeforeAssignEventArgs.cancel* is set to `True`.

If no error is raised then *AfterAssignEventArgs.canceled* can be used to check if an event was canceled in *BeforeAssignEventArgs*

See also:

- *KwArg*

- *KwArg.kwargs_helper*
- *KwargsHelper*
- *KwargsHelper.add_handler_after_assign()*
- *KwargsHelper.add_handler_after_assign_auto()*
- *KwargsHelper.add_handler_before_assign()*
- *KwargsHelper.add_handler_before_assign_auto()*
- *AfterAssignEventArgs*
- *BeforeAssignEventArgs*
- *KwargsHelper Callback Usage*

Kw_assign Default Value

Default value can be assigned by adding `default` args to *kw_assign()* method.

In the following `msg` arg is assigned a default of `Result:`

```
from kwhelp import KwArg

def my_method(**kwargs) -> str:
    kw = KwArg(**kwargs)
    # assign args
    kw.kw_assign(key='first', require=True, types=[int])
    kw.kw_assign(key='second', require=True, types=[int])
    kw.kw_assign(key='msg', types=[str], default='Result:')
    kw.kw_assign(key='end', types=[str])
    _result = kw.first + kw.second
    if kw.is_attribute_exist('end'):
        return_msg = f'{kw.msg} {_result}{kw.end}'
    else:
        return_msg = f'{kw.msg} {_result}'
    return return_msg
```

Method result contains prefix of `msg` default value

```
>>> result = my_method(first=10, second=22, end="!")
>>> print(result)
Result: 32!
```

Method result contains `Tally` as `msg` default is now overridden by assiging value.

```
>>> result = my_method(first=10, second=22, end="!", msg="Tally")
>>> print(result)
Tally 32!
```

Method will raise an error as `msg` must be of type `str` as defined by `types`

```
>>> result = my_method(first=10, second=22, end="!", msg=-34)
TypeError: KwArg arg 'msg' is expected to be of '<class 'str'>' but got 'int'
```

Kw_assign field

Field value can be assigned by adding `field` arg to *kw_assign()* method. Sometimes it may be necessary to pass an arg with a name but change it in *KwArg* instance.

In the following `is_key_existing` is assigned to `is_key_exist` of *KwArg* instance. This avoids a *ReservedAttributeError* because `is_key_existing` is a reserved keyword of *KwArg*.

```
from kwhelp import KwArg

def is_key(**kwargs) -> str:
    keys = ('one','two', 'four', 'eight')
    kw = KwArg(**kwargs)
    kw.kw_assign(key='is_key_existing', field='is_key_exist', require=True, types=[str])
    if kw.is_key_exist in keys:
        return True
    return False
```

```
>>> result = is_key(is_key_existing="one")
>>> print(result)
True
>>> result = is_key(is_key_existing="three")
>>> print(result)
False
```

Kw_assign Require Arg

Values can be required by adding `require` args to *kw_assign()* method.

In the following `first` and `second` args are required

```
from kwhelp import KwArg

def my_method(**kwargs) -> str:
    kw = KwArg(**kwargs)
    # assign args
    kw.kw_assign(key='first', require=True, types=[int])
    kw.kw_assign(key='second', require=True, types=[int])
    kw.kw_assign(key='msg', types=[str], default='Result:')
    kw.kw_assign(key='end', types=[str])
    _result = kw.first + kw.second
    if kw.is_attribute_exist('end'):
        return_msg = f'{kw.msg} {_result}{kw.end}'
    else:
        return_msg = f'{kw.msg} {_result}'
    return return_msg
```

`first` and `second` are required. `msg` is not required and has a default value. `end` is not required.

```
>>> result = my_method(first=10, second=22)
>>> print(result)
Result: 32
```

Output when optional `args` end and `msg` are included.

```
>>> result = my_method(first=10, second=22, end="!", msg="Tally")
>>> print(result)
Tally 32!
```

Method will raise an error as `msg` must be of type `str` as defined by `types`

```
>>> result = my_method(first=10)
ValueError: KwArg arg 'second' is required
```

Kw_assign Rule Checking

Rule checking can be done by adding `rules_all` or `rules_any` to *kw_assign()* method. Rule checking ensures a value of `**kwargs` values match rules before assigning to current instance of class.

All rules

In the following example args `first` and `second` must be a positive `int` with a maximum value of 100. Arg `rules_all` of *kw_assign()* method validates as `True` only if all *IRule* match. Trying to assign any other type or value to `first` or `second` results in an error.

Custom Rule for a maximum `int` value of 100

```
import kwhelp.rules as rules

class RuleIntMax100(rules.IRule):
    def validate(self) -> bool:
        if self.field_value > 100:
            if self.raise_errors:
                raise ValueError(
                    f"Arg error: '{self.key}' must be equal or less than 100")
            return False
        return True
```

```
from kwhelp import rules, KwArg

def my_method(**kwargs) -> str:
    kw = KwArg(**kwargs)
    kw.kw_assign(key='first', require=True, rules_all=[
            rules.RuleIntPositive,
            RuleIntMax100
        ])
    kw.kw_assign(key='second', require=True, rules_all=[
            rules.RuleIntPositive,
            RuleIntMax100
        ])
    kw.kw_assign(key='msg', types=[str], default='Result:')
    kw.kw_assign(key='end', types=[str])
    first:int = kw.first
    second:int = kw.second
    msg: str = kw.msg
    _result = first + second
```

(continues on next page)

(continued from previous page)

```
    if kw.is_attribute_exist('end'):
        return_msg = f'{msg} {_result}{kw.end}'
    else:
        return_msg = f'{msg} {_result}'
    return return_msg
```

Assign positive `int`.

```
>>> result = my_method(first = 10, second = 22)
>>> print(result)
Result: 32
```

Assign positive `int` and `int` greater than 100.

```
>>> result = my_method(first = 10, second = 122)
kwhelp.exceptions.RuleError: RuleError: Argument: 'second' failed validation.
Rule 'RuleIntMax100' Failed validation.
Expected all of the following rules to match: RuleIntPositive, RuleIntMax100.
Inner Error Message: ValueError: Arg error: 'second' must be equal or less than 100
```

Assign negative `int`.

```
>>> result = my_method(first = -10, second = -22)
kwhelp.exceptions.RuleError: RuleError: Argument: 'first' failed validation.
Rule 'RuleIntPositive' Failed validation.
Expected all of the following rules to match: RuleIntPositive, RuleIntMax100.
Inner Error Message: ValueError: Arg error: 'first' must be a positive int value
```

Assigning `float` result is a `TypeError`

```
>>> result = my_method(first = 10, second = 22.33)
kwhelp.exceptions.RuleError: RuleError: Argument: 'second' failed validation.
Rule 'RuleIntPositive' Failed validation.
Expected all of the following rules to match: RuleIntPositive, RuleIntMax100.
Inner Error Message: TypeError: Argument Error: 'second' is expecting type of 'int'. Got␣
↪type of 'float'
```

Assigning negative `int` results in a `ValueError`.

```
>>> result = my_method(first = 10, second = -5)
kwhelp.exceptions.RuleError: RuleError: Argument: 'second' failed validation.
Rule 'RuleIntPositive' Failed validation.
Expected all of the following rules to match: RuleIntPositive, RuleIntMax100.
Inner Error Message: ValueError: Arg error: 'second' must be a positive int value
```

Any rule

In the following example args `first` and `second` can be a negative or zero `float` or `int` zero. Arg `rules_any` of *kw_assign()* method validates as `True` if any *IRule* matches. Trying to assign any other type to `first` or `second` results in an error.

```
from kwhelp import rules, KwArg

def my_method(**kwargs) -> str:
    kw = KwArg(**kwargs)
    kw.kw_assign(key='first', require=True, rules_any=[
            rules.RuleFloatNegativeOrZero,
            rules.RuleIntZero
        ])
    kw.kw_assign(key='second', require=True, rules_any=[
            rules.RuleFloatNegativeOrZero,
            rules.RuleIntZero
        ])
    kw.kw_assign(key='msg', types=[str], default='Result:')
    kw.kw_assign(key='end', types=[str])
    first:int = kw.first
    second:int = kw.second
    msg: str = kw.msg
    _result = first + second
    if kw.is_attribute_exist('end'):
        return_msg = f'{msg} {_result}{kw.end}'
    else:
        return_msg = f'{msg} {_result}'
    return return_msg
```

Assign `int` zero and `float` negative.

```
>>> result = my_method(first = 0, second = -22.5)
>>> print(result)
Result: -22.5
```

Assign `float` zero and `float` negative.

```
>>> result = my_method(first = 0.0, second = -22.5)
>>> print(result)
Result: -22.5
```

Assign `int` zero and `float` zero.

```
>>> result = my_method(first = 0, second = 0.0)
>>> print(result)
Result: 0.0
```

Assign `float` negative and `float` negative.

```
>>> result = my_method(first = -10.8, second = -8.68)
>>> print(result)
Result: -19.48
```

Assign `int` and `float` zero.

```
>>> result = my_method(first = 12, second = 0.0)
kwhelp.exceptions.RuleError: RuleError: Argument: 'first' failed validation.
Rule 'RuleFloatNegativeOrZero' Failed validation.
Expected at least one of the following rules to match: RuleFloatNegativeOrZero,␣
↪RuleIntZero.
Inner Error Message: TypeError: Argument Error: 'first' is expecting type of 'float'.␣
↪Got type of 'int'
```

Assign `float` positive and `int`.

```
>>> result = my_method(first = 12.46, second = 0)
kwhelp.exceptions.RuleError: RuleError: Argument: 'first' failed validation.
Rule 'RuleFloatNegativeOrZero' Failed validation.
Expected at least one of the following rules to match: RuleFloatNegativeOrZero,␣
↪RuleIntZero.
Inner Error Message: ValueError: Arg error: 'first' must be equal to 0.0 or a negative␣
↪float value
```

Assign `float` negative and `float` positive.

```
>>> result = my_method(first = -12.46, second = 1.2)
kwhelp.exceptions.RuleError: RuleError: Argument: 'second' failed validation.
Rule 'RuleFloatNegativeOrZero' Failed validation.
Expected at least one of the following rules to match: RuleFloatNegativeOrZero,␣
↪RuleIntZero.
Inner Error Message: ValueError: Arg error: 'second' must be equal to 0.0 or a negative␣
↪float value
```

Assigning a `str` results in an error.

```
>>> result = my_method(first=-10.5, second="0")
kwhelp.exceptions.RuleError: RuleError: Argument: 'second' failed validation.
Rule 'RuleFloatNegativeOrZero' Failed validation.
Expected at least one of the following rules to match: RuleFloatNegativeOrZero,␣
↪RuleIntZero.
Inner Error Message: TypeError: Argument Error: 'second' is expecting type of 'float'.␣
↪Got type of 'str'
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*

- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*
- *RuleIntZero*
- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*
- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

Kw_assign Type Checking

Type checking can be done by adding `types` to *kw_assign()* method. Type checking ensures the type of `**kwargs` values that are assigned to attributes of current instance of class.

```
from kwhelp import KwArg

def speed_msg(**kwargs) -> str:
    kw = KwArg(**kwargs)
    kw.kw_assign(key="speed", require=True, types=[int, float])
    if kw.speed > 100:
        msg = f"Speed of '{kw.speed}' is fast. Caution recommended."
    elif kw.speed < -40:
        msg = f"Reverse speed of '{kw.speed}' is fast. Caution recommended."
    elif kw.speed < 0:
        msg = f"Reverse speed of '{kw.speed}'. Normal operations."
    else:
        msg = f"speed of '{kw.speed}'. Normal operations."
    return msg
```

speed_msg `float` value.

```
>>> result = speed_msg(speed = 35.8)
>>> print(result)
speed of '35.8'. Normal operations.
```

speed_msg `float` fast value.

```
>>> result = speed_msg(speed = 227.59)
>>> print(result)
Speed of '227.59' is fast. Caution recommended.
```

speed_msg `int` value.

```
>>> result = speed_msg(speed = -43)
>>> print(result)
Reverse speed of '-43' is fast. Caution recommended.
```

speed_msg no params.

```
>>> result = speed_msg()
ValueError: KwArg arg 'speed' is required
```

speed_msg wrong type.

```
>>> result = speed_msg(speed = "Fast")
TypeError: KwArg arg 'speed' is expected to be of '<class 'float'> | <class 'int'>' but␣
↪got 'str'
```

2.1.3 Decorators

Decorators can be used to validate functions.

Decorator Index

AcceptedTypes Usage

AcceptedTypes Decorator that decorates methods that requires args to match types specificed.

Includes features:

- *type_instance_check*
- *ftype*
- *opt_all_args*
- *opt_args_filter*
- *opt_logger*
- *opt_return*

Normal Function

The following example requres:

- one is of type `int`
- two is of type `float` or `int`
- three is of type `str`.

```
from kwhelp.decorator import AcceptedTypes

@AcceptedTypes(int, (float, int), str)
def foo(one, two, three):
    result = [one, two, three]
    return result
```

```
>>> result = foo(1, 2.2, "Hello")
>>> print(result)
[1, 2.2, 'Hello']
```

```
>>> result = foo(1, "2.2", "Hello")
>>> print(result)
TypeError: Arg 'two' in 2nd position is expected to be of 'float' or 'int' but got 'str'.
AcceptedTypes decorator error.
```

*args

`*args` can be used a parameters. When using `*args` with *AcceptedTypes* the total number of args for the function must match the number of types passed into `AcceptedTypes`.

```
from kwhelp.decorator import AcceptedTypes

@AcceptedTypes(int, (float, int), int, (int, str), int)
def foo(one, two, three, *args):
    result = [one, two, three]
    for arg in args:
        result.append(arg)
    return result
```

All value of type `int`

```
>>> result = foo(1, 2, 3, 4, 5)
>>> print(result)
[1, 2, 3, 4, 5]
```

Alternative type for args that support them.

```
>>> result = foo(1, 2.77, 3, "Red", 5)
>>> print(result)
[1, 2.77, 3, 'Red', 5]
```

Last arg is not of type `int` and raised an error

```
>>> result = foo(1, 2, 3, 4, 5.766)
TypeError: Arg in 5th position of is expected to be of 'int' but got 'float'.
AcceptedTypes decorator error.
```

Too many args passed into Function result in an error

```
 >>> result = foo(1, 2, 3, 4, 5, 1000)
ValueError: Invalid number of arguments for foo()
AcceptedTypes decorator error.
```

**kwargs

**kwargs can be used a parameters. When using *args with *AcceptedTypes* the total number of args for the function must match the number of types passed into AcceptedTypes.

```
@AcceptedTypes(int, (float, int), int, (int, str), int)
def foo(one, *args, **kwargs):
    result_args = [*args]
    result_args.insert(0, one)
    result_dic = {**kwargs}
    return result_args, result_dic
```

All int values with last arg as key, value.

```
>>> result = foo(1, 2, 3, 4, last=5)
>>> print(result)
([1, 2, 3, 4], {'last': 5})
```

```
>>> result = foo(1, 2, 3, 4, last=5, exceeded=None)
ValueError: Invalid number of arguments for foo()
AcceptedTypes decorator error.
```

Class Method

AcceptedTypes can be applied to class methods. When appling to class method set the ftype arg to match *DecFuncEnum*.

Regular Class Method

Class method applying to constructor.

```
from kwhelp.decorator import AcceptedTypes, DecFuncEnum

class Foo:
    @AcceptedTypes((int, float), (int, float), ftype=DecFuncEnum.METHOD)
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
```

```
>>> f = Foo(1, 99.9)
>>> print(f.start, f.stop)
1 99.9
```

```
>>> f = Foo(1, None)
TypeError: Arg 'stop' in 2nd position is expected to be of 'int' or 'float' but got
↪'NoneType'.
AcceptedTypes decorator error.
```

Static Class Method

AcceptedTypes can be use on static method of a class as well by setting `ftype` to *DecFuncEnum* `METHOD_STATIC` option.

```
from kwhelp.decorator import AcceptedTypes, DecFuncEnum, ReturnType

class Foo:
    @staticmethod
    @AcceptedTypes(int, int, ftype=DecFuncEnum.METHOD_STATIC)
    @ReturnType(int)
    def add(first, last):
        return first + last
```

```
>>> print(Foo.add(34, 76))
110
```

```
>>> print(Foo.add(7.2, 76))
TypeError: Arg 'first' in 1st position is expected to be of 'int' but got 'float'.
AcceptedTypes decorator error.
```

Class Method

AcceptedTypes can be use on class method of a class as well by setting `ftype` to *DecFuncEnum* `METHOD_CLASS` option.

```
from kwhelp.decorator import AcceptedTypes, DecFuncEnum, ReturnType

class Foo:
    @classmethod
    @AcceptedTypes(int, int, ftype=DecFuncEnum.METHOD_CLASS)
    @ReturnType(int)
    def add(cls, first, last):
        return first + last
```

```
>>> print(Foo.add(34, 76))
110
```

```
>>> print(Foo.add(7.2, 76))
TypeError: Arg 'first' in 1st position is expected to be of 'int' but got 'float'.
AcceptedTypes decorator error.
```

Option opt_all_args

`opt_all_args` argument allows the last class type passed into *AcceptedTypes* to validate all remaining arguments of wrapped function.

For more examples see *opt_all_args*.

```
@AcceptedTypes(float, (float, int), opt_all_args=True)
def sum_num(*args):
    return sum(args)
```

The first arg of `sum_num` must be a `float`. Remaining args can be `float` or `int`.

```
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88))
130.736
>>> print(sum_num(1, 44.556, 10, 22, 45, 7.88))
TypeError: Arg in 1st position of is expected to be of 'float' but got 'int'.
AcceptedTypes decorator error.
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88, "77"))
TypeError: Arg in 7th position of is expected to be of 'float' or 'int' but got 'str'.
AcceptedTypes decorator error.
```

Opton opt_args_filter

The arguments are validated by *AcceptedTypes* can be filtered by setting `opt_args_filter` option.

For more examples see *opt_args_filter*.

In the following example all `*args` must of of type `float` or `int`. `opt_args_filter=DecArgEnum.ARGS` filters `AcceptedTypes` to only process `*args`.

```
from kwhelp.decorator import AcceptedTypes, DecArgEnum

@AcceptedTypes((float, int), opt_all_args=True, opt_args_filter=DecArgEnum.ARGS)
def sum_num(*args, msg: str):
    _sum = sum(args)
    return msg + str(_sum)
```

```
>>> result = sum_num(1, 2, 3, 4, 5, 6, msg='Total: ')
>>> print(result)
Total: 21
```

Combined Decorators

AcceptedTypes can be combined with other decorators.

The following example limits how many args are allowed by applying *ArgsMinMax* decorator.

```
from kwhelp.decorator import AcceptedTypes, ArgsMinMax

@ArgsMinMax(max=6)
@AcceptedTypes(float, (float, int), opt_all_args=True)
def sum_num(*args):
    return sum(args)
```

```
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88))
130.736
>>> print(sum_num(1, 44.556, 10, 22, 45, 7.88, 100))
ValueError: Invalid number of args pass into 'sum_num'.
Expected max of '6'. Got '7' args.
ArgsMinMax decorator error.
```

Multiple AcceptedTypes

Multiple *AcceptedTypes* can be applied to a single function.

```
from kwhelp.decorator import AcceptedTypes, DecArgEnum

@AcceptedTypes(str, opt_all_args=True, opt_args_filter=DecArgEnum.ARGS)
@AcceptedTypes((int, float), opt_all_args=True, opt_args_filter=DecArgEnum.NO_ARGS)
def foo(*args, first, last=1001, **kwargs):
    return [*args] + [first, last] + [v for _, v in kwargs.items()]
```

```
>>> result = foo("a", "b", "c", first=-100, one=101.11, two=22.22, third=33.33)
>>> print(result)
['a', 'b', 'c', -100, 1001, 101.11, 22.22, 33.33]
>>> result = foo("a", "b", 1, first=-100, one=101.11, two=22.22, third=33.33)
TypeError: Arg in 3rd position of is expected to be of 'str' but got 'int'.
AcceptedTypes decorator error.
>>> result = foo("a", "b", "c", first=-100, one=101.11, two="2nd", third=33.33)
TypeError: Arg 'two' in 4th position is expected to be of 'float' or 'int' but got 'str'.
AcceptedTypes decorator error.
```

ArgsLen Usage

ArgsLen decorartor that sets the number of args that can be added to a function.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*

Single Length

Decorator can be applied with a single set Length. In the following example if anything other than 3 args are passed into `foo` a `ValueError` will be raised

```
from kwhelp.decorator import ArgsLen

@ArgsLen(3)
def foo(*args):
    return len(args)
```

Passing in three arg values works as expected.

```
>>> result = foo("a", "b", "c")
>>> print(result)
3
```

Passing in two args when three are expected raises a `ValueError`

```
>>> result = foo("a", "b")
ValueError: Invalid number of args pass into 'foo'.
Expected Length: '3'. Got '4' args.
ArgsLen decorator error.
```

Multiple Lengths

It is possible to allow more then one length to function by passing in multilple `int` values to decorator.

```
from kwhelp.decorator import ArgsLen

@ArgsLen(3, 5)
def foo(*args):
    return len(args)
```

Passing in 3 args.

```
>>> result = foo("a", "b", "c")
>>> print(result)
3
```

Passing in 5 args.

```
>>> result = foo("a", "b", "c", "d", "e")
>>> print(result)
5
```

Passing in 4 args result in a `ValueError`.

```
>>> result = foo("a", "b", "c", "d")
ValueError: Invalid number of args pass into 'foo'.
Expected Lengths: '3' or '5'. Got '4' args.
ArgsLen decorator error.
```

Ranges

It is possible to allow more then one length to function by passing in pairs of `int` values in the form of iterable values such as list or tuple values to decorator.

The following example allows 3, 4, 5, 7, 8, 9 args. Note that 1, 2, 6 or greater than 9 args will result in a `ValueError`.

```
from kwhelp.decorator import ArgsLen

@ArgsLen((3, 5), (7, 9))
def foo(*args):
    return len(args)
```

Passing in 3 args.

```
>>> result = foo("a", "b", "c")
>>> print(result)
3
```

Passing in 8 args.

```
from kwhelp.decorator import ArgsLen

>>> result = foo("a", "b", "c", "d", "e", "f", "g", "h")
>>> print(result)
8
```

Passing in 6 args.

```
>>> result = foo("a", "b", "c", "d", "e", "f")
ValueError: Invalid number of args pass into 'foo'.
Expected Ranges: (3, 5) or (7, 9). Got '6' args.
ArgsLen decorator error.
```

Ranges & Lengths

Ranges and lengths can be combined when needed.

The following example allows 3, 4, 5, 7, 8, 9 args. Note that 1, 2, 6 or greater than 9 args will result in a `ValueError`.

```
from kwhelp.decorator import ArgsLen

@ArgsLen(3, 4, 5, (7, 9))
def foo(*args):
    return len(args)
```

Passing in 3 args.

```
>>> result = foo("a", "b", "c")
>>> print(result)
3
```

Passing in 8 args.

```
>>> result = foo("a", "b", "c", "d", "e", "f", "g", "h")
>>> print(result)
8
```

Passing in 6 args.

```
>>> result = foo("a", "b", "c", "d", "e", "f")
ValueError: Invalid number of args pass into 'foo'.
Expected Lengths: '3', '4', or '5'. Expected Range: (7, 9). Got '6' args.
ArgsLen decorator error.
```

Class

Decorator can be used on class methods by setting `ftype` arg. to a value of *DecFuncEnum*.

Normal class

```
from kwhelp.decorator import ArgsLen

class Foo:
    @ArgsLen(0, (2, 4), ftype=DecFuncEnum.METHOD)
    def __init__(self, *args): pass

    @ArgsLen(3, 5, ftype=DecFuncEnum.METHOD)
    def bar(self, *args): pass
```

Static method

```
from kwhelp.decorator import ArgsLen

class Foo:
    @staticmethod
    @ArgsLen(3, 5, ftype=DecFuncEnum.METHOD_STATIC)
    def bar(self, *args): pass
```

Class method

```
from kwhelp.decorator import ArgsLen

class Foo:
    @staticmethod
    @ArgsLen(3, 5, ftype=DecFuncEnum.METHOD_CLASS)
    def bar(self, *args): pass
```

ArgsMinMax Usage

ArgsMinMax decorartor that sets the min and/or max number of args that can be added to a function.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*

Single Length

Decorator can be applied with a min Length. In the following example if less than 3 args are passed into `foo` a `ValueError` will be raised

```
from kwhelp.decorator import ArgsMinMax

@ArgsMinMax(min=3)
def foo(*args):
    return len(args)
```

Passing in three arg values works as expected.

```
>>> result = foo("a", "b", "c")
>>> print(result)
3
```

Passing in two args when three are expected raises a `ValueError`

```
>>> result = foo("a", "b")
ValueError: Invalid number of args pass into 'foo'.
Expected min of '3'. Got '2' args.
ArgsMinMax decorator error.
```

Min and Max

It is possible to set min and max allowed arguments.

```
from kwhelp.decorator import ArgsMinMax

@ArgsMinMax(min=3, max=5)
def foo(*args):
    return len(args)
```

Passing in 3 args.

```
>>> result = foo("a", "b", "c")
>>> print(result)
3
```

Passing in 5 args.

```
>>> result = foo("a", "b", "c", "d", "e")
>>> print(result)
5
```

Passing in 6 args result in a `ValueError`.

```
>>> result = foo("a", "b", "c", "d", "e", "f")
ValueError: Invalid number of args pass into 'foo'.
Expected min of '3'. Expected max of '5'. Got '6' args.
ArgsMinMax decorator error.
```

Class

Decorator can be used on class methods by setting `ftype` arg. to a value of *DecFuncEnum*.

Normal class

```
from kwhelp.decorator import ArgsMinMax

class Foo:
    @ArgsMinMax(max=6, ftype=DecFuncEnum.METHOD)
    def __init__(self, *args): pass

    @ArgsMinMax(3, 5, ftype=DecFuncEnum.METHOD)
    def bar(self, *args): pass
```

Static method

```
from kwhelp.decorator import ArgsMinMax

class Foo:
    @staticmethod
    @ArgsMinMax(min=3 max=5, ftype=DecFuncEnum.METHOD_STATIC)
    def bar(self, *args): pass
```

Class method

```
from kwhelp.decorator import ArgsMinMax

class Foo:
    @staticmethod
    @ArgsMinMax(min=3 max=5, ftype=DecFuncEnum.METHOD_CLASS)
    def bar(self, *args): pass
```

DefaultArgs Usage

DefaultArgs decorator that defines default values for **kwargs of a function.

```
from kwhelp.decorator import DefaultArgs

@DefaultArgs(speed=45, limit=60, name="John")
def speed_msg(**kwargs) -> str:
    name = kwargs.get('name')
    limit = kwargs.get('limit')
    speed = kwargs.get('speed')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

```
>>> result = speed_msg()
>>> print(result)
Current speed is '45'. John may go faster as the limit is '60'.
```

```
>>> result = speed_msg(name="Sue", speed=47)
>>> print(result)
Current speed is '47'. Sue may go faster as the limit is '60'.
```

RequireArgs Usage

RequireArgs decorator defines required args for **kwargs of a function.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*

```
from kwhelp.decorator import TypeCheckKw, RequireArgs

@RequireArgs("speed", "limit", "name")
@TypeCheckKw(arg_info={"speed": 0, "limit": 0, "hours": 0, "name": 1},
                types=[(int, float), str])
def speed_msg(**kwargs) -> str:
    name = kwargs.get('name')
    limit = kwargs.get('limit')
    speed = kwargs.get('speed')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
```

(continues on next page)

(continued from previous page)

```
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

```
>>> result = speed_msg(speed=45, limit=60, name="John")
>>> print(result)
Current speed is '45'. John may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=45, limit=60)
>>> print(result)
ValueError: 'name' is a required arg.
```

```
>>> result = speed_msg(speed="Fast", limit=60, name="John")
>>> print(result)
TypeError: Arg 'speed' is expected to be of '<class 'int'> | <class 'float'>' but got
↪'str'
```

ReturnRuleAll usage

ReturnRuleAll decorator that decorates methods that require return value to match all of the rules specificed.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*

Function Usage

In the following example return value must be an `int` or a negative `float` value.

```
from kwhelp.decorator import ReturnRuleAll
from kwhelp import rules

@ReturnRuleAll(rules.RuleIntPositive)
def req_test(*arg):
    return sum(arg)
```

```
>>> result = req_test(2, 4)
>>> print(result)
6
```

Argument of `float` raised error.

```
>>> result = req_test(2, 2.4)
kwhelp.exceptions.RuleError: RuleError: 'req_test' error. Argument: 'return' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: TypeError: Argument Error: 'return' is expecting type of 'int'. Got␣
↪type of 'float'
```

Negative return value raises error.

```
>>> result = req_test(2, -3)
kwhelp.exceptions.RuleError: RuleError: 'req_test' error. Argument: 'return' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: ValueError: Arg error: 'return' must be a positive int value
```

Class Usage

In the following example `ReturnRuleAll` is applied to a class method. Note that `ftype` arg is set to `DecFuncEnum.METHOD` for class methods.

```
from kwhelp.decorator import ReturnRuleAll
from kwhelp import rules

class T:
    @ReturnRuleAll(rules.RuleIntPositive, ftype=DecFuncEnum.METHOD)
    def req_test(self, *arg):
        return sum(arg)
```

```
>>> t = T()
>>> result = t.req_test(2, 4)
>>> print(result)
6
```

Argument of `float` raised error.

```
>>> t = T()
>>> result = t.req_test(2, 2.5)
kwhelp.exceptions.RuleError: RuleError: 'req_test' error. Argument: 'return' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: TypeError: Argument Error: 'return' is expecting type of 'int'. Got␣
↪type of 'float'
```

Negative return value raises error.

```
>>> t = T()
>>> result = t.req_test(2, -3)
kwhelp.exceptions.RuleError: RuleError: 'req_test' error. Argument: 'return' failed␣
↪validation.
```

(continues on next page)

(continued from previous page)

```
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: ValueError: Arg error: 'return' must be a positive int value
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*
- *RuleIntZero*
- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*
- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

ReturnRuleAny Usage

ReturnRuleAny decorator that decorates methods that require return value to match at least one of the rules specificed.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*

Function Usage

In the following example return value must be an `int` or a negative `float` value.

```
from kwhelp.decorator import ReturnRuleAny
from kwhelp import rules

@ReturnRuleAny(rules.RuleInt, rules.RuleFloatNegative)
def req_test(*arg):
    return sum(arg)
```

Postiive return `int` value is valid

```
>>> result = req_test(2, 4)
>>> print(result)
6
```

Negative return `float` value is valid

```
>>> result = req_test(2, -5.2)
>>> print(result)
-3.2
```

Positive return `float` value raised error.

```
>>> result = req_test(2, 2.4)
kwhelp.exceptions.RuleError: RuleError: 'req_test' error. Argument: 'return' failed␣
↪validation.
Rule 'RuleInt' Failed validation.
Expected at least one of the following rules to match: RuleInt, RuleFloatNegative.
Inner Error Message: TypeError: Argument Error: 'return' is expecting type of 'int'. Got␣
↪type of 'float'
```

Class Usage

In the following example `ReturnRuleAny` is applied to a class method. Note that `ftype` arg is set to `DecFuncEnum.METHOD` for class methods.

```
from kwhelp.decorator import ReturnRuleAny
from kwhelp import rules

class T:
    @ReturnRuleAny(rules.RuleInt, rules.RuleFloatNegative, ftype=DecFuncEnum.METHOD)
    def req_test(self, *arg):
        return sum(arg)
```

```
>>> t = T()
>>> result = t.req_test(2, 4)
>>> print(result)
6
```

Returning a postitive float result in an error.

```
>>> t = T()
>>> result = t.req_test(2, 2.5)
kwhelp.exceptions.RuleError: RuleError: 'req_test' error. Argument: 'return' failed␣
↪validation.
Rule 'RuleInt' Failed validation.
Expected at least one of the following rules to match: RuleInt, RuleFloatNegative.
Inner Error Message: TypeError: Argument Error: 'return' is expecting type of 'int'. Got␣
↪type of 'float'
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*

- *RuleIntZero*
- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*
- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

ReturnType Usage

ReturnType decorator reqires that return matches `type`.

Includes features:

- *type_instance_check*
- *opt_logger*
- *opt_return*

Single Arg Usage

The following example requres return type of `str` by applying *ReturnType* decorator with parameter of `str`.

```
from kwhelp.decorator import ReturnType

@ReturnType(str)
def foo(*args):
    result = None
    for i, arg in enumerate(args):
        if i == 0:
            result = arg
        else:
            result = result + ", " + arg
    return result
```

Result when type of `str` is returned.

```
>>> result = foo("Hello", "World", "Happy", "Day")
>>> print(result)
Hello, World, Happy, Day
```

Error is raises when retrun type is not valid.

```
>>> result = foo(2)
TypeError: Return Value is expected to be of 'str' but got 'int'.
ReturnType decorator error.
```

Multiple Arg Usage

ReturnType can accept multiple return types.

In the following example return type must be `int` or `str`.

```
from kwhelp.decorator import ReturnType

@ReturnType(int, str)
def ret_test(start, length, end):
    result = start + length + end
    return result
```

```
>>> result = ret_test(2, 4, 6)
>>> print(result)
12
```

```
>>> result = ret_test("In the beginning ", "and forever more, ", "time is everlasting.")
>>> print(result)
In the beginning and forever more, time is everlasting.
```

```
>>> result = ret_test(1.33, 4, 6)
TypeError: Return Value is expected to be of 'int' or 'str' but got 'float'.
ReturnType decorator error.
```

RuleCheckAll Usage

RuleCheckAll decorator reqires that each args of a function match all rules specified.

Includes features:

- *ftype*
- *opt_args_filter*
- *opt_logger*
- *opt_return*
- *raise_error*

Decorating with *args

This example requires that all args positive `int`

```
from kwhelp.decorator import RuleCheckAll
import kwhelp.rules as rules

@RuleCheckAll(rules.RuleIntPositive)
def add_positives(*args) -> float:
    result = 0
    for arg in args:
        result += arg
    return result
```

Adding positive numbers works as expected.

```
>>> result = add_positives(1, 4, 6, 3, 10)
>>> print(result)
24
```

Because decorator rules dictate that only postiive numbers are allowed. A negative number will raise an error.

```
>>> result = add_positives(2, 1, -1)
kwhelp.exceptions.RuleError: RuleError: 'add_positives' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
```

Rules dictate that if a type is not `int` then an error will be raised.

```
>>> result = add_positives(2, 1, 3.55)
kwhelp.exceptions.RuleError: RuleError: 'add_positives' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: TypeError: Argument Error: 'arg' is expecting type of 'int'. Got␣
↪type of 'float'
```

Decorating with Key, Value

This example requires that all args positive `int`.

```
from kwhelp.decorator import RuleCheckAll
import kwhelp.rules as rules

@RuleCheckAll(rules.RuleIntPositive)
def speed_msg(speed, limit, **kwargs) -> str:
    if limit > speed:
        msg = f"Current speed is '{speed}'. You may go faster as the limit is '{limit}'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. You are at the limit."
```

(continues on next page)

(continued from previous page)

```
    else:
        msg = f"Please slow down limit is '{limit}' and you are currenlty going '{speed}
↪'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

Adding positive numbers works as expected.

```
>>> result = speed_msg(speed=45, limit=60)
>>> print(result)
Current speed is '45'. You may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=66, limit=60, hours=3)
>>> print(result)
Please slow down limit is '60' and you are currenlty going '66'. Current driving hours␣
↪is '3'.
```

Because decorator rules dictate that only postiive numbers are allowed. A negative number will raise an error.

```
>>> result = speed_msg(speed=-2, limit=60)
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'speed' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: ValueError: Arg error: 'speed' must be a positive int value
```

```
>>> result = speed_msg(speed=66, limit=60, hours=-2)
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'hours' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: ValueError: Arg error: 'hours' must be a positive int value
```

Rules dictate that if a type is not `int` then an error will be raised.

```
>>> result = speed_msg(speed=45, limit="Fast")
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'limit' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: TypeError: Argument Error: 'limit' is expecting type of 'int'. Got␣
↪type of 'str'
```

Opton opt_args_filter

The arguments are validated by *RuleCheckAll* can be filtered by setting `opt_args_filter` option.

For more examples see *opt_args_filter*.

In the following example all `*args` must be positive `int`. `opt_args_filter=DecArgEnum.ARGS` filters `RuleCheckAll` to only process `*args`.

```
from kwhelp.decorator import RuleCheckAll, DecArgEnum
from kwhelp import rules

@RuleCheckAll(rules.RuleIntPositive, opt_args_filter=DecArgEnum.ARGS)
def foo(*args, first, last="!", **kwargs):
    return [*args] + [first, last] + [v for _, v in kwargs.items()]
```

```
>>> result = foo(101, 223, 778, 887, first="1st", one='one', two="2nd")
>>> print(result)
[101, 223, 778, 887, '1st', '!', 'one', '2nd']
>>> result = foo(101, 223, -778, 887, first="1st", one='one', two="2nd")
kwhelp.exceptions.RuleError: RuleError: 'foo' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*
- *RuleIntZero*
- *RuleIterable*
- *RuleNone*

- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*
- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

RuleCheckAllKw Usage

RuleCheckAllKw decorator allows each arg of a function match all rules specified. Each arg can have seperate rules applied.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*
- *raise_error*

RuleCheckAllKw constructor args `arg_info` and `rules` work together. `arg_info` is a dictionary with a key of `str` that matches an arg name of the function that is being decorated. `arg_info` value is one of the following:

- `int` is an index of an item in `rules`
- `IRule` a rule to match
- `Iterator[IRule]` a list of rules to match

`arg_info` can be *mixed*.

`rules` is a list of rules to match. Each element is an *IRule* or a list of *IRule*.

Custom IRule

Custom *IRule* class that limits value to a max `int` value of `100`.

```
import kwhelp.rules as rules

class RuleIntMax100(rules.IRule):
    def validate(self) -> bool:
        if self.field_value > 100:
            if self.raise_errors:
```

(continues on next page)

(continued from previous page)

```
                raise ValueError(
                    f"Arg error: '{self.key}' must be equal or less than 100")
            return False
        return True
```

Example Usage

RuleCheckAllKw decorated function.

```
from kwhelp.decorator import RuleCheckAllKw
import kwhelp.rules as rules

@RuleCheckAllKw(arg_info={"speed": 0, "limit": 1, "hours": 1, "name": 2},
                rules=[rules.RuleIntPositive,
                       (rules.RuleIntPositive, RuleIntMax100),
                       rules.RuleStrNotNullEmptyWs])
def speed_msg(speed, limit, **kwargs) -> str:
    name = kwargs.get('name', 'You')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

```
>>> result = speed_msg(speed=45, limit=60)
>>> print(result)
Current speed is '45'. You may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=45, limit=60, name="John")
>>> print(result)
Current speed is '45'. John may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=66, limit=60, hours=3, name="John")
>>> print(result)
Please slow down limit is '60' and current speed is '66'. Current driving hours is '3'.
```

If any rule fails validation then a *RuleError* is raised.

```
>>> result = speed_msg(speed=-2, limit=60)
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'speed' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAllKw decorator error.
Inner Error Message: ValueError: Arg error: 'speed' must be a positive int value
```

```
>>> result = speed_msg(speed=66, limit=60, name="  ")
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'name' failed␣
↪validation.
Rule 'RuleStrNotNullEmptyWs' Failed validation.
Expected the following rule to match: RuleStrNotNullEmptyWs.
RuleCheckAllKw decorator error.
Inner Error Message: ValueError: Arg error: 'name' must not be empty or whitespace str
```

speed_msg decorated with a mixed arg_info with IRule instance and index to rules.

```
from kwhelp.decorator import RuleCheckAllKw
import kwhelp.rules as rules

@RuleCheckAllKw(arg_info={"speed": rules.RuleIntPositive, "limit": 0,
                    "hours": 0, "name": rules.RuleStrNotNullEmptyWs},
                rules=[(rules.RuleIntPositive, RuleIntMax100)])
def speed_msg(speed, limit, **kwargs) -> str:
    name = kwargs.get('name', 'You')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*

- *RuleIntZero*
- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*
- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

RuleCheckAny Usage

`RuleCheckAny` decorator reqires that each args of a function match one or more rules.

Includes features:

- *ftype*
- *opt_args_filter*
- *opt_logger*
- *opt_return*
- *raise_error*

Decorating with `*args`

This example requires that all args positive `int` or positive `float`.

```
from kwhelp.decorator import RuleCheckAny
import kwhelp.rules as rules

@RuleCheckAny(rules.RuleIntPositive, rules.RuleFloatPositive)
def add_positives(*args) -> float:
    result = 0.0
    for arg in args:
        result += float(arg)
    return result
```

Adding positive numbers works as expected.

```
>>> result = add_positives(1, 4, 6.9, 3.9, 7.3)
>>> print(result)
23.1
```

Because decorator rules dictate that only postiive numbers are allowed. A negative number will raise an error.

```
>>> result = add_positives(2, 1.2, -1)
kwhelp.exceptions.RuleError: RuleError: 'add_positives' error.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
RuleCheckAny decorator error.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
```

Rules dictate that if a type is not `int` or `float` then an error will be raised.

```
>>> result = add_positives(2, 1.2, "4")
kwhelp.exceptions.RuleError: RuleError: 'add_positives' error.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
RuleCheckAny decorator error.
Inner Error Message: TypeError: Argument Error: 'arg' is expecting type of 'int'. Got␣
↪type of 'str'
```

Decorating with Key, Value

This example requires that all args are positive `int` or positive `float`.

```
from kwhelp.decorator import RuleCheckAny
import kwhelp.rules as rules

@RuleCheckAny(rules.RuleIntPositive, rules.RuleFloatPositive)
def speed_msg(speed, limit, **kwargs) -> str:
    if limit > speed:
        msg = f"Current speed is '{speed}'. You may go faster as the limit is '{limit}'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. You are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and you are currenlty going '{speed}
↪'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'"
    return msg
```

Adding positive numbers works as expected.

```
>>> result = speed_msg(speed=45, limit=60)
>>> print(result)
Current speed is '45'. You may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=66, limit=60, hours=4.7)
>>> print(result)
Please slow down limit is '60' and you are currenlty going '66'. Current driving hours␣
↪is '4.7
```

Because decorator rules dictate that only postiive numbers are allowed. A negative number will raise an error.

```
>>> result = speed_msg(speed=-2, limit=60)
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'speed' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
RuleCheckAny decorator error.
Inner Error Message: ValueError: Arg error: 'speed' must be a positive int value
```

```
>>> result = speed_msg(speed=66, limit=60, hours=-0.2)
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'hours' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
RuleCheckAny decorator error.
Inner Error Message: TypeError: Argument Error: 'hours' is expecting type of 'int'. Got␣
↪type of 'float'
```

Rules dictate that if a type is not `int` or `float` then an error will be raised.

```
>>> result = speed_msg(speed=45, limit="Fast")
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'limit' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
RuleCheckAny decorator error.
Inner Error Message: TypeError: Argument Error: 'limit' is expecting type of 'int'. Got␣
↪type of 'str'
```

Opton opt_args_filter

The arguments are validated by *RuleCheckAny* can be filtered by setting `opt_args_filter` option.

For more examples see *opt_args_filter*.

In the following example all `*args` must be positive `int` or postive `float`. `opt_args_filter=DecArgEnum.ARGS` filters `RuleCheckAll` to only process `*args`.

```
from kwhelp.decorator import RuleCheckAny, DecArgEnum
from kwhelp import rules

@RuleCheckAny(rules.RuleIntPositive, rules.RuleFloatPositive, opt_args_filter=DecArgEnum.
↪ARGS)
```

(continues on next page)

(continued from previous page)

```
def foo(*args, first, last="!", **kwargs):
    return [*args] + [first, last] + [v for _, v in kwargs.items()]
```

```
>>> result = foo(101.11, 223.77, 778, 887, first="1st", one='one', two="2nd")
>>> print(result)
[101.11, 223.77, 778, 887, '1st', '!', 'one', '2nd']
>>> result = foo(101.11, -223.77, 778, 887, first="1st", one='one', two="2nd")
kwhelp.exceptions.RuleError: RuleError: 'foo' error.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
RuleCheckAny decorator error.
Inner Error Message: TypeError: Argument Error: 'arg' is expecting type of 'int'. Got␣
↪type of 'float'
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*
- *RuleIntZero*
- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*

- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

RuleCheckAnyKw Usage

RuleCheckAnyKw decorator allows each arg of a function match one of the rules specified. Each arg can have seperate rules applied.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*
- *raise_error*

RuleCheckAnyKw constructor args `arg_info` and `rules` work together. `arg_info` is a dictionary with a key of `str` that matches an arg name of the function that is being decorated. `arg_info` value is one of the following:

- `int` is an index of an item in `rules`
- `IRule` a rule to match
- `Iterator[IRule]` a list of rules to match

`arg_info` can be *mixed*.

`rules` is a list of rules to match. Each element is an *IRule* or a list of *IRule*.

Example Usage

RuleCheckAnyKw decorated function.

```
from kwhelp.decorator import RuleCheckAnyKw
import kwhelp.rules as rules

@RuleCheckAnyKw(arg_info={"speed": 0, "limit": 0, "hours": 0, "name": 1},
                rules=[(rules.RuleIntPositive, rules.RuleFloatPositive),
                       rules.RuleStrNotNullEmptyWs])
def speed_msg(speed, limit, **kwargs) -> str:
    name = kwargs.get('name', 'You')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
```

(continues on next page)

(continued from previous page)

```
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

```
>>> result = speed_msg(speed=45, limit=60)
>>> print(result)
Current speed is '45'. You may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=45, limit=60, name="John")
>>> print(result)
Current speed is '45'. John may go faster as the limit is '60'.
```

If all rules fail validation then a *RuleError* is raised.

```
>>> result = speed_msg(speed=-2, limit=60)
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'speed' failed␣
↪validation.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
RuleCheckAnyKw decorator error.
Inner Error Message: ValueError: Arg error: 'speed' must be a positive int value
```

```
>>> result = speed_msg(speed=66, limit=60, name="  ")
kwhelp.exceptions.RuleError: RuleError: 'speed_msg' error. Argument: 'name' failed␣
↪validation.
Rule 'RuleStrNotNullEmptyWs' Failed validation.
Expected the following rule to match: RuleStrNotNullEmptyWs.
RuleCheckAnyKw decorator error.
Inner Error Message: ValueError: Arg error: 'name' must not be empty or whitespace str
```

speed_msg decorated with a mixed arg_info with IRule instance and index to rules.

```
from kwhelp.decorator import RuleCheckAllKw
import kwhelp.rules as rules

@RuleCheckAnyKw(arg_info={"speed": 0, "limit": 0, "hours": 0, "name": rules.
↪RuleStrNotNullEmptyWs},
                rules=[(rules.RuleIntPositive, rules.RuleFloatPositive)])
def speed_msg(speed, limit, **kwargs) -> str:
    name = kwargs.get('name', 'You')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*
- *RuleIntZero*
- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*
- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

SubClass Usage

SubClass decorator that requires args of a function to match or be a subclass of types specificed in constructor.

Includes features:

- *ftype*
- *opt_all_args*
- *opt_args_filter*
- *opt_logger*
- *opt_return*
- *raise_error*

Sample Classes

```
from enum import IntEnum, auto

class Color(IntEnum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()

    def __str__(self) -> str:
        return self._name_

class Base:
    def __str__(self) -> str:
        return self.__class__.__name__
class Obj:
    def __str__(self) -> str:
        return self.__class__.__name__
class Foo(Base): pass
class Bar(Foo): pass
class FooBar(Bar): pass
class ObjFoo(Obj): pass
```

Decorating with *args

Single arg match

This example requires that all args positive `Foo` or positive `ObjFoo`. Each subclass type in `SubClass` constructor represents one positional arg. In the following example two postional args are expected.

```
from kwhelp.decorator import SubClass

@SubClass(Foo, ObjFoo)
def do_something(*args):
    return [str(arg) for arg in args]
```

Expected args pass validatation.

```
>>> print(do_something(Foo(), ObjFoo()))
    ['Foo', 'ObjFoo']
```

Types dictate that if first arg is not a subclass of `Foo` or first arg is not a subclass of `ObjFoo` then an error will be raised.

```
>>> print(do_something(Foo(), Obj()))
TypeError: Arg in 2nd position is expected to be of a subclass of 'ObjFoo'.
SubClass decorator error.
```

Arguments passed into function must match the same number of SubClass Types. If not the same count then a `ValueError` is rasied.

```
>>> do_something(Foo(), ObjFoo(), Bar())
ValueError: Invalid number of arguments for do_something()
SubClass decorator error.
```

Multi Choice

```
from kwhelp.decorator import SubClass

@SubClass((FooBar, ObjFoo),(Color, Obj))
def do_something(*args):
    return str(first), str(last)
```

This call to `do_something` raises no errors.

```
>>> print(do_something(FooBar(), Color.RED))
['FooBar', 'RED']
```

This call to `do_something` raised `TypeError` due to first arg not being a subclass of `FooBar` or `ObjFoo`.

```
>>> print(do_something(Foo(), Color.RED))
TypeError: Arg in 1st position is expected to be of a subclass of 'ObjFoo' or 'FooBar'.
SubClass decorator error.
```

Decorating with Key, Value

Decorating when a function has key, value pairs for arguments is the same pattern as `*args`. SubClass type one matches position one of function. SubClass type two matches postiion two of function etc...

```
from kwhelp.decorator import SubClass

@SubClass(Foo, ObjFoo, Color)
def do_something(first, last, color=Color.GREEN):
    return str(first), str(last) , str(color)
```

```
>>> print(do_something(last=ObjFoo(), first=Foo()))
('Foo', 'ObjFoo', 'GREEN')
```

```
>>> print(do_something(last=ObjFoo(), first=1))
TypeError: Arg 'first' is expected be a subclass of 'Foo'.
SubClass decorator error.
```

Primitive Types

In python numbers and str instances are classes. *SubClass* can also be used to test for numbers and strings.

```
@SubClass(int, (int, float), str)
def do_something(first, last, end):
    return first, last , end
```

```
>>> print(do_something(1, 17, "!!!"))
(1, 17, '!!!')
>>> do_something(1, 44.556, "!!!")
(1, 44.556, '!!!')
>>> print(do_something(1, 44.556))
ValueError: Invalid number of arguments for do_something()
SubClass decorator error.
>>> print(do_something(1, 44.556, 10))
TypeError: Arg 'end' is expected be a subclass of 'str'.
SubClass decorator error.
```

Option opt_all_args

`opt_all_args` argument allows the last class type passed into *SubClass* to validate all remaining arguments of wrapped function.

For more examples see *opt_all_args*.

```
@SubClass(float, (float, int), opt_all_args=True)
def sum_num(*args):
    return sum(args)
```

The first arg of `sum_num` must be a `float`. Remaining args can be `float` or `int`.

```
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88))
130.736
>>> print(sum_num(1, 44.556, 10, 22, 45, 7.88))
TypeError: Arg in 1st position is expected to be of a subclass of 'float'.
SubClass decorator error.
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88, "77"))
TypeError: Arg in 7th position is expected to be of a subclass of 'float' or 'int'.
SubClass decorator error.
```

Opton opt_args_filter

The arguments are validated by *SubClass* can be filtered by setting `opt_args_filter` option.

For more examples see *opt_args_filter*.

In the following example all `*args` must of or derived from clase `Base`. `opt_args_filter=DecArgEnum.ARGS` filters `SubClass` to only process `*args`.

```
from kwhelp.decorator import SubClass, DecArgEnum

@SubClass(Base, opt_all_args=True, opt_args_filter=DecArgEnum.ARGS)
def foo(*args, msg: str):
    result = [str(t) for t in args]
    return msg + ', '.join(result)
```

```
>>> result = foo(Foo(), Bar(), FooBar(), ObjFoo(), msg='Summary: ')
>>> print(result)
Summary: Foo, Bar, FooBar, Foo
>>> result = foo(Foo(), Bar(), FooBar(), ObjFoo(), msg='Summary: ')
TypeError: Arg in 4th position is expected to be of a subclass of 'Base'.
SubClass decorator error.
```

Combined Decorators

SubClass can be combined with other decorators.

The following example limits how many args are allowed by applying *ArgsMinMax* decorator.

```
from kwhelp.decorator import SubClass, ArgsMinMax

@ArgsMinMax(max=6)
@SubClass(float, (float, int), opt_all_args=True)
def sum_num(*args):
    return sum(args)
```

```
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88))
130.736
>>> print(sum_num(1, 44.556, 10, 22, 45, 7.88, 100))
ValueError: Invalid number of args pass into 'sum_num'.
Expected max of '6'. Got '7' args.
ArgsMinMax decorator error.
```

SubClasskKw Usage

SubClasskKw decorator that requires args of a function to match or be a subclass of types specificed in constructor. Each arg can have seperate rules applied.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*
- *type_instance_check*

SubClasskKw constructor args `arg_info` and `rules` work together. `arg_info` is a dictionary with a key of `str` that matches an arg name of the function that is being decorated. `arg_info` value is one of the following:

- `int` is an index of an item in `rules`
- `type` a type to match

`arg_info` can be *mixed*.

`types` is a list of type to match.

Example Usage

Sample Classes

```
from enum import IntEnum, auto

class Color(IntEnum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()

    def __str__(self) -> str:
        return self._name_

class Base:
    def __str__(self) -> str:
        return self.__class__.__name__
class Obj:
    def __str__(self) -> str:
        return self.__class__.__name__
class Foo(Base): pass
class Bar(Foo): pass
class FooBar(Bar): pass
class ObjFoo(Obj): pass
```

SubClasskKw decorated function.

```
from kwhelp.decorator import SubClasskKw

@SubClasskKw(arg_info={"first": 0, "second": 0, "obj": 0, "color": 1},
             types=[(Foo, Obj), Color])
```

(continues on next page)

(continued from previous page)

```
def myfunc(first, second, **kwargs):
    color = kwargs.get("color", Color.BLUE)
    return (str(first), str(second), str(kwargs['obj']), str(color))
```

```
>>> result = myfunc(first=Foo(), second=ObjFoo(), obj=FooBar())
>>> print(result)
('Foo', 'ObjFoo', 'FooBar', 'BLUE')
```

```
>>> result = myfunc(first=Foo(), second=ObjFoo(), color=Color.RED, obj=FooBar())
>>> print(result)
('Foo', 'ObjFoo', 'FooBar', 'RED')
```

If types fail validation then a `TypeError` is raised.

```
>>> result = myfunc(first=Foo(), second=ObjFoo(), color=1, obj=FooBar())
TypeError: Arg 'color' is expected to be of a subclass of 'Color'.
SubClasskKw decorator error.
```

SubClasskKw `arg_info` contains types and indexes. Types of `arg_info` are requied to match function arguments directly. Indexes are an index of `types` that match function arguments.

```
from kwhelp.decorator import SubClasskKw

@SubClasskKw(arg_info={"first": 0, "second": 0, "obj": 0, "color": Color},
             types=[(Foo, Obj), Color])
def myfunc(first, second, **kwargs):
    color = kwargs.get("color", Color.BLUE)
    return (str(first), str(second), str(kwargs['obj']), str(color))
```

```
>>> result = myfunc(first=Foo(), second=ObjFoo(), obj=FooBar())
>>> print(result)
('Foo', 'ObjFoo', 'FooBar', 'BLUE')
```

```
>>> result = myfunc(first=Foo(), second=ObjFoo(), color=1, obj=FooBar())
TypeError: Arg 'color' is expected to be of a subclass of 'Color'.
SubClasskKw decorator error.
```

Primitive Types

In python numbers and str instances are classes. *SubClasskKw* can also be used to test for numbers and strings.

```
from kwhelp.decorator import SubClasskKw

@SubClasskKw(arg_info={"first": 0, "second": 0, "obj": 0, "last": 1},
             types=[(int, float), str])
def myfunc(first, second, **kwargs):
    last = kwargs.get("last", "The End!")
    return (first, second, kwargs['obj'], last)
```

```
>>> result = myfunc(first=22.55, second=555, obj=-12.45, last="!!!")
>>> print(result)
(22.55, 555, -12.45, '!!!')
```

```
>>> result = myfunc(first=22.55, second=555, obj=None, last="!!!")
>>> print(result)
TypeError: Arg 'obj' is expected to be of a subclass of 'float' or 'int'.
SubClasskKw decorator error.
```

TypeCheck Usage

TypeCheck decorator reqires that each args of a function match a `type`.

Includes features:

- *ftype*
- *opt_args_filter*
- *opt_logger*
- *opt_return*
- *raise_error*
- *type_instance_check*

Decorating with `*args`

This example requires that all args `int` or `float`.

```
from kwhelp.decorator import TypeCheck

@TypeCheck(int, float)
def add_numbers(*args) -> float:
    result = 0.0
    for arg in args:
        result += float(arg)
    return result
```

Adding numbers works as expected.

```
>>> result = add_numbers(1, 4, 6.9, 3.9, 7.3)
>>> print(result)
23.1
```

Types dictate that if a type is not `int` or `float` then an error will be raised.

```
>>> result = add_numbers(2, 1.2, "4")
TypeError: Arg Value is expected to be of 'float' or 'int' but got 'str'.
TypeCheck decorator error.
```

Decorating with Key, Value

This example requires that all args are `int` or `float`.

```
from kwhelp.decorator import TypeCheck

@TypeCheck(int, float)
def speed_msg(speed, limit, **kwargs) -> str:
    if limit > speed:
        msg = f"Current speed is '{speed}'. You may go faster as the limit is '{limit}'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. You are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and you are currenlty going '{speed}
↪'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'"
    return msg
```

Adding positive numbers works as expected.

```
>>> result = speed_msg(speed=45, limit=60)
>>> print(result)
Current speed is '45'. You may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=66, limit=60, hours=4.7)
>>> print(result)
Please slow down limit is '60' and you are currenlty going '66'. Current driving hours␣
↪is '4.7
```

Types dictate that if a type is not `int` or `float` then an error will be raised.

```
>>> result = speed_msg(speed=45, limit="Fast")
TypeError: Arg 'limit' is expected to be of 'float' or 'int' but got 'str'.
TypeCheck decorator error.
```

Opton opt_args_filter

The arguments are validated by *TypeCheck* can be filtered by setting `opt_args_filter` option.

For more examples see *opt_args_filter*.

Single TypeCheck

In the following example all `*args` must of of type `float` or `int`. `opt_args_filter=DecArgEnum.ARGS` filters `TypeCheck` to only process `*args`.

```
from kwhelp.decorator import TypeCheck, DecArgEnum

@TypeCheck(float, int, opt_args_filter=DecArgEnum.ARGS)
def sum_num(*args, msg: str):
```

(continues on next page)

(continued from previous page)

```
    _sum = sum(args)
    return msg + str(_sum)
```

```
>>> result = sum_num(102, 2.45, 34.55, -24, 5.8, -6, msg='Total: ')
>>> Total: 114.8
Total: 21
>>> sum_num(102, "two", 34.55, -24, 5.8, -6, msg='Total: ')
TypeError: Arg Value is expected to be of 'float' or 'int' but got 'str'.
TypeCheck decorator error.
```

Multi TypeCheck

By combining `TypeCheck` decorators with different `opt_args_filter` settings it is possible to required diferent types for `*args`, `**kwargs` and Named Args.

```
from kwhelp.decorator import TypeCheck, DecArgEnum

@TypeCheck(str, opt_args_filter=DecArgEnum.NAMED_ARGS)
@TypeCheck(float, int, opt_args_filter=DecArgEnum.ARGS)
def sum_num(*args, msg: str):
    _sum = sum(args)
    return msg + str(_sum)
```

```
>>> result = sum_num(102, 2.45, 34.55, -24, 5.8, -6, msg='Total: ')
>>> Total: 114.8
Total: 21
>>> sum_num(102, "two", 34.55, -24, 5.8, -6, msg='Total: ')
TypeError: Arg Value is expected to be of 'float' or 'int' but got 'str'.
TypeCheck decorator error.
>>> sum_num(102, 2.45, 34.55, -24, 5.8, -6, msg=22)
TypeError: Arg 'msg' is expected to be of 'str' but got 'int'.
TypeCheck decorator error.
```

TypeCheckKw Usage

TypeCheckKw decorator allows each arg of a function match one of the **types** specified. Each arg can have seperate rules applied.

Includes features:

- *ftype*
- *opt_logger*
- *opt_return*
- *raise_error*
- *type_instance_check*

TypeCheckKw constructor args `arg_info` and `rules` work together. `arg_info` is a dictionary with a key of `str` that matches an arg name of the function that is being decorated. `arg_info` value is one of the following:

- `int` is an index of an item in `rules`
- `type` a type to match

`arg_info` can be *mixed*.

`types` is a list of type to match.

Example Usage

TypeCheckKw decorated function.

```
from kwhelp.decorator import TypeCheckKw

@TypeCheckKw(arg_info={"speed": 0, "limit": 0, "hours": 0, "name": 1},
                types=[(int, float), str])
def speed_msg(speed, limit, **kwargs) -> str:
    name = kwargs.get('name', 'You')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

```
>>> result = speed_msg(speed=45, limit=60)
>>> print(result)
Current speed is '45'. You may go faster as the limit is '60'.
```

```
>>> result = speed_msg(speed=45, limit=60, name="John")
>>> print(result)
Current speed is '45'. John may go faster as the limit is '60'.
```

If types fail validation then a `TypeError` is raised.

```
>>> result = speed_msg(speed=-2, limit=60, name=17, hours=5)
TypeError: Arg 'name' is expected to be of 'str' but got 'int'.
TypeCheckKw decorator error.
```

TypeCheckKw `arg_info` contains types and indexes. Types of `arg_info` are requied to match function arguments directly. Indexes are an index of `types` that match function arguments.

```
from kwhelp.decorator import TypeCheckKw

@TypeCheckKw(arg_info={"speed": 0, "limit": 0, "hours": 0, "name": str},
                types=[(int, float)])
def speed_msg(speed, limit, **kwargs) -> str:
    name = kwargs.get('name', 'You')
    if limit > speed:
        msg = f"Current speed is '{speed}'. {name} may go faster as the limit is '{limit}
↪'."
```

(continues on next page)

(continued from previous page)

```
    elif speed == limit:
        msg = f"Current speed is '{speed}'. {name} are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and current speed is '{speed}'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'."
    return msg
```

Callcounter Usage

callcounter Decorator method that adds `call_count` attribute to decorated method.

`call_count` is `0` if method has not been called.
`call_count` increases by 1 each time method is been called.

```
from kwhelp.decorator import calltracker

@callcounter
def foo(msg):
    print(msg)
```

```
>>> print("Call Count:", foo.call_count)
0
>>> foo("Hello")
Hello
>>> print("Call Count:", foo.call_count)
1
>>> foo("World")
World
>>> print("Call Count:", foo.call_count)
2
```

Note: This decorator needs to be the topmost decorator applied to a method

Calltracker Usage

calltracker decorator method that adds `has_been_called` attribute to decorated method.

`has_been_called` is `False` if method has not been called.
`has_been_called` is `True` if method has been called.

```
from kwhelp.decorator import calltracker

@calltracker
def foo(msg):
    print(msg)
```

```
>>> print(foo.has_been_called)
False
>>> foo("Hello World")
Hello World
>>> print(foo.has_been_called)
True
```

Note: This decorator needs to be the topmost decorator applied to a method

Singleton Usage

singleton decorator that makes a class a singleton class

Example class that becomes a singleton class onece attribute is applied.

```
from kwhelp.decorator import singleton, RuleCheckAll
from kwhelp import rules

@singleton
class Logger:
    @RuleCheckAll(rules.RuleStrNotNullEmptyWs, ftype=DecFuncEnum.METHOD)
    def log(self, msg):
        print(msg)
```

All created instances are the same instance.

```
>>> logger1 = Logger()
>>> logger2 = Logger()
>>> print(logger1 is logger1)
True
```

See also:

RuleCheckAll Usage

Example

The following example ensures all function args are a positive `int` or a positive `float`.

```
from kwhelp.decorator import RuleCheckAny
import kwhelp.rules as rules

@RuleCheckAny(rules.RuleIntPositive, rules.RuleFloatPositive)
def speed_msg(speed, limit, **kwargs) -> str:
```

(continues on next page)

(continued from previous page)

```
    if limit > speed:
        msg = f"Current speed is '{speed}'. You may go faster as the limit is '{limit}'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. You are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and you are currenlty going '{speed}
↪'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'"
    return msg
```

2.2 Decorators

Decorators can be used to validate functions.

See also:

Decorator Usage

2.2.1 Example

The following example ensures all function args are a positive `int` or a positive `float`.

```
from kwhelp.decorator import RuleCheckAny
import kwhelp.rules as rules

@RuleCheckAny(rules.RuleIntPositive, rules.RuleFloatPositive)
def speed_msg(speed, limit, **kwargs) -> str:
    if limit > speed:
        msg = f"Current speed is '{speed}'. You may go faster as the limit is '{limit}'."
    elif speed == limit:
        msg = f"Current speed is '{speed}'. You are at the limit."
    else:
        msg = f"Please slow down limit is '{limit}' and you are currenlty going '{speed}
↪'."
    if 'hours' in kwargs:
        msg = msg + f" Current driving hours is '{kwargs['hours']}'"
    return msg
```

2.2.2 Features

ftype

`ftype` option is *DecFuncEnum* argument. `ftype` sets the type of `function`, `method`, `property` etc.

Function

Default for `ftype` is function. Therefore it is not necessary to set ftype on functions.

```
@RuleCheckAll(rules.RuleIntPositive)
def fib(n: int):
    def _fib(_n, memo={}):
        if _n in memo: return memo[_n]
        if _n <= 2: return 1
        memo[_n] = _fib(_n - 1, memo) + _fib(_n - 2, memo)
        return memo[_n]
    return _fib(n)
```

Run function.

```
>>> print(fib(7))
8
>>> print(fib(8))
13
>>> print(fib(6))
21
>>> print(fib(50))
2586269025
>>> print(fib(-50))
kwhelp.exceptions.RuleError: RuleError: 'fib' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
>>> print(fib("3"))
kwhelp.exceptions.RuleError: RuleError: 'fib' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: TypeError: Argument Error: 'arg' is expecting type of 'int'. Got␣
↪type of 'str'
```

Class

For class method `ftype` is set to `DecFuncEnum.METHOD`

```
class Calc:
    @RuleCheckAll(rules.RuleIntPositive, ftype=DecFuncEnum.METHOD)
    def fib(self, n: int):
        def _fib(_n, memo={}):
            if _n in memo: return memo[_n]
            if _n <= 2: return 1
            memo[_n] = _fib(_n - 1, memo) + _fib(_n - 2, memo)
            return memo[_n]
        return _fib(n)
```

```
>>> c = Calc()
>>> print(c.fib(7))
8
```

(continues on next page)

(continued from previous page)

```
>>> print(c.fib(8))
13
>>> print(c.fib(6))
21
>>> print(c.fib(50))
2586269025
>>> print(c.fib(-50))
kwhelp.exceptions.RuleError: RuleError: 'fib' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
```

Class Property

For class property `ftype` is set to `DecFuncEnum.PROPERTY_CLASS`

```
class Foo:

    @RuleCheckAny(RuleIntPositive, RuleFloatPositive, ftype=DecFuncEnum.METHOD)
    def __init__(self, test_value) -> None:
        self._test = test_value

    @property
    def test(self):
        return self._test

    @test.setter
    @RuleCheckAny(RuleIntPositive, RuleFloatPositive, ftype=DecFuncEnum.PROPERTY_CLASS)
    def test(self, value):
        self._test = value
```

```
>>> f = Foo(test_value=22)
>>> print(f.test)
22
>>> f.test = 127.899
>>> print(f.test)
127.899
>>> f.test = -33
kwhelp.exceptions.RuleError: RuleError: 'test' error.
Rule 'RuleIntPositive' Failed validation.
Expected at least one of the following rules to match: RuleIntPositive,␣
↪RuleFloatPositive.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
```

Class staticmethod

For staticmethod method `ftype` is set to `DecFuncEnum.METHOD_STATIC`

```
class Calc:
    @staticmethod
    @RuleCheckAll(rules.RuleIntPositive, ftype=DecFuncEnum.METHOD_STATIC)
    def fib(n: int):
        def _fib(_n, memo={}):
            if _n in memo: return memo[_n]
            if _n <= 2: return 1
            memo[_n] = _fib(_n - 1, memo) + _fib(_n - 2, memo)
            return memo[_n]
        return _fib(n)
```

```
>>> print(Calc.fib(7))
8
>>> print(Calc.fib(8))
13
>>> print(Calc.fib(6))
21
>>> print(Calc.fib(50))
2586269025
```

Class classmethod

For classmethod `ftype` is set to `DecFuncEnum.METHOD_CLASS`

```
class Calc:
    @classmethod
    @RuleCheckAll(rules.RuleIntPositive, ftype=DecFuncEnum.METHOD_CLASS)
    def fib(cls, n: int):
        def _fib(_n, memo={}):
            if _n in memo: return memo[_n]
            if _n <= 2: return 1
            memo[_n] = _fib(_n - 1, memo) + _fib(_n - 2, memo)
            return memo[_n]
        return _fib(n)
```

```
>>> print(Calc.fib(7))
8
>>> print(Calc.fib(8))
13
>>> print(Calc.fib(6))
21
>>> print(Calc.fib(50))
2586269025
```

opt_all_args

`opt_all_args` argument allows the last paramater passed into `*args` to validate all remaining arguments of wrapped function.

```
from kwhelp.decorator import AcceptedTypes

@AcceptedTypes(float, (float, int), opt_all_args=True)
def sum_num(*args):
    return sum(args)
```

The first arg of `sum_num` must be a `float`. Remaining args can be `float` or `int`.

```
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88))
130.736
>>> print(sum_num(1, 44.556, 10, 22, 45, 7.88))
TypeError: Arg in 1st position of is expected to be of '<class 'float'>' but got 'int'
AcceptedTypes decorator error.
>>> print(sum_num(1.3, 44.556, 10, 22, 45, 7.88, "77"))
TypeError: Arg in 3rd position of is expected to be of '(<class 'float'>, <class 'int'>)
↪' but got 'str'
AcceptedTypes decorator error.
```

opt_args_filter

`opt_args_filter` allows decorators that support it filter which type of argument will be validated.

The following example is with *AcceptedTypes* but is similar for all decorators that support `opt_args_filter` option.

ARGS

`DecArgEnum.ARGS` filter. Only `*args` are validated.

```
from kwhelp.decorator import AcceptedTypes, DecArgEnum

@AcceptedTypes(int, opt_all_args=True, opt_args_filter=DecArgEnum.ARGS)
def foo(*args, first, last, **kwargs):
    return [*args] + [first, last] + [v for _, v in kwargs.items()]
```

```
>>> result = foo(1, 2, 3, 4, 5, 6, first="one", one="1st", two="2nd", third="3rd")
>>> print(result)
[1, 2, 3, 4, 5, 6, 'one', '!!!', '1st', '2nd', '3rd']
>>> foo(1, 45.66, 3, 4, 5, 6, first="one", one="1st", two="2nd", third="3rd")
TypeError: Arg in 2nd position of is expected to be of 'int' but got 'float'.
AcceptedTypes decorator error.
```

NAMED_ARGS

DecArgEnum.NAMED_ARGS filter. Only named args are validated. *args and **kwargs are excluded.

```
from kwhelp.decorator import AcceptedTypes, DecArgEnum

@AcceptedTypes(int, int, opt_args_filter=DecArgEnum.NAMED_ARGS)
def foo(*args, first, last, **kwargs):
    return [*args] + [first, last] + [v for _, v in kwargs.items()]
```

```
>>> result = foo("a", "b", "c", first=1, last= 3, one="1st", two="2nd", third="3rd")
>>> print(result)
['a', 'b', 'c', 1, 3, '1st', '2nd', '3rd']
>>> result = foo("a", "b", "c", first=1.5, last=3, one="1st", two="2nd", third="3rd")
TypeError: Arg 'first' in 1st position is expected to be of 'int' but got 'float'.
AcceptedTypes decorator error.
```

KWARGS

DecArgEnum.KWARGS filter. Only **kwargs are validated. *args and named args are excluded.

```
from kwhelp.decorator import AcceptedTypes, DecArgEnum

@AcceptedTypes((int, float), opt_all_args=True, opt_args_filter=DecArgEnum.KWARGS)
def foo(*args, first, last="!", **kwargs):
    return [*args] + [first, last] + [v for _, v in kwargs.items()]
```

```
>>> result = foo("a", "b", "c", first=-100, one=101, two=2.2, third=33.33)
>>> print(result)
['a', 'b', 'c', -100, '!', 101, 2.2, 33.33]
>>> result = foo("a", "b", "c", first=-100, one=101, two="two", third=33.33)
TypeError: Arg 'two' in 2nd position is expected to be of 'float' or 'int' but got 'str'.
AcceptedTypes decorator error.
```

NO_ARGS

DecArgEnum.NO_ARGS filter. Only all args except for *args.

```
from kwhelp.decorator import AcceptedTypes, DecArgEnum

@AcceptedTypes((int, float), opt_all_args=True, opt_args_filter=DecArgEnum.NO_ARGS)
def foo(*args, first, last=1001, **kwargs):
    return [*args] + [first, last] + [v for _, v in kwargs.items()]
```

```
>>> result = foo("a", "b", "c", first=-100, one=101, two=22.22, third=33.33)
>>> print(result)
['a', 'b', 'c', -100, 1001, 101, 22.22, 33.33]
>>> result = foo("a", "b", "c", first=-100, one="1st", two=22.22, third=33.33)
TypeError: Arg 'one' in 3rd position is expected to be of 'float' or 'int' but got 'str'.
AcceptedTypes decorator error
```

(continues on next page)

(continued from previous page)

```
>>> result = foo("a", "b", "c", first=-100, one=101.11, two="2nd", third=33.33)
TypeError: Arg 'two' in 4th position is expected to be of 'float' or 'int' but got 'str'.
AcceptedTypes decorator error.
```

opt_logger

`opt_logger` option requires a logger. When a logger is passed in then the decorator will log to the logger when validation fails.

Note: When *opt_return* option is set and/or *raise_error* is `False` then logging will **not** take place as no validation error are raised.

Example logger and function with a decorator that uses logger.

```
import logging
from pathlib import Path
from kwhelp.decorator import RuleCheckAll
from kwhelp.rules import RuleIntPositive

def _create_logger(level:int, log_path: Path) -> logging.Logger:
    """
    Creates a logging object and returns it
    """
    logger = logging.getLogger("default_logger")
    logger.setLevel(level)
    # create the logging file handler
    fh = logging.FileHandler(str(log_path))

    fmt = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    formatter = logging.Formatter(fmt)
    fh.setFormatter(formatter)

    # add handler to logger object
    logger.addHandler(fh)
    return logger

default_logger = _create_logger(level=logging.DEBUG ,log_path=(Path.home() / 'mylog.log
↪'))

@RuleCheckAll(RuleIntPositive, opt_logger=default_logger)
def add_positives(*args) -> int:
    return sum(args)
```

Call sample function.

```
>>> try:
>>>     result = add_positives(1, 33, -3)
>>>     print(result)
>>> except Exception as e:
>>>     print(e)
```

(continues on next page)

(continued from previous page)

```
RuleError: 'add_positives' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
```

Listing 3: Log file contents

```
2021-11-18 07:58:17,995 - default_logger - ERROR - RuleError: 'add_positives' error.
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
RuleCheckAll decorator error.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
Traceback (most recent call last):
File "/home/paul/Documents/Projects/Python/Publish/kwargs/kwhelp/checks/__init__.py",␣
↪line 237, in _validate_rules_all
    result = result & rule_instance.validate()
File "/home/paul/Documents/Projects/Python/Publish/kwargs/kwhelp/rules/__init__.py",␣
↪line 347, in validate
    raise ValueError(
ValueError: Arg error: 'arg' must be a positive int value

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
File "/home/paul/Documents/Projects/Python/Publish/kwargs/kwhelp/decorator/__init__.py",␣
↪line 1648, in wrapper
    is_valid = self._rulechecker.validate_all(**arg_name_values)
File "/home/paul/Documents/Projects/Python/Publish/kwargs/kwhelp/checks/__init__.py",␣
↪line 311, in validate_all
    result = result & self._validate_rules_all(key=k, field=k, value=v)
File "/home/paul/Documents/Projects/Python/Publish/kwargs/kwhelp/checks/__init__.py",␣
↪line 240, in _validate_rules_all
    raise RuleError(
kwhelp.exceptions.RuleError: RuleError:
Rule 'RuleIntPositive' Failed validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: ValueError: Arg error: 'arg' must be a positive int value
```

opt_return

Many decorators supports *optoinal* `opt_return` option. `opt_return` sets what will be be returned if validation fails. Noramlly decorators will raise an error if validation fails.

In the following example by setting `opt_return` to `False` then when validation fails `False` will be returned instead of an error being raised.

```
from kwhelp.decorator import ArgsLen, RuleCheckAll
from kwhelp import rules

@ArgsLen(2, opt_return=False)
```

(continues on next page)

(continued from previous page)

```
@RuleCheckAll(rules.RuleStrNotNullEmptyWs, opt_return=False)
def is_ab(*args):
    if args[0].lower() == "a" and args[1].lower() == "b":
        return True
    return False
```

```
>>> print(is_ab())
False
>>> print(is_ab(1))
False
>>> print(is_ab(1, 2))
False
>>> print(is_ab('ab'))
False
>>> print(is_ab('a', 'b', 'c'))
False
>>> print(is_ab('a', 'b'))
True
>>> print(is_ab('A', 'B'))
True
>>> print(is_ab(["a", "b"]))
False
>>> print(is_ab(*["a", "b"]))
True
```

raise_error

Some decorators supports *optoinal* `raise_error` bool option. `raise_error` sets if a an error will be validation fails.

If `raise_error` is set to `False` then a attribute will be added to the decorated funciton.

```
from kwhelp.decorator import RuleCheckAll
from kwhelp import rules

@RuleCheckAll(rules.RuleIntPositive, raise_error=False)
def add_positives(*args) -> float:
    result = 0
    for arg in args:
        result += arg
    return result
```

```
>>> print(add_positives.is_rules_all_valid)
True
>>> result = add_positives(2, -4)
>>> print(add_positives.is_rules_all_valid)
False
>>> result = add_positives(2, 4)
>>> print(add_positives.is_rules_all_valid)
True
>>> print(result)
6
```

type_instance_check

Many `type` validation decorators supports *optoinal* `type_instance_check` bool option.

True

When `type_instance_check` is `True` then if type does not match then an instance check will also be done.

```
from pathlib import Path
from kwhelp.decorator import AcceptedTypes

@AcceptedTypes(Path, type_instance_check=True)
def is_file_exist(arg):
    return True
```

```
>>> p = Path("home", "user")
>>> print(is_file_exist(arg=p))
True
```

False

```
from pathlib import Path
from kwhelp.decorator import AcceptedTypes

@AcceptedTypes(Path, type_instance_check=False)
def is_file_exist(arg):
    return True
```

```
>>> p = Path("home", "user")
>>> print(is_file_exist(arg=p))
TypeError: Arg 'arg' in 1st position is expected to be of '<class 'pathlib.Path'>' but␣
↪got 'PosixPath'
```

Explanation

In some cases such as **Mulitple Inheritance** type check may not pass

In the following example type check for `str` is `True` however, `Path` is false.

```
>>> from pathlib import Path
>>> s = ""
>>> print(type(s) is str)
True
>>> p = Path("home", "user")
>>> print(type(p) is Path)
False
```

In the following example instance check for `Path` is `True`

```
>>> from pathlib import Path
>>> p = Path("home", "user")
>>> print(isinstance(p, Path))
False
```

2.2.3 Decorator Index

- *AcceptedTypes*
- *ArgsLen*
- *ArgsMinMax*
- *AutoFill*
- *AutoFillKw*
- *DefaultArgs*
- *RequireArgs*
- *ReturnRuleAll*
- *ReturnRuleAny*
- *ReturnType*
- *RuleCheckAll*
- *RuleCheckAllKw*
- *RuleCheckAny*
- *RuleCheckAnyKw*
- *SubClass*
- *SubClasskKw*
- *TypeCheck*
- *TypeCheckKw*
- *callcounter*
- *calltracker*
- *singleton*

2.3 Rules

Rules allow validation of `kwargs` values.

2.3.1 IRule

New rules can be created by inheriting from *IRule* interface/class

Example Rule:

```
from kwhelp.rules import IRule
class RuleIntRangeZeroNine(IRule):
    '''
    Rule to ensure a integer from 0 to 9.
    '''
    def validate(self) -> bool:
        if not isinstance(self.field_value, int):
            return False
        if self.field_value < 0 or self.field_value > 9:
            if self.raise_errors:
                raise ValueError(
                    f"Arg error: '{self.key}' must be a num from 0 to 9")
            return False
        return True
```

2.3.2 Related

- *KwargsHelper Assign Rule Checking*
- *KwArg Kw_assign Rule Checking*
- *Simple Uasage Example*

2.3.3 Included Rules

- *RuleAttrExist*
- *RuleAttrNotExist*
- *RuleBool*
- *RuleByteSigned*
- *RuleByteUnsigned*
- *RuleFloat*
- *RuleFloatNegative*
- *RuleFloatNegativeOrZero*
- *RuleFloatPositive*
- *RuleFloatZero*
- *RuleInt*
- *RuleIntNegative*
- *RuleIntNegativeOrZero*
- *RuleIntPositive*
- *RuleIntZero*

- *RuleIterable*
- *RuleNone*
- *RuleNotIterable*
- *RuleNotNone*
- *RuleNumber*
- *RulePath*
- *RulePathExist*
- *RulePathNotExist*
- *RuleStr*
- *RuleStrEmpty*
- *RuleStrNotNullEmptyWs*
- *RuleStrNotNullOrEmpty*
- *RuleStrPathExist*
- *RuleStrPathNotExist*

2.4 Examples

2.4.1 Using Callbacks

Using callback:

Example:

```
from kwhelp import KwargsHelper, AfterAssignEventArgs, BeforeAssignEventArgs,␣
↪AssignBuilder

class MyClass:
    def __init__(self, **kwargs):
        self._loop_count = -1
        kw = KwargsHelper(originator=self, obj_kwargs={**kwargs})
        ab = AssignBuilder()
        kw.add_handler_before_assign(self._arg_before_cb)
        kw.add_handler_after_assign(self._arg_after_cb)
        ab.append(key='exporter', require=True, types=[str])
        ab.append(key='name', require=True, types=[str],
                default='unknown')
        ab.append(key='file_name', require=True, types=[str])
        ab.append(key='loop_count', types=[int],
                default=self._loop_count)
        result = True
        # by default assign will raise errors if conditions are not met.
        for arg in ab:
            result = kw.assign_helper(arg)
            if result == False:
                break
```

(continues on next page)

(continued from previous page)

```
        if result == False:
            raise ValueError("Error parsing kwargs")

    def _arg_before_cb(self, helper: KwargsHelper,
                       args: BeforeAssignEventArgs) -> None:
        # callback function before value assigned to attribute
        if args.key == 'loop_count' and args.field_value < 0:
            # cancel will raise CancelEventError unless
            # KwargsHelper constructor has cancel_error=False
            args.cancel = True
        if args.key == 'name' and args.field_value == 'unknown':
            args.field_value = 'None'

    def _arg_after_cb(self, helper: KwargsHelper,
                      args: AfterAssignEventArgs) -> None:
        # callback function after value assigned to attribute
        if args.key == 'name' and args.field_value == 'unknown':
            raise ValueError(
                f"{args.key} This should never happen. value was suppose to be reassigned
↪")

    @property
    def exporter(self) -> str:
        return self._exporter

    @property
    def file_name(self) -> str:
        return self._file_name

    @property
    def name(self) -> str:
        return self._name

    @property
    def loop_count(self) -> int:
        return self._loop_count
```

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=3)
>>> print(my_class.exporter)
json
>>> print(my_class.file_name)
data.json
>>> print(my_class.name)
None
>>> print(my_class.loop_count)
3
```

See also:

KwargsHelper, *AfterAssignEventArgs*, *BeforeAssignEventArgs*, *AssignBuilder*

2.4.2 Using Callbacks & Rules

Using rules with Callback:

Example:

```
from kwhelp import KwargsHelper, AfterAssignEventArgs, BeforeAssignEventArgs,␣
↪AssignBuilder
import kwhelp.rules as rules

class MyClass:
    def __init__(self, **kwargs):
        self._loop_count = -1
        kw = KwargsHelper(self, {**kwargs})
        ab = AssignBuilder()
        kw.add_handler_before_assign(self._arg_before_cb)
        kw.add_handler_after_assign(self._arg_after_cb)
        ab.append(key='exporter', rules=[rules.RuleStr])
        ab.append(key='name', require=True, rules=[rules.RuleStr],
                default='unknown')
        ab.append(key='file_name', require=True, rules_all=[
            rules.RuleStr, rules.RuleStrNotNullOrEmpty])
        ab.append(key='loop_count', require=True, rules_all=[
                rules.RuleInt, rules.RuleIntPositive],
                default=self._loop_count)
        result = True
        # by default assign will raise errors if conditions are not met.
        for arg in ab:
            result = kw.assign_helper(arg)
            if result == False:
                break
        if result == False:
            raise ValueError("Error parsing kwargs")

    def _arg_before_cb(self, helper: KwargsHelper,
                       args: BeforeAssignEventArgs) -> None:
        if args.key == 'name' and args.field_value == 'unknown':
            args.field_value = 'None'

    def _arg_after_cb(self, helper: KwargsHelper,
                       args: AfterAssignEventArgs) -> None:
        # callback function after value assigned to attribute
        if args.key == 'name' and args.field_value == 'unknown':
            raise ValueError(
                f"{args.key} This should never happen. value was suppose to be reassigned
↪")

    @property
    def exporter(self) -> str:
        return self._exporter

    @property
    def file_name(self) -> str:
        return self._file_name
```

(continues on next page)

(continued from previous page)

```
    @property
    def name(self) -> str:
        return self._name

    @property
    def loop_count(self) -> int:
        return self._loop_count
```

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=3)
>>> print(my_class.exporter)
json
>>> print(my_class.file_name)
data.json
>>> print(my_class.name)
None
>>> print(my_class.loop_count)
3
```

See also:

- *KwargsHelper*,
- *KwargsHelper Assign Rule Checking*
- *KwArg Kw_assign Rule Checking*
- *AfterAssignEventArgs*,
- *BeforeAssignEventArgs*,
- *AssignBuilder*,
- *Rules*
- *Simple Uasage Example*

2.4.3 Simple usage

Example:

Listing 4: Custom Class

```
from kwhelp import KwargsHelper
import kwhelp.rules as rules

class MyClass:
    def __init__(self, **kwargs):
        self._loop_count = -1
        kw = KwargsHelper(self, {**kwargs}, field_prefix='')
        kw.assign(key='exporter', types=[str], default='None')
        kw.assign(key='name', types=[str], default='unknown')
        kw.assign(key='file_name', rules_all=[rules.RuleStrNotNullOrEmpty])
        kw.assign(key='loop_count', rules_all=[rules.RuleIntPositive],
                default=self._loop_count)
```

Listing 5: Simple assignment

```
>>> my_class = MyClass(file_name='data.html', name='Best Doc', loop_count=1)
>>> print(my_class.exporter)
None
>>> print(my_class.file_name)
data.html
>>> print(my_class.name)
Best Doc
>>> print(my_class.loop_count)
1
```

Listing 6: Simple assignment

```
>>> my_class = MyClass(exporter='json', file_name='data.json', loop_count=3)
>>> print(my_class.exporter)
json
>>> print(my_class.file_name)
data.json
>>> print(my_class.name)
None
>>> print(my_class.loop_count)
3
```

Validation Failure

Raises an error because `loop_count` is default is `-1` and *`RuleIntPositive`* is added to rules.

Listing 7: Fails validation example

```
>>> try:
>>>     my_class = MyClass(exporter='html', file_name='data.html', name='Best Doc')
>>> except Exception as e:
>>>    print(e)
RuleError: Argument: 'loop_count' failed validation. Rule 'RuleIntPositive' Failed␣
↪validation.
Expected the following rule to match: RuleIntPositive.
Inner Error Message: ValueError: Arg error: 'loop_count' must be a positive int value
```

See also:

KwargsHelper, *Rules*

2.5 Release Notes

2.5.1 Version 2.7.1

Update `opt_logger` accept `Logger` or a `LoggerAdapter`. See *opt_logger*.

2.5.2 Version 2.7.0

Added option `opt_logger` to many decorators. See *opt_logger*.

2.5.3 Versoin 2.6.0

Added Rules:

- RulePath
- RulePathExist
- RulePathNotExist
- RuleStrPathExist
- RuleStrPathNotExist

2.5.4 Version 2.5.0

Added option `opt_args_filter` to `AcceptedTypes`, `RuleCheckAll`, `RuleCheckAny`, `SubClass`, `TypeCheck` See *opt_args_filter*.

Upgraded underling engine to use OrderedDict to ensure order of keys for python <= 3.6

2.5.5 Version 2.4.0

Added SubClass, SubClassKw decorators.

Added `opt_all_args` feature to `AcceptedTypes` decorator. See *opt_all_args*

Update AcceptedTypes decorator. Now passing enum types into constructor no longer require enum type to be passed in as iterable object.

Updated many decorator error message. Now they are a little more human readable.

2.5.6 Version 2.3.0

Added decorator `ArgsMinMax`

Added Rules:

- RuleIterable
- RuleNotIterable

Added `opt_return` feature to many decorators. See *opt_return*

2.5.7 Version 2.2.1

`ArgsLen` decorator now allows zero length args.

```
@ArgsLen(0, 2)
def foo(*args, **kwargs): pass
```

2.5.8 Version 2.2.0

Added Decorator ArgsLen.

Added Rules:

- RuleByteSigned
- RuleByteUnsigned

2.5.9 Version 2.1.4

Bug fix for `AcceptedTypes` Decorator when function has leading named args before positional args.

The following will now work.

```
@AcceptedTypes(float, str, int, [Color], int, bool)
def myfunc(arg1, arg2, *args, opt=True): pass
```

2.5.10 Version 2.1.3

Update fix for python DeprecationWarning: Using or importing the ABCs from 'collections' instead of from 'collections.abc' is deprecated

Added Install documentation.

Added Development documentation.

2.5.11 Version 2.1.2

Fix for Decorator `AcceptedTypes` not working correctly with optional arguments.

2.5.12 Version 2.1.1

Fix for version 2.1.0 setup not building correctly.

2.5.13 Version 2.1.0

New Features

Added Decorators that provided a large range of options for validating function, class input and return values. Also added decorators that provide other functionality such as singleton pattern.

2.6 Install

2.6.1 PIP

`kwargshelper` from PyPI:
If you use pip, you can install `kwargshelper` with:

```
pip install kwargshelper
```

Also when using pip, it's good practice to use a virtual environment - see *Reproducible Installs* below for why, and this guide for details on using virtual environments.

2.6.2 REPRODUCIBLE INSTALLS

As libraries get updated, results from running your code can change, or your code can break completely. It's important to be able to reconstruct the set of packages and versions you're using. Best practice is to:

1. use a different environment per project you're working on,
2. record package names and versions using your package installer; each has its own metadata format for this:
    - Conda: conda environments and environment.yml
    - Pip: virtual environments and requirements.txt
    - Poetry: virtual environments and pyproject.toml

2.7 Project Development

2.7.1 Environment

`kwargshelper` use a `conda` environment.

Creating environment

Listing 8: terminal

```
# from root dir of project
$ conda env create --prefix env --file environment.yml
```

2.8 kwhelp

2.8.1 AfterAssignAutoEventArgs Class

class kwhelp.**AfterAssignAutoEventArgs**(*key: str*, *originator: object*)
After Assign Auto Event Args

See also:

Auto_assign Callback

__init__(*key: str*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – key arg
- **originator** (*object*) – object that originated event

property canceled: bool
Get if assigment was canceled by before events

property field_name: str
The field that will be assigned representing **key**.

Getter Gets the field that will be assigned representing **key**.

Setter Sets the field that will be assigned representing **key**.

property key: str
Gets the key of the key value pair

property originator: object
Gets object that attributes assigned/modified for

property success: bool
Get assigning of attribue/value succeeded

2.8.2 AfterAssignEventArgs Class

class kwhelp.**AfterAssignEventArgs**(*help_args:* kwhelp.HelperArgs, *originator: object*)
After Assign Event Args

See also:

Callback Usage

__init__(*help_args:* kwhelp.HelperArgs, *originator: object*) → None
Constructor

Parameters

- **help_args** (HelperArgs) – HelperArgs object
- **originator** (*object*) – Origanating object

property canceled: bool
Get if assigment was canceled by before events

property field_name: str
The name of the field that value was assigned

property field_value: object
The value that is assigned to *field_name*

property helper_args: *kwhelp.HelperArgs*
'Get the args used to for modify/creating attribute

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes assigned/modified for

property rules_passed: bool
Get if all applied rules passed

property success: bool
Determinins if assigning of attribue/value succeeded

Getter Get assigning of attribue/value succeeded

Setter Sets assigning of attribue/value succeeded

2.8.3 AssignBuilder Class

class kwhelp.**AssignBuilder**
Helper class for building list to use with "KwargsHelper.Assing() method

__init__() → None
Constructor

append(*key: str*, *field: Optional[str] = None*, *require: bool = False*, *default: Optional[object] = None*, *types: Optional[List[type]] = None*, *rules_all: Optional[List[Callable[[*kwhelp.rules.IRule*], bool]]] = None*, *rules_any: Optional[List[Callable[[*kwhelp.rules.IRule*], bool]]] = None*)
Appends dictionary item of parameters to list

Parameters

- **key** (*str*) – the key of the key, value pair.
- **field** (*Optional[str], optional*) – the name of the field.. Defaults to `None`.
- **require** (*bool, optional*) – Determins if `key` is required. Defaults to `False`.
- **default** (*Optional[object], optional*) – default value to assign if key value is missing. Defaults to `None`.
- **types** (*Optional[List[type]], optional*) – list of one or more types that the value of the key value pair must match. Defaults to `None`.
- **rules** (*Optional[List[Callable[[IRule], bool]]], optional*) – Rules to apply. Defaults to `None`.

Raises

- **TypeError** – if `key` is not instance of``str``.
- **ValueError** – if `key` is empty or whitespace str.
- **ValueError** – if `key` has already exist.

append_helper(*helper:* kwhelp.HelperArgs)
Appends dictionary item of parameters to list

Parameters **helper** (`HelperArgs`) – parameters to append

Raises

- **TypeError** – if `helper` is not instance of `HelperArgs`
- **ValueError** – if `helper.key` is empty string.
- **ValueError** – if `helper.key` already exist.

extend(*other:* kwhelp.AssignBuilder) → None
Extents this instance by merging values from other instance of `AssignBuilder`.

Parameters **other** (`AssignBuilder`) – instance to merge

Raises **NotImplementedError** – if `other` is not instance of `AssignBuilder`.

remove(*item:* kwhelp.HelperArgs) → None
Removes an instance of `HelperArgs` from this instance

Parameters **item** (`HelperArgs`) – Object to remove

Raises **TypeError** – if `item` is not instance of `HelperArgs`

Returns `None`

Return type [obj]

2.8.4 BeforeAssignAutoEventArgs Class

class kwhelp.**BeforeAssignAutoEventArgs**(*key: str*, *value: object*, *field: str*, *originator: object*)
Before Assign Auto Event Args

See also:

Auto_assign Callback

__init__(*key: str*, *value: object*, *field: str*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – Key arg
- **value** (*object*) – Value to be assigned
- **field** (*str*) – Field that `value` is to be assigned to.
- **originator** (*object*) – Object that `value` is to be assigned to.
- **all_rules** (*bool*) – Determines if all rules or any rules are to be matched

property cancel: bool
Cancel arg. If `True` process will be canceled.

Getter Gets cancel arg

Setter Sets cancel arg

property field_name: str
The field that will be assigned representing `key`

Getter Gets the field that will be assigned representing `key`

Setter Setts the field that will be assigned representing `key`

property field_value: object
The value to be blindly assigned to property

Getter Gets the value to be blindly assigned to property

Setter Sets the value to be blindly assigned to property

property key: str
Gets the key of the key value pair

property originator: object
Gets object that attributes assigned/modified for

2.8.5 BeforeAssignEventArgs Class

class kwhelp.**BeforeAssignEventArgs**(*help_args:* kwhelp.HelperArgs, *originator: object*)
Before assign event args

See also:

Callback Usage

__init__(*help_args:* kwhelp.HelperArgs, *originator: object*)
Constructor

Parameters

- **help_args** (HelperArgs) – HelperArgs object
- **originator** (*object*) – Origanating object

property cancel: bool
Cancel arg. If True process will be canceled.

Getter Gets cancel arg

Setter Sets cancel arg

property field_name: str
The name of the field that value will be assigned

Getter Gets the name of the field that value will be assigned.

Setter Sets the name of the field that value will be assigned.

property field_value: object
The value that will be assigned field_name.

Getter Gets the value that will be assigned field_name.

Setter Sets the value that will be assigned field_name.

property helper_args: *kwhelp.HelperArgs*
'Get the args used to for modify/creating attribute

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes assigned/modified for

2.8.6 HelperArgs Class

class kwhelp.**HelperArgs**(*key: str*, ***kwargs*)

Helper class that provides KwArgs arguments

__init__(*key: str*, ***kwargs*)

Constructor

Parameters **key** (*str*) – Key Arg

Keyword Arguments

- **default** (*obj, optional*) – Default arg. Default NO_THING
- **field** (*str, optional*) – field arg. Default None
- **require** (*bool, optional*) – require arg. Default False
- **rules_all** (*Iterable, optional*) – rules_any list. Default Empty List.
- **rules_any** (*Iterable, optional*) – rules_all list. Default Empty List.
- **types** (*set, optional*) – types arg. Default Empty set

to_dict() → dict

Gets a dictionary representation of current instance fields

property default: object

Default Value

Getter Gets default value

Setter Sets default value

property field: Optional[str]

Field Value

Getter Gets field value

Setter Sets field value

property key: str

Key Value

Getter Gets Key Value

Setter Sets key value

property require: bool

Require Value

Getter Gets require value

Setter Sets require value

property rules_all: List[Callable[[*kwhelp.rules.IRule*], bool]]

Rules values

Getter Gets rules_all

Setter Sets rules_all

property rules_any: List[Callable[[*kwhelp.rules.IRule*], bool]]

Rules values

Getter Gets rules_any

Setter Sets rules_any

property types: Set[type]
Types values

Getter Gets types values

Setter Sets types values

2.8.7 KwArg Class

class kwhelp.**KwArg**(***kwargs*)
Class for assigning kwargs to autogen fields with type checking and testing

__init__(***kwargs*)
Constructor

Keyword Arguments **kwargs** (*dict*) – dictionary of args

Example

```
from kwhelp import KwArg

def my_method(**kwargs) -> str:
    kw = KwArg(**kwargs)
    # assign args
    kw.kw_assign(key='first', require=True, types=[int])
    kw.kw_assign(key='second', require=True, types=[int])
    kw.kw_assign(key='msg', types=[str], default='Result:')
    kw.kw_assign(key='end', types=[str])
    _result = kw.first + kw.second
    if kw.is_attribute_exist('end'):
        return_msg = f'{kw.msg} {_result}{kw.end}'
    else:
        return_msg = f'{kw.msg} {_result}'
    return return_msg
```

is_attribute_exist(*attrib_name: str*) → bool
Gets if `attrib_name` exist the current instance.

Use this method when:

- When assigning a key that is not required it may not exist in the current instance.
- When assing key that are required then the field will exist in the current instance.

Parameters **attrib_name** (*str*) – This is usually the `key` value of *kw_assign()* or `field` value of *kw_assign()*.

Returns `True` if `attrib_name` exist in current instance; Otherwise, `False`.

Return type bool

is_key_existing(*key: str*) → bool
Gets if the key exist in current kwargs. Basically shortcut for *key in kwargs*

kw_assign(*key: str, field: Optional[str] = None, require: bool = False, default: Optional[object] = <kwhelp.helper.NoThing object>, types: Optional[List[type]] = None, rules_all: Optional[List[Callable[[kwhelp.rules.IRule], bool]]] = None, rules_any: Optional[List[Callable[[kwhelp.rules.IRule], bool]]] = None*) → bool

Assigns attribute value to current instance passed in to constructor. Attributes automatically.

Parameters

- **key** (*str*) – the key of the key, value pair that is required or optional in `kwargs` passed into to constructor.
- **all_rules** (*bool, optional*) – Determines if all rules or any rules are to be matched. If `True` then all rules included in `rules` must be valid to be considered a success. If `False` then any rule included in `rules` that is valid is considered a success. Default `False`.
- **field** (*str, optional*) – the name of the field to assign a value. if `field` is omitted then field name is built using `key`. If included then `kwargs_helper.field_prefix` will be ignored. Defaults to **Empty string**.

  See also: *Kw_assign field*, *KwArg.kwargs_helper*
- **require** (*bool, optional*) – Determins if `key` is required to be in *kwargs* passed into to constructor. if `default` is passed in then `require` is ignored. Defaults to `False`.

  See also: *Kw_assign Require Arg*
- **default** (*object, optional*) – default value to assign to key attribute if no value is found in `kwargs` passed into to constructor. If `default` is passed in then `require` is ignored. Defaults to `NO_THING`.

  See also: *Kw_assign Default Value*
- **types** (*List[type], optional*) – a type list of one or more types that the value of the key value pair must match. For example if a value is required to be only `str` then `types=[str]`. In this example if value is not type `str` then `TypeError` is raised If value is required to be *str* or *int* then `types=[str, int]`. Defaults to `None`.

  See also: *Kw_assign Type Checking*
- **rules_all** (*List[Callable[[IRule], bool]], optional*) – List of rules that must be passed before assignment can take place. If `types` is included then `types` takes priority over this arg. All rules must validate as `True` before assignment takes place. Defaults to `None`.

  See also: *Kw_assign Rule Checking*
- **rules_any** (*List[Callable[[IRule], bool]], optional*) – List of rules that must be passed before assignment can take place. If `types` is included then `types` takes priority over this arg. Any rule that validates as `True` results in assignment taking place. Defaults to `None`.

  See also: *Kw_assign Rule Checking*

Raises

- ***RuleError*** – If `kwargs_helper.rule_error` is `True` and Validation of `rules_all` or `rules_any` fails.
- **TypeError** – If validation of `types` fails.
- ***ReservedAttributeError*** – if `key` is a reserved keyword
- ***ReservedAttributeError*** – if `field` is a reserved keyword

Returns `True` if attribute assignment is successful; Otherwise, `False`

Return type bool

See also:

- *Kw_assign Default Value*
- *Kw_assign field*
- *Kw_assign Require Arg*
- *Kw_assign Rule Checking*
- *Kw_assign Type Checking*

property kw_unused_keys: Set[str]
Gets any unused keys passed into constructor via `**kwargs`

This would be a set of keys that were never used passed into the constructor.

property kwargs_helper: *kwhelp.KwargsHelper*
Get instance of KwargsHelper used to add fields current instance

2.8.8 KwargsHelper Class

class kwhelp.**KwargsHelper**(*originator: object*, *obj_kwargs: dict*, ***kwargs*)
kwargs helper class. Assigns attributes to class with various checks

Example

```
class MyClass:
    def __init__(self, **kwargs):
        self._msg = ''
        kw = KwargsHelper(self, {**kwargs})
        kw.assign(key='msg', require=True, types=['str'])
        kw.assign(key='length', types=['int'], default=-1)

    @property
    def msg(self) -> str:
        return self._msg

    @property
    def length(self) -> str:
        return self._length
```

```
>>> my_class = MyClass(msg='Hello World')
>>> print(my_class.msg)
Hello World
>>> print(my_class.length)
-1
```

__init__(*originator: object*, *obj_kwargs: dict*, ***kwargs*)
Constructor

Parameters

- **originator** (*object*) – object that attributes are assigned to via `assign()` method. This is usually a class.
- **obj_kwargs** (*dict*) – The dictionary of key value args used to set values of attributes. Often passed in as `obj_kwargs = {**kwargs}`

Keyword Arguments

- **field_prefix** (*str, optional*) – sets the `field_prefix` property. Default _.
- **name** (*str, optional*) – sets the *field_prefix* property. Default is the name of `originator` object.
- **cancel_error** (*bool, optional*) – sets the *cancel_error* property. Default `True`.
- **rule_error** (*bool, optional*) – sets the `rule_error` property. Default `True`.
- **assign_true_not_required** (*bool, optional*) – sets the `assign_true_not_required` property. Default `True`.
- **type_instance_check** (*bool, optional*) – If `True` and `KwargsHelper.assign`() arg `types` is set then values will be tested also for isinstance rather then just type check if type check is `False`. If `False` then values willl only be tested as type. Default `True`

Raises **TypeError** – if any arg is not of the correct type.

add_handler_after_assign(*callback: Callable[[*kwhelp.KwargsHelper, kwhelp.AfterAssignEventArgs*], None]*)

Add handler afer assign

Parameters **callback** (*Callable[['KwargsHelper', AfterAssignEventArgs], None]*) – Callback Method

See also:

Callback Usage

add_handler_after_assign_auto(*callback: Callable[[*kwhelp.KwargsHelper, kwhelp.AfterAssignAutoEventArgs*], None]*)

Adds handler for after assign auto

Parameters **callback** (*Callable[['KwargsHelper', AfterAssignAutoEventArgs], None]*) – Callback Method

See also:

Callback Usage

add_handler_before_assign(*callback: Callable[[*kwhelp.KwargsHelper, kwhelp.BeforeAssignEventArgs*], None]*)

Add handler before assign

Parameters **callback** (*Callable[['KwargsHelper', BeforeAssignEventArgs], None]*) – Callback Method

See also:

Callback Usage

add_handler_before_assign_auto(*callback: Callable[[*kwhelp.KwargsHelper, kwhelp.BeforeAssignAutoEventArgs*], None]*)

Adds handler for before assigne auto

Parameters **callback** (*Callable[['KwargsHelper', BeforeAssignAutoEventArgs], None]*) – Callback Method

See also:

Callback Usage

assign(*key: str, field: Optional[str] = None, require: bool = False, default: Optional[object] = <kwhelp.helper.NoThing object>, types: Optional[Iterable[type]] = None, rules_all: Optional[Iterable[Callable[[kwhelp.rules.IRule], bool]]] = None, rules_any: Optional[Iterable[Callable[[kwhelp.rules.IRule], bool]]] = None*) → bool

Assigns attribute value to `obj` passed in to constructor. Attributes are created if they do not exist.

Parameters

- **key** (*str*) – the key of the key, value pair that is required or optional in `obj_kwargs` passed into to constructor.
- **field** (*str, optional*) – the name of the field to assign a value. If `field` is omitted then field name is built using `instance.field_prefix` + `key`. If included then `instance.field_prefix` will be ignored. Defaults to `None`.

  See also: *Assign Field Value*
- **require** (*bool, optional*) – Determins if `key` is required to be in `obj_kwargs` passed into to constructor. if `default` is passed in then `require` is ignored. Defaults to `False`.

  See also: *Assign Require Arg*
- **default** (*object, optional*) – default value to assign to key attribute if no value is found in `obj_kwargs` passed into to constructor. If `default` is passed in then `require` is ignored. Defaults to `NO_THING` which will result in default being ignored.

  See also: *Assign Default Value*
- **types** (*Iterable[type], optional*) – a type list of one or more types that the value of the key value pair must match. For example if a value is required to be only `str` then `types=[str]`. If value is required to be `str` or `int` then `types=[str, int]`. In this example if value is not type `str` then `TypeError` is raised. If `types` is omitted then a value can be any type unless there is a rule in `rules` that is otherwise. Defaults to `None`.

  See also: *Assign Type Checking*
- **rules_all** (*Iterable[Callable[[IRule], bool]], optional*) – List of rules that must be passed before assignment can take place. If `types` is included then `types` takes priority over this arg. All rules must validate as `True` before assignment takes place. Defaults to `None`.

  See also: *Assign Rule Checking*
- **rules_any** (*Iterable[Callable[[IRule], bool]], optional*) – List of rules that must be passed before assignment can take place. If `types` is included then `types` takes priority over this arg. Any rule that validates as `True` results in assignment taking place. Defaults to `None`.

  See also: *Assign Rule Checking*

Returns `True` if attribute assignment is successful; Otherwise, `False`

Return type bool

Raises

- ***RuleError*** – If *rule_error* is `True` and Validation of `rules_all` or `rules_any` fails.
- **TypeError** – If validation of `types` fails.

See also:

- *Assign Field Value*
- *Assign Default Value*
- *Assign Type Checking*
- *Assign Rule Checking*
- *auto_assign()*

assign_helper(*helper:* kwhelp.HelperArgs) → bool
Assigns attribute value using instance of HelperArgs. See *assign()* method.

Parameters **helper** (HelperArgs) – instance to assign.

Returns True if attribute assignment is successful; Otherwise, False.

Return type bool

auto_assign(*types: Optional[Iterable[type]] = None*, *rules_all: Optional[Iterable[Callable[[*kwhelp.rules.IRule*], bool]]] = None*, *rules_any: Optional[Iterable[Callable[[*kwhelp.rules.IRule*], bool]]] = None*) → bool
Assigns all of the key, value pairs of obj_kwargs passed into constructor to originator, unless the event is canceled in *BeforeAssignAutoEventArgs* then key, value pair will be added automacally to originator.

Parameters

- **types** (*Iterable[type], optional*) – a type list of one or more types that the value of the key value pair must match. For example if all values are required to be only str then types=[str]. If all values are required to be str or int then types=[str, int].

  If types is omitted then values can be any type unless there is a rule in rules that is otherwise. Defaults to None.

  See also: *Assign Type Checking*

- **rules_all** (*Iterable[Callable[[IRule], bool]], optional*) – List of rules that must be passed before assignment can take place. If types is included then types takes priority over this arg. All rules must validate as True before assignment takes place. Defaults to None.

  See also: *Assign Rule Checking*

- **rules_any** (*Iterable[Callable[[IRule], bool]], optional*) – List of rules that must be passed before assignment can take place. If types is included then types takes priority over this arg. Any rule that validates as True results in assignment taking place. Defaults to None.

  See also: *Assign Rule Checking*

Returns True of all key, value pairs are added; Otherwise, False.

Return type bool

Raises

- ***RuleError*** – If *rule_error* is True and Validation of rules fails.
- **TypeError** – If validation of types fails.

Note: Call back events are supported via *add_handler_before_assign_auto()* and *add_handler_after_assign_auto()* methods.

See also:

- *Auto_assign Usage*
- *Auto_assign Callback*
- *assign()*

`is_key_existing`(*key: str*) → bool
Gets if the key exist in kwargs dictionary passed in to the constructor by `obj_kwargs` arg.

Parameters **key** (*str*) – Key value to check for existance.

Returns `True` if `key` exist; Otherwise, `False`.

Return type bool

`property assign_true_not_required: bool`
Determines `assign_true_not_required` Option. If `True` then and a non-required arg is assigned via *assign()*. then *assign()* returns `True` even if the arg failed to be applied. In an `after callback` method set by *add_handler_after_assign()* success in *AfterAssignEventArgs.success* property is `False` if arg was not assigned. Default `True`

Getter Gets option value.

Setter Sets option value.

`property cancel_error: bool`
Determins if an error will be raised if cancel is set in *BeforeAssignEventArgs* of a callback. Default `True`.

Getter Gets cancel_error option.

Setter Sets cancel_error option.

`property field_prefix: str`
Determines the field prefix option. The prefix use when setting attributes: Default: `_`. To add args without a prefix set the value `field_prefix = ''` This parameter is ignored when `field` is used in *assign()* method such as: `assign(msg='hi', field='message')`

Getter Gets the field prefix option.

Setter Sets the field prefix option.

`property kw_args: Dict[str, Any]`
Gets the kwargs dictionary passed in to the constructor by `obj_kwargs` arg

`property name: str`
Determins name option. `name` that represents the `originator` in error messages. Default: `type(originator).__name__`

Getter Gets name option.

Setter Sets name option

`property originator: object`
Gets originator option

object that attributes are assigned to via *assign()* method. This is usually a class.

`property rule_error: bool`
Determins rule_error option. Default `True`

Getter Gets rule_error option.

Setter Sets rule_error option.

property rule_test_before_assign: bool
Determines rule_test_before_assign option. If `True` rule testing will occur before assign value to attribute. If `True` and *rule_error* is `True` then rule errors will prevent assigning value. If `False` then attribute values will be assigned even if rules does not validate. Validation can still fail and errors can still be raised, except now the validation will take place after attribute value has been assigned. Default: `True`

Getter Gets rule_test_before_assign option.

Setter Sets rule_test_before_assign option.

property unused_keys: Set[str]
Gets any unused keys passed into constructor via `obj_kwargs`

This would be a set of keys that were never used passed into the constructor.

2.8.9 Helper Module

class kwhelp.helper.**Formatter**
String Fromat Methods

static get_formated_names(*names: List[str]*, ***kwargs*) → str
Gets a formated string of a list of names

Parameters **names** (*List[str]*) – List of names

Keyword Arguments

- **conj** (*str, optional*) – Conjunction used to join list. Default and.
- **wrapper** (*str, optional*) – String to prepend and append to each value. Default '.

Returns formated such as `'final' and 'end'` or `'one', 'final', and 'end'`

Return type str

static get_formated_types(*types: Iterator[type]*, ***kwargs*) → str
Gets a formated string from a list of types.

Parameters **types** (*Iterator[type]*) – Types to create fromated string.

Keyword Arguments

- **conj** (*str, optional*) – Conjunction used to join list. Default and.
- **wrapper** (*str, optional*) – String to prepend and append to each value. Default '.

Returns Formated String

Return type str

static get_missing_args_error_msg(*missing_names: List[str]*, *name: Optional[str] = ''*)
Get an error message for a list of names.

Parameters

- **missing_names** (*List[str]*) – List of names that generated the error. Such as a list of missing arguments of a function.
- **name** (*Optional[str], optional*) – Function, class, method name. Defaults to "".

Returns Formated string for `missing_names` has elements; Otherwise, empty string is returned.

Return type [type]

static get_ordinal(*num: int*) → str
Returns the ordinal number of a given integer, as a string.

Parameters **num** (*int*) – integer to get ordinal value of.

Returns num as ordinal str. eg. 1 -> 1st, 2 -> 2nd, 3 -> 3rd, etc.

Return type str

static get_star_num(*num: int*) → str
Gets a str in format of `'*#'`, eg: `'*0'`, `'*1'`, `'*2'`.

Parameters **num** (*int*) – int to convert

Returns [str in format of `'*#'`

Return type str

static is_star_num(*name: str*) → bool
Gets if arg name is a match to foramt `'*#'`, eg: `'*0'`, `'*1'`, `'*2'`.

Parameters **name** (*str*) – Name to match

Raises **TypeError** – if name is not a `str` value.

Returns `True` if `arg_name` is a match; Otherwise, `False`

Return type bool

class kwhelp.helper.**NoThing**(**args*, ***kwargs*)
Singleton Class to mimic null

class kwhelp.helper.**Singleton**
Singleton abstrace class

kwhelp.helper.**is_iterable**(*arg: object*, *excluded_types: Iterable[type] = (<class 'str'>,)*) → bool
Gets if `arg` is iterable.

Parameters

- **arg** (*object*) – object to test
- **excluded_types** (*Iterable[type], optional*) – Iterable of type to exlcude. If `arg` matches any type in `excluded_types` then `False` will be returned. Default `(str,)`

Returns `True` if `arg` is an iterable object and not of a type in `excluded_types`; Otherwise, `False`.

Return type bool

Note: if `arg` is of type str then return result is `False`.

Example

```
# non-string iterables
assert is_iterable(arg=("f", "f"))       # tuple
assert is_iterable(arg=["f", "f"])       # list
assert is_iterable(arg=iter("ff"))       # iterator
assert is_iterable(arg=range(44))        # generator
assert is_iterable(arg=b"ff")            # bytes (Python 2 calls this a string)

# strings or non-iterables
```

(continues on next page)

(continued from previous page)

```
assert not is_iterable(arg=u"ff")        # string
assert not is_iterable(arg=44)           # integer
assert not is_iterable(arg=is_iterable)  # function

# excluded_types, optionally exlcude types
from enum import Enum, auto

class Color(Enum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()

assert is_iterable(arg=Color)             # Enum
assert not is_iterable(arg=Color, excluded_types=(Enum, str)) # Enum
```

kwhelp.helper.**NO_THING = <kwhelp.helper.NoThing object>**
Singleton Class instance that represents null object.

2.8.10 checks

RuleChecker Class

class kwhelp.checks.**RuleChecker**(*rules_all: Optional[Iterable[*kwhelp.rules.IRule*]] = None*, *rules_any: Optional[Iterable[*kwhelp.rules.IRule*]] = None*, ***kwargs*)
Class that validates args match a given rule

__init__(*rules_all: Optional[Iterable[*kwhelp.rules.IRule*]] = None*, *rules_any: Optional[Iterable[*kwhelp.rules.IRule*]] = None*, ***kwargs*)
Constructor

Parameters

- **rules_all** (*Iterable[IRule], optional*) – List of rules that must all be matched. Defaults to None.
- **rules_any** (*Iterable[IRule], optional*) – List of rules that any one must be matched. Defaults to None.

Keyword Arguments **raise_error** (*bool, optional*) – If True then rules can raise errors when validation fails. Default False.

Raises

- **TypeError** – If rule_all is not an iterable object
- **TypeError** – If rule_any is not an iterable object

validate_all(**args*, ***kwargs*) → bool
Validates all. All *args and **kwargs must match *rules_all*

Returns True if all *args and **kwargs are valid; Otherwise, False

Return type bool

Raises **Exception** – If *raise_error* is True and validation Fails. The type of exception raised is dependend on the *IRule* that caused validation failure. Most rules raise a ValueError or a TypeError.

validate_any(**args*, ***kwargs*) → bool
Validates any. All `*args` and `**kwargs` must match on ore more of *rules_any*

Returns `True` if all `*args` and `**kwargs` are valid; Otherwise, `False`

Return type bool

Raises **Exception** – If *raise_error* is `True` and validation Fails. The type of exception raised is dependend on the *IRule* that caused validation failure. Most rules raise a `ValueError` or a `TypeError`.

property raise_error: bool
Determines if errors will be raised during validation

If `True` then errors will be raised when validation fails. Default value is `True`.

Getter Gets if errors can be raised.

Setter Sets if errors can be raised.

property rules_all: Iterable[*kwhelp.rules.IRule*]
Gets rules passed into `rules_all` of constructor used for validation of args.

property rules_any: Iterable[*kwhelp.rules.IRule*]
Gets rules passed into `rules_any` of constructor used for validation of args.

SubClassChecker Class

class kwhelp.checks.**SubClassChecker**(**args: type*, ***kwargs*)
Class that validates args is a subclass of a give type

__init__(**args: type*, ***kwargs*)
Constructor

Other Arguments: args (type): One or more types used for Validation purposes.

Keyword Arguments

- **raise_error** (*bool, optional*) – If `True` then an error will be raised if a *validate()* fails: Othwewise *validate()* will return a boolean value indicationg success or failure. Default `True`
- **opt_inst_only** (*bool, optional*) – If `True` then validation will requires all values being tested to be an instance of a class. If `False` valadition will test class instance and class type. Default `True`

validate(**args*, ***kwargs*) → bool
Validates all `*args` and all `**kwargs` against `types` that are passed into constructor.

Returns `True` if all `*args` and all `**kwarg` match a valid class; Otherwise; `False`.

Return type bool

Raises **TypeError** – if `raise_error` is `True` and validation fails.

property instance_only: bool
Determines if validation requires instance of class

If `True` then validation will fail when a type is validated, rather then an instance of a class.

Getter Gets instance_only.

Setter Sets instance_only.

property raise_error: bool
Determines if errors will be raised during validation

If `True` then errors will be raised when validation fails.

Getter Gets if errors can be raised.

Setter Sets if errors can be raised.

property types: Tuple[type]
Gets the types passed into constructor that are used for validating args

TypeChecker Class

class kwhelp.checks.**TypeChecker**(**args: type*, ***kwargs*)
Class that validates args match a given type

__init__(**args: type*, ***kwargs*)
Constructor

Other Arguments: args (type): One or more types used for Validation purposes.

Keyword Arguments

- **raise_error** (*bool, optional*) – If `True` then an error will be raised if a *validate()* fails: Othwewise *validate()* will return a boolean value indicationg success or failure. Default `True`
- **type_instance_check** (*bool, optional*) – If `True` then *validate()* args are tested also for isinstance if type does not match, rather then just type check if type is a match. If `False` then values willl only be tested as type. Default `True`

validate(**args*, ***kwargs*) → bool
Validates all `*args` and all `**kwargs` against `types` that are passed into constructor.

Returns `True` if all `*args` and all `**kwarg` match a type; Otherwise; `False`.

Return type bool

Raises **TypeError** – if `raise_error` is `True` and validation fails.

property raise_error: bool
Determines if errors will be raised during validation

If `True` then errors will be raised when validation fails.

Getter Gets if errors can be raised.

Setter Sets if errors can be raised.

property type_instance_check: bool
Determines if instance checking is done with type checking.

If `True` then `validate`() args are tested also for isinstance if type does not match, rather then just type check if type is a match. If `False` then values willl only be tested as type.

Getter Gets type_instance_check value

Setter Sets type_instance_check value

property types: Tuple[type]
Gets the types passed into constructor that are used for validating args

2.8.11 decorator

AcceptedTypes Class

class kwhelp.decorator.**AcceptedTypes**(**args: Union[type, Iterable[type]]*, ***kwargs*)

Decorator that decorates methods that requires args to match types specificed in a list

See also:

AcceptedTypes Usage

__init__(**args: Union[type, Iterable[type]]*, ***kwargs*)

Constructor

Parameters **args** (*Union[type, Iterable[type]]*) – One or more types or Iterator[type] for validation.

Keyword Arguments

- **type_instance_check** (*bool, optional*) – If `True` then args are tested also for `isinstance()` if type does not match, rather then just type check. If `False` then values willl only be tested as type. Default `True`
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_all_args** (*bool, optional*) – If `True` then the last subclass type passed into constructor will define any remaining args. This allows for one subclass to define required match of all arguments that decorator is applied to. Default `False`
- **opt_args_filter** (DecArgEnum, *optional*) – Filters the arguments that are validated. Default `DecArgEnum.ALL`.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

ArgsLen Class

class kwhelp.decorator.**ArgsLen**(**args: Union[type, Iterable[type]]*, ***kwargs*)

Decorartor that sets the number of args that can be added to a function

Raises

- **ValueError** – If wrong args are passed into construcor.
- **ValueError** – If validation of arg count fails.

See also:

ArgsLen Usage

__init__(**args: Union[type, Iterable[type]]*, ***kwargs*)

Constructor

Parameters **args** (*Union[int, iterable[int]]*) – One or more int or Iterator[int] for validation.

- Single `int` values are to match exact.

- `iterable[int]` must be a pair of `int` with the first `int` less then the second `int`.

Keyword Arguments

- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

ArgsMinMax Class

class kwhelp.decorator.**ArgsMinMax**(*min: Optional[int] = 0*, *max: Optional[int] = None*, ***kwargs*)

Decorartor that sets the min and or max number of args that can be added to a function

See also:

ArgsMinMax Usage

__init__(*min: Optional[int] = 0*, *max: Optional[int] = None*, ***kwargs*)

Constructor

Parameters

- **min** (*int, optional*) – Min number of args for a function. Defaults to 0.
- **max** (*int, optional*) – Max number of args for a function. Defaults to None.

Keyword Arguments

- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

AutoFill Class

class kwhelp.decorator.**AutoFill**(**args*, ***kwargs*)

Class decorator that replaces the `__init__` function with one that sets instance attributes with the specified argument names and default values. The original `__init__` is called with no arguments after the instance attributes have been assigned.

Example

```
>>> @AutoFill('a', 'b', c=3)
... class Foo: pass
>>> sorted(Foo(1, 2).__dict__.items())
[('a', 1), ('b', 2), ('c', 3)]
```

AutoFillKw Class

class kwhelp.decorator.**AutoFillKw**(*cls*)

Class decorator that replaces the `__init__` function with one that sets instance attributes with the specified key, value of `kwargs`. The original `__init__` is called with any `*args` after the instance attributes have been assigned.

Example

```
>>> @AutoFillKw
... class Foo: pass
>>> sorted(Foo(a=1, b=2, End="!").__dict__.items())
[('End', '!'), ('a', 1), ('b', 2)]
```

DecArgEnum

class kwhelp.decorator.**DecArgEnum**(*value*)

Represents options for the type of function arguments to process

ARGS = 1

Process `*args`

All_ARGS = 7

Process All Args

KWARGS = 2

Process `**kwargs`

NAMED_ARGS = 4

Process named keyword args

NO_ARGS = 6

Process Named Keyword args and `**kwargs` only

DecFuncEnum

class kwhelp.decorator.**DecFuncEnum**(*value*)

Represents options for type of Function or Method

FUNCTION = 1

Normal Unbound function

METHOD = 3

Class Method

METHOD_CLASS = 4

Class Method (@classmethod)

METHOD_STATIC = 2
Class Static Method (@staticmethod)

PROPERTY_CLASS = 5
Class Property (@property)

DefaultArgs Class

class kwhelp.decorator.**DefaultArgs**(***kwargs: Dict[str, object]*)
Decorator that defines default values for **kwargs of a function.

See also:

DefaultArgs Usage

__init__(***kwargs: Dict[str, object]*)
Constructor

Keyword Arguments **kwargs** (*Dict[str, object]*) – One or more Key, Value pairs to assign to wrapped function args as defaults.

RequireArgs Class

class kwhelp.decorator.**RequireArgs**(**args: str*, ***kwargs*)
Decorator that defines required args for **kwargs of a function.

See also:

RequireArgs Usage

__init__(**args: str*, ***kwargs*)
Constructor

Parameters **args** (*type*) – One or more names of wrapped function args to require.

Keyword Arguments

- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default DecFuncType.FUNCTION
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If opt_return is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

ReturnRuleAll Class

class kwhelp.decorator.**ReturnRuleAll**(**args:* kwhelp.rules.IRule, ***kwargs*)
Decorator that decorates methods that require return value to match all rules specificed.

See also:

ReturnRuleAll usage

__init__(**args:* kwhelp.rules.IRule, ***kwargs*)
Constructor

Parameters **args** (IRule) – One or more rules to use for validation

Keyword Arguments

- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default DecFuncType.FUNCTION Default True
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If opt_return is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

ReturnRuleAny Class

class kwhelp.decorator.**ReturnRuleAny**(**args:* kwhelp.rules.IRule, ***kwargs*)

Decorator that decorates methods that require return value to match any of the rules specificed.

See also:

ReturnRuleAny Usage

__init__(**args:* kwhelp.rules.IRule, ***kwargs*)

Constructor

Parameters **args** (IRule) – One or more rules to use for validation

Keyword Arguments

- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default DecFuncType.FUNCTION
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If opt_return is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

ReturnType Class

class kwhelp.decorator.**ReturnType**(**args: type*, ***kwargs*)

Decorator that decorates methods that require return value to match a type specificed.

See also:

ReturnType Usage

__init__(**args: type*, ***kwargs*)

Constructor

Parameters **args** (*type*) – One ore more types that is used to validate return type.

Keyword Arguments

- **type_instance_check** (*bool, optional*) – If True then args are tested also for isinstance() if type does not match, rather then just type check. If False then values willl only be tested as type. Default True
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If opt_return is supplied then it will be return when validation fails and no error will be raised.

- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

RuleCheckAll Class

class kwhelp.decorator.**RuleCheckAll**(**args:* kwhelp.rules.IRule, ***kwargs*)

Decorator that decorates methods that require args to match all rules specificed in `rules` list.

If a function arg does not match all rules in `rules` list then validation will fail.

See also:

RuleCheckAll Usage

__init__(**args:* kwhelp.rules.IRule, ***kwargs*)

Constructor

Parameters **args** (IRule) – One or more rules to use for validation

Keyword Arguments

- **raise_error** (*bool, optional*) – If `True` then an Exception will be raised if a validation fails. The kind of exception raised depends on the rule that is invalid. Typically a `TypeError` or a `ValueError` is raised.

  If `False` then an attribute will be set on decorated function named `is_rules_all_valid` indicating if validation status. Default `True`.

- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`

- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.

- **opt_args_filter** (DecArgEnum, *optional*) – Filters the arguments that are validated. Default `DecArgEnum.ALL`.

- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

RuleCheckAllKw Class

class kwhelp.decorator.**RuleCheckAllKw**(*arg_info: Dict[str, Union[int,* kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]], rules: Optional[Iterable[Union[*kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]]] = None, **kwargs*)

Decorator that decorates methods that require specific args to match rules specificed in `rules` list.

If a function specific args do not match all matching rules in `rules` list then validation will fail.

See also:

RuleCheckAllKw Usage

__init__(*arg_info: Dict[str, Union[int,* kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]], rules: Optional[Iterable[Union[*kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]]] = None, **kwargs*)

Constructor

Parameters

- **arg_info** (*Dict[str, Union[int, IRule, Iterable[IRule]]]*) – Dictionary of Key and int, IRule, or Iterable[IRule]. Each Key represents that name of an arg to check with one or more rules. If value is int then value is an index that corresponds to an item in `rules`.
- **rules** (*Iterable[Union[IRule, Iterable[IRule]]], optional*) – List of rules for arg_info entries to match. Default `None`

Keyword Arguments

- **raise_error** (*bool, optional*) – If `True` then an Exception will be raised if a validation fails. The kind of exception raised depends on the rule that is invalid. Typically a `TypeError` or a `ValueError` is raised.

  If `False` then an attribute will be set on decorated function named `is_rules_kw_all_valid` indicating if validation status. Default `True`.
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

RuleCheckAny Class

class kwhelp.decorator.**RuleCheckAny**(**args:* kwhelp.rules.IRule, ***kwargs*)

Decorator that decorates methods that require args to match a rule specificed in `rules` list.

If a function arg does not match at least one rule in `rules` list then validation will fail.

See also:

RuleCheckAny Usage

__init__(**args:* kwhelp.rules.IRule, ***kwargs*)

Constructor

Parameters **args** (IRule) – One or more rules to use for validation

Keyword Arguments

- **raise_error** (*bool, optional*) – If `True` then an Exception will be raised if a validation fails. The kind of exception raised depends on the rule that is invalid. Typically a `TypeError` or a `ValueError` is raised.

  If `False` then an attribute will be set on decorated function named `is_rules_any_valid` indicating if validation status. Default `True`.
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_args_filter** (DecArgEnum, *optional*) – Filters the arguments that are validated. Default `DecArgEnum.ALL`.

- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

RuleCheckAnyKw Class

class kwhelp.decorator.**RuleCheckAnyKw**(*arg_info: Dict[str, Union[int,* kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]], rules: Optional[Iterable[Union[*kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]]] = None*, ***kwargs*)

Bases: *kwhelp.decorator.RuleCheckAllKw*

Decorator that decorates methods that require specific args to match rules specificed in `rules` list.

If a function specific args do not match at least one matching rule in `rules` list then validation will fail.

See also:

RuleCheckAnyKw Usage

__init__(*arg_info: Dict[str, Union[int,* kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]], rules: Optional[Iterable[Union[*kwhelp.rules.IRule*, Iterable[*kwhelp.rules.IRule*]]]] = None*, ***kwargs*)

Constructor

Parameters

- **arg_info** (*Dict[str, Union[int,* IRule*, Iterable[*IRule*]]]*) – Dictionary of Key and int, IRule, or Iterable[IRule]. Each Key represents that name of an arg to check with one or more rules. If value is int then value is an index that corresponds to an item in `rules`.
- **rules** (*Iterable[Union[*IRule*, Iterable[*IRule*]]], optional*) – List of rules for arg_info entries to match. Default `None`

Keyword Arguments

- **raise_error** (*bool, optional*) – If `True` then an Exception will be raised if a validation fails. The kind of exception raised depends on the rule that is invalid. Typically a `TypeError` or a `ValueError` is raised.

  If `False` then an attribute will be set on decorated function named `is_rules_kw_all_valid` indicating if validation status. Default `True`.
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

property args: Iterable[object]

Gets/sets wrapped function args

property fn_cache: Dict[str, object]

Gets function level cache

property kwargs: Dict[str, Any]

Gets/sets wrapped function kwargs

SubClass Class

class kwhelp.decorator.**SubClass**(**args: Union[type, Iterable[type]]*, ***kwargs*)

Decorator that requires args of a function to match or be a subclass of types specificed in constructor.

See also:

SubClass Usage

__init__(**args: Union[type, Iterable[type]]*, ***kwargs*)

Constructor

Parameters **args** (*Union[type, Iterable[type]]*) – One or more types or Iterator[type] for validation.

Keyword Arguments

- **type_instance_check** (*bool, optional*) – If `True` then args are tested also for `isinstance()` if type does not match, rather then just type check. If `False` then values willl only be tested as type. Default `True`
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_inst_only** (*bool, optional*) – If `True` then validation will requires all values being tested to be an instance of a class. If `False` valadition will test class instance and class type. Default `True`
- **opt_all_args** (*bool, optional*) – If `True` then the last subclass type passed into constructor will define any remaining args. This allows for one subclass to define required match of all arguments that decorator is applied to. Default `False`
- **opt_args_filter** (`DecArgEnum`, *optional*) – Filters the arguments that are validated. Default `DecArgEnum.ALL`.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

SubClasskKw Class

class kwhelp.decorator.**SubClasskKw**(*arg_info: Dict[str, Union[int, type, Iterable[type]]]*, *types: Optional[Iterable[Union[type, Iterable[type]]]] = None*, ***kwargs*)

Decorator that requires args of a function to match or be a subclass of types specificed in constructor.

See also:

SubClasskKw Usage

__init__(*arg_info: Dict[str, Union[int, type, Iterable[type]]]*, *types: Optional[Iterable[Union[type, Iterable[type]]]] = None*, ***kwargs*)

Constructor

Parameters

- **arg_info** (*Dict[str, Union[int, type, Iterable[type]]]*) – Dictionary of Key and int, type, or Iterable[type]. Each Key represents that name of an arg to match one or more types(s). If value is int then value is an index that corresponds to an item in `types`.

- **types** (*Iterable[Union[type, Iterable[type]]], optional*) – List of types for arg_info entries to match. Default `None`

Keyword Arguments

- **type_instance_check** (*bool, optional*) – If `True` then args are tested also for `isinstance()` if type does not match, rather then just type check. If `False` then values willl only be tested as type. Default `True`
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

TypeCheck Class

class kwhelp.decorator.**TypeCheck**(**args: Union[type, Iterable[type]], **kwargs*)

Decorator that decorates methods that requires args to match a type specificed in a list

See also:

TypeCheck Usage

__init__(**args: Union[type, Iterable[type]], **kwargs*)

Constructor

Parameters **args** (*type*) – One or more types for wrapped function args to match.

Keyword Arguments

- **raise_error** – (bool, optional): If `True` then a `TypeError` will be raised if a validation fails. If `False` then an attribute will be set on decorated function named `is_types_valid` indicating if validation status. Default `True`.
- **type_instance_check** (*bool, optional*) – If `True` then args are tested also for `isinstance()` if type does not match, rather then just type check. If `False` then values willl only be tested as type. Default `True`
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_args_filter** (DecArgEnum, *optional*) – Filters the arguments that are validated. Default `DecArgEnum.ALL`.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

Raises

- **TypeError** – If `types` arg is not a iterable object such as a list or tuple.
- **TypeError** – If any arg is not of a type listed in `types`.

TypeCheckKw Class

class kwhelp.decorator.**TypeCheckKw**(*arg_info: Dict[str, Union[int, type, Iterable[type]]]*, *types: Optional[Iterable[Union[type, Iterable[type]]]] = None*, ***kwargs*)

Decorator that decorates methods that require key, value args to match a type specificed in a list

See also:

TypeCheckKw Usage

__init__(*arg_info: Dict[str, Union[int, type, Iterable[type]]]*, *types: Optional[Iterable[Union[type, Iterable[type]]]] = None*, ***kwargs*)

Constructor

Parameters

- **arg_info** (*Dict[str, Union[int, type, Iterable[type]]]*) – Dictionary of Key and int, type, or Iterable[type]. Each Key represents that name of an arg to match one or more types(s). If value is int then value is an index that corresponds to an item in `types`.
- **types** (*Iterable[Union[type, Iterable[type]]], optional*) – List of types for arg_info entries to match. Default `None`

Keyword Arguments

- **raise_error** – (bool, optional): If `True` then a `TypeError` will be raised if a validation fails. If `False` then an attribute will be set on decorated function named `is_types_kw_valid` indicating if validation status. Default `True`.
- **type_instance_check** (*bool, optional*) – If `True` then args are tested also for `isinstance()` if type does not match, rather then just type check. If `False` then values willl only be tested as type. Default `True`
- **ftype** (*DecFuncType, optional*) – Type of function that decorator is applied on. Default `DecFuncType.FUNCTION`
- **opt_return** (*object, optional*) – Return value when decorator is invalid. By default an error is rasied when validation fails. If `opt_return` is supplied then it will be return when validation fails and no error will be raised.
- **opt_logger** (*Union[Logger, LoggerAdapter], optional*) – Logger that logs exceptions when validation fails.

callcounter Decorator

Decorator method that adds `call_count` attribute to decorated method. `call_count` is `0` if method has not been called. `call_count` increases by 1 each time method is been called.

Note: This decorator needs to be the topmost decorator applied to a method

Example

```
>>> @callcounter
>>> def foo(msg):
>>>     print(msg)

>>> print("Call Count:", foo.call_count)
0
>>> foo("Hello")
Hello
>>> print("Call Count:", foo.call_count)
1
>>> foo("World")
World
>>> print("Call Count:", foo.call_count)
2
```

See also:

Callcounter Usage

calltracker Decorator

Decorator method that adds `has_been_called` attribute to decorated method. `has_been_called` is `False` if method has not been called. `has_been_called` is `True` if method has been called.

Note: This decorator needs to be the topmost decorator applied to a method

---

Example

```
>>> @calltracker
>>> def foo(msg):
>>>     print(msg)

>>> print(foo.has_been_called)
False
>>> foo("Hello World")
Hello World
>>> print(foo.has_been_called)
True
```

See also:

Calltracker Usage

singleton Decorator

Decorator that makes a class a singleton class

Example

```
@singleton
class Logger:
    def log(self, msg):
        print(msg)

logger1 = Logger()
logger2 = Logger()
assert logger1 is logger
```

See also:

Singleton Usage

2.8.12 rules

IRule Class

class kwhelp.rules.**IRule**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Abstract Interface Class for rules

See also:

Rules

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

abstract validate() → bool

Gets attrib field and value are valid

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleAttrExist Class

class kwhelp.rules.**RuleAttrExist**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule to ensure an attribute does exist before its value is set.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that `field_name` is an existing attribute of `originator` instance.

Raises **AttributeError** – If `raise_errors` is `True` and `field_name` is not an attribue of `originator` instance.

Returns `True` if `field_name` is an existing attribue of `originator` instance; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleAttrNotExist Class

class kwhelp.rules.**RuleAttrNotExist**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.IRule*

Rule to ensure an attribute does not exist before it is added to class.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that `field_name` is not an existing attribute of `originator` instance.

Raises **AttributeError** – If `raise_errors` is `True` and `field_name` is already an attribue of `originator` instance.

Returns `True` if `field_name` is not an existing attribue of `originator` instance; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleBool Class

class kwhelp.rules.**RuleBool**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is instance of bool.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to field_name
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is a bool

Raises **TypeError** – If raise_errors is True and field_value is not instance of bool.

Returns True if field_value is a bool; Otherwise, False.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to field_name

Getter Gets value assigned to field_name

Setter Sets value assigned to field_name

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleByteSigned Class

class kwhelp.rules.**RuleByteSigned**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleInt*

Signed Byte rule, range from `-128` to `127`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Valids

Raises **ValueError** – If `raise_errors` is `False` and value is less then `-128` or greater than `128`.

Returns `True` if Validation passes; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleByteUnsigned Class

class kwhelp.rules.**RuleByteUnsigned**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleInt*

Unsigned Byte rule, range from `0` to `255`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Valids

Raises **ValueError** – If `raise_errors` is `False` and value is less then `0` or greater than `255`.

Returns `True` if Validation passes; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleFloat Class

class kwhelp.rules.**RuleFloat**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is to type `float`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is a float

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not a float.

Returns `True` if `field_value` is a positive float; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleFloatNegative Class

class kwhelp.rules.**RuleFloatNegative**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleFloat*

Rule that matched only if value is less than `0.0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a negative float

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not a negative float.

Returns `True` if `field_value` is a negative float; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleFloatNegativeOrZero Class

class kwhelp.rules.**RuleFloatNegativeOrZero**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleFloat*

Rule that matched only if value is equal or less than `0.0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is equal to `0.0` or a negative float

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not a negative float.

Returns `True` if `field_value` is equal to `0.0` or a negative float; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleFloatPositive Class

class kwhelp.rules.**RuleFloatPositive**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleFloat*

Rule that matched only if value is equal or greater than `0.0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a positive float

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not a positive float.

Returns `True` if `field_value` is a positive float; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleFloatZero Class

class kwhelp.rules.**RuleFloatZero**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.RuleFloat*

Rule that matched only if value is equal to `0.0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign equals `0.0` float

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not equal to `0.0` float.

Returns `True` if `field_value` equals `0.0` float; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleInt Class

class kwhelp.rules.**RuleInt**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is instance of `int`.

Note: If value is a of type `bool` then validation will fail for this rule.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is an int

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not an int.

Returns `True` if `field_value` is an `int`; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleIntNegative Class

class kwhelp.rules.**RuleIntNegative**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleInt*

Rule that matched only if value is less than `0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a negative int

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not a negative int.

Returns `True` if `field_value` is a negative int; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleIntNegativeOrZero Class

class kwhelp.rules.**RuleIntNegativeOrZero**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleInt*

Rule that matched only if value is equal or less than `0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is equal to zero or a negative int

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not a negative int.

Returns `True` if `field_value` is equal to zero or a negative int; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleIntPositive Class

class kwhelp.rules.**RuleIntPositive**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleInt*

Rule that matched only if value is equal or greater than `0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a posivite int

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not a positive int.

Returns `True` if `field_value` is a positive int; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleIntZero Class

class kwhelp.rules.**RuleIntZero**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.RuleInt*

Rule that matched only if value is equal to `0`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is equal to `0` int.

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not equal to `0` int.

Returns `True` if `field_value` equals `0` int; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleIterable Class

class kwhelp.rules.**RuleIterable**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is iterable such as list, tuple, set.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to field_name
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is iterable

Raises **TypeError** – If raise_errors is True and field_value is not iterable.

Returns True if field_value is a iterable; Otherwise, False.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to field_name

Getter Gets value assigned to field_name

Setter Sets value assigned to field_name

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleNone Class

class kwhelp.rules.**RuleNone**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is `None`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign to attribute is `None`.

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not `None`.

Returns `True` if `field_value` is `None`; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleNotIterable Class

class kwhelp.rules.**RuleNotIterable**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.IRule*

Rule that matched only if value is not iterable.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is not iterable

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is iterable.

Returns `True` if `field_value` is a not iterable; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleNotNone Class

class kwhelp.rules.**RuleNotNone**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is not None.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to field_name
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign to attribute is not None.

Raises **ValueError** – If raise_errors is True and field_value is None.

Returns True if field_value is not None; Otherwise, False.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to field_name

Getter Gets value assigned to field_name

Setter Sets value assigned to field_name

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleNumber Class

class kwhelp.rules.**RuleNumber**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is a valid number.

Note: If value is a of type `bool` then validation will fail for this rule.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is a number

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not a number.

Returns `True` if `field_value` is a number; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RulePath Class

class kwhelp.rules.**RulePath**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is instance of `Path`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is a Path

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not instance of Path.

Returns `True` if `field_value` is a bool; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RulePathExist Class

class kwhelp.rules.**RulePathExist**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.RulePath*

Rule that matched only if value is instance of `Path` and path exist.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is a path that exist

Raises **FileNotFoundError** – If `raise_errors` is `True` and `field_value` is path does not exist.

Returns `True` if `field_value` is an existing path; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RulePathNotExist Class

class kwhelp.rules.**RulePathNotExist**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RulePath*

Rule that matched only if value is instance of `Path` and path does not exist.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a Path that does not exist

Raises **FileExistsError** – If `raise_errors` is `True` and `field_value` is path that is existing.

Returns `True` if `field_value` is a p ath that does not exist; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleStr Class

class kwhelp.rules.**RuleStr**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.IRule*

Rule that matched only if value is of type `str`.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is a string

Raises **TypeError** – If `raise_errors` is `True` and `field_value` is not instance of string.

Returns `True` if `field_value` is a string; Otherwise, `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleStrEmpty Class

class kwhelp.rules.**RuleStrEmpty**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Bases: *kwhelp.rules.RuleStr*

Rule that matched only if value is equal to empty string.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)
Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool
Validates that value to assign is a string and is an empty string.

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not an empty string.

Returns `True` if value is an empty string; Otherwise; `False`.

Return type bool

property field_name: str
Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object
The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str
Gets the key currently being read

property originator: object
Gets object that attributes validated for

property raise_errors: bool
Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleStrNotNullEmptyWs Class

class kwhelp.rules.**RuleStrNotNullEmptyWs**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleStrNotNullOrEmpty*

Rule that matched only if value is not `None`, empty or whitespace.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a string and is not a empty or whitespace string.

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not instance of string or is empty or whitespace string

Returns `True` if value is valid; Otherwise; `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleStrNotNullOrEmpty Class

class kwhelp.rules.**RuleStrNotNullOrEmpty**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleStr*

Rule that matched only if value is not `None` or empty string.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a string and is not a empty string.

Raises **ValueError** – If `raise_errors` is `True` and `field_value` is not instance of string or is empty string

Returns `True` if value is valid; Otherwise; `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleStrPathExist Class

class kwhelp.rules.**RuleStrPathExist**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleStr*

Rule that matched only if value is instance of str and path exist.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a str path that exist

Raises **FileNotFoundError** – If `raise_errors` is `True` and `field_value` is path does not exist.

Returns `True` if `field_value` is a path that does exist; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

RuleStrPathNotExist Class

class kwhelp.rules.**RuleStrPathNotExist**(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Bases: *kwhelp.rules.RuleStr*

Rule that matched only if value is instance of str and path is not existing.

__init__(*key: str*, *name: str*, *value: object*, *raise_errors: bool*, *originator: object*)

Constructor

Parameters

- **key** (*str*) – the key that rule is to apply to.
- **name** (*str*) – the name of the field that value was assigned
- **value** (*object*) – the value that is assigned to `field_name`
- **raise_errors** (*bool*) – determinins if rule could raise an error when validation fails
- **originator** (*object*) – the object that attributes validated for

Raises **TypeError** – If any arg is not of the correct type

validate() → bool

Validates that value to assign is a path not existing

Raises **FileExistsError** – If `raise_errors` is `True` and `field_value` is a path that is not existing.

Returns `True` if `field_value` is a path that does not exist; Otherwise, `False`.

Return type bool

property field_name: str

Name of the field assigned.

Getter Gets the name of the field assigned

Setter Sets the name of the field assigned

property field_value: object

The value assigned to `field_name`

Getter Gets value assigned to `field_name`

Setter Sets value assigned to `field_name`

property key: str

Gets the key currently being read

property originator: object

Gets object that attributes validated for

property raise_errors: bool

Determines if a rule can raise an error when validation fails.

Getter Gets if a rule could raise an error when validation fails

Setter Sets if a rule could raise an error when validation fails

2.8.13 exceptions

CancelEventError Class

class kwhelp.exceptions.**CancelEventError**
Cancel Event Error

ReservedAttributeError Class

class kwhelp.exceptions.**ReservedAttributeError**
Error when a reserved attribute is attempted to be set

RuleError Class

class kwhelp.exceptions.**RuleError**(***kwargs*)
Rule Error

__init__(***kwargs*)
Constructor

Keyword Arguments

- **err_rule** (*Type[IRule], optional*) – Rule that caused exception.
- **rules_all** (*Iterable[Type[IRule]], optional*) – List of rules that were to all be matched. One of these rules is usually the reason this exception is being raised.
- **rules_any** (*Iterable[Type[IRule]], optional*) – List of rules that required one or more matches. One of these rules is usually the reason this exception is being raised.
- **arg_name** (*str, optional*) – Name of the argument for this exception.
- **errors** (*Union[Exception, Iterable[Exception]], optional*) – Exception or Exceptions that cause this error.
- **fn_name** (*str, optional*) – Name of function/property that raise error.
- **msg** (*str, optional*) – Optional message to append.

static from_rule_error(*rule_error:* kwhelp.exceptions.RuleError, ***kwargs*) → *kwhelp.exceptions.RuleError*
Creates a new RuleError from an existing RuleError

Parameters **rule_error** (RuleError) – Current instance of RuleError use to base return value on.

Keyword Arguments **kwargs** – One or more Key, Value properties that will replace property of rule_error.

Returns New RuleError instance with updated properties included in **kwargs.

Return type *RuleError*

property arg_name: Optional[str]
Gets Name of the argument for this exception

property err_rule: Optional[Type[*kwhelp.rules.IRule*]]
Gets rule that caused exception.

property errors: Optional[Union[Exception, Iterable[Exception]]]
Gets Exception or Exceptions that cause this error

property fn_name: Union[None, str]
Gets the function/property name that raised the error

property msg: Optional[str]
Gets any messsage that is appended

property rules_all: List[Type[*kwhelp.rules.IRule*]]
Gets list of rules that were to all be matched.

property rules_any: List[Type[*kwhelp.rules.IRule*]]
Gets of rules that required one or more matches.

CHAPTER

THREE

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

k

INDEX

Symbols

G

H

I

K

M

N

O

P

R

S

T

U

V