pycheckstype Documentation

Release 1.0

hubo

Jan 22, 2019

Contents:

CHAPTER 1

Reference

exception pychecktype.**CheckFailedException**
Raise this exception in a customized checker/converter to replace the default message

__weakref__
list of weak references to the object (if defined)

class pychecktype.**CustomizedChecker**(**args*, ***kwargs*)
Inherit from this class to create a customized type checker

__init__(**args*, ***kwargs*)
Call bind()

__weakref__
list of weak references to the object (if defined)

bind()
Allow delayed init

final_check_type(*value*, *current_result*, *recursive_check_type*)
Second-step check for value. This step can use recursive check.

Parameters

- **value** – value to be checked
- **current_result** – value returned by pre_check_type. If this value is not None, the return value must be the same object.
- **recursive_check_type** – a function recursive_check_type(value, type, path=None) to be called to do recursive type check. When the call fails, path is automatically joined to create a property path

pre_check_type(*value*)
First-step check for value. This step should not do any recursive check.

Parameters **value** – value to be checked

Returns

An object if recursive creation if needed, or None if not needed.

TypeMismatchException should be raised if there is something wrong.

class pychecktype.**DictChecker**(**args*, ***kwargs*)

Default *{}* type implementation

Examples:

```
>>> dict_({})
{}
>>> dict_([])
Traceback (most recent call last):
  ...
InvalidTypeException: [] is not a valid type: must be a dict
```

__repr__()

Return repr(self).

bind(*type_*, *allowed_type=<class 'dict'>*, *created_type=<class 'dict'>*)

Parameters

- **type** – a dict describing the input format
- **allowed_type** – limit input type to a sub type, or a tuple of sub types
- **created_type** – create a subtype of dict instead (e.g. OrderedDict)

final_check_type(*value*, *current_result*, *recursive_check_type*)

Second-step check for value. This step can use recursive check.

Parameters

- **value** – value to be checked
- **current_result** – value returned by pre_check_type. If this value is not None, the return value must be the same object.
- **recursive_check_type** – a function recursive_check_type(value, type, path=None) to be called to do recursive type check. When the call fails, path is automatically joined to create a property path

pre_check_type(*value*)

First-step check for value. This step should not do any recursive check.

Parameters **value** – value to be checked

Returns

An object if recursive creation if needed, or None if not needed.

TypeMismatchException should be raised if there is something wrong.

class pychecktype.**ExtraChecker**(**args*, ***kwargs*)

Do extra checks around a basic type

Examples:

```
>>> extra([])
extra([])
>>> check_type({"age": 15}, extra({"age": int}))
{'age': 15}
>>> check_type({"age": 15}, extra({"age": int},
```

(continues on next page)

(continued from previous page)

```
... check = lambda x: 14 < x['age'] < 18))
{'age': 15}
>>> check_type({"age": 19}, extra({"age": int},
... check = lambda x: 14 < x['age'] < 18))
Traceback (most recent call last):
    ...
TypeMismatchException: {'age': 19} cannot match type extra({'age': <... 'int'>}):␣
↪check returns False
>>> check_type({"age": 19}, extra({"age": int},
... check = (lambda x: 14 < x['age'] < 18, 'invalid age')))
Traceback (most recent call last):
    ...
TypeMismatchException: {'age': 19} cannot match type extra({'age': <... 'int'>}):␣
↪invalid age
>>> e_t = extra(None, precreate = lambda x: {})
Traceback (most recent call last):
    ...
InvalidTypeException: extra(None) is not a valid type: precreate and merge must␣
↪be used together
>>> e_t = extra()
>>> e_t.bind(tuple_((str, [e_t])),
...          check_before = (lambda x: len(x) >= 2, 'must have 2 items'),
...          check = lambda x: len(x[1]) <= 3,
...          convert_before = lambda x: x[:2],
...          convert = lambda x: (x[0], x[1], len(x[1])),
...          precreate = lambda x: {},
...          merge = lambda c, r:
...                      c.update((
...                          ("name", r[0]),
...                          ("children", r[1]),
...                          ("childcount", r[2])
...                      ))
...          )
>>> check_type(("a",), e_t)
Traceback (most recent call last):
    ...
TypeMismatchException: ('a',) cannot match type extra(tuple_((<... 'str'>, [...
↪]))): must have 2 items
>>> e_t = extra()
>>> e_t.bind(tuple_((str, [e_t])),
...          check_before = lambda x: len(x) >= 2,
...          check = lambda x: len(x[1]) <= 3,
...          convert_before = lambda x: x[:2],
...          convert = lambda x: (x[0], x[1], len(x[1])),
...          precreate = lambda x: {},
...          merge = lambda c, r:
...                      c.update((
...                          ("name", r[0]),
...                          ("children", r[1]),
...                          ("childcount", r[2])
...                      ))
...          )
>>> check_type(("a",), e_t)
Traceback (most recent call last):
    ...
TypeMismatchException: ('a',) cannot match type extra(tuple_((<... 'str'>, [...
↪]))): check_before returns False
```

(continues on next page)

(continued from previous page)

```
>>> check_type(("a",[],123), e_t) == \
... {'name': 'a', 'children': [], 'childcount': 0}
True
>>> d = ("a",[])
>>> d[1].append(d)
>>> d[1].append(d)
>>> r = check_type(d, e_t)
>>> r['name']
'a'
>>> r['childcount']
2
>>> len(r['children'])
2
>>> r['children'][0] is r
True
>>> r['children'][1] is r
True
```

__repr__()
Return repr(self).

bind(*basictype=<class 'object'>*, *check=None*, *check_before=None*, *convert=None*, *convert_before=None*, *precreate=None*, *merge=None*)
Do extra check/convert around a basic type check. Added steps are:

1. if check_before is not None, call check_before(value), raises Exception if it returns False
2. if precreate is not None, create result_obj = precreate(value)
3. if convert_before is not None, value = convert_before(value)
4. do the basic type check against basictype, get result
5. if check is not None, call check(result), raises Exception if it returns False
6. if convert is not None, result = convert(result)
7. if merge is not None, call merge(result_obj, result), then result = result_obj

Use precreate and merge to create a recursive object (e.g. an instance referencing itself): first create an empty object, do check type, and merge the result to the pre-created object.

Use bind() to delay the initialize of this type to create recursive types:

```
new_type = extra()
new_type.bind([new_type])
```

Take care of convert_before / convert: do not break the recursive structure.

check/check_before can be a callable, or a tuple(callable, message) to customize the error message when check fails. You can also raises an *CheckFailedException* instance to customize the error message, in this case the error message will be *str(check_failed_exception)*.

final_check_type(*value*, *current_result*, *recursive_check_type*)
Second-step check for value. This step can use recursive check.

Parameters

- **value** – value to be checked
- **current_result** – value returned by pre_check_type. If this value is not None, the return value must be the same object.

- **recursive_check_type** – a function recursive_check_type(value, type, path=None) to be called to do recursive type check. When the call fails, path is automatically joined to create a property path

pre_check_type(*value*)
First-step check for value. This step should not do any recursive check.

Parameters **value** – value to be checked

Returns

An object if recursive creation if needed, or None if not needed.

TypeMismatchException should be raised if there is something wrong.

exception pychecktype.**InvalidTypeException**(*type_*, *info=None*)

__init__(*type_*, *info=None*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

class pychecktype.**ListChecker**(**args*, ***kwargs*)
Default *[]* type implementation

Examples:

```
>>> list_([])
[]
>>> list_({})
Traceback (most recent call last):
  ...
InvalidTypeException: {} is not a valid type: must be a list
```

__repr__()
Return repr(self).

bind(*type_*, *strict=False*, *allowed_type=(<class 'list'>, <class 'tuple'>)*)
*type_* must be a list type [] / [sub_type]

Parameters

- **strict** – if True, auto-convert from a single value to a list is disabled
- **allowed_type** – a tuple of allowed class of input

final_check_type(*value*, *current_result*, *recursive_check_type*)
Second-step check for value. This step can use recursive check.

Parameters

- **value** – value to be checked
- **current_result** – value returned by pre_check_type. If this value is not None, the return value must be the same object.
- **recursive_check_type** – a function recursive_check_type(value, type, path=None) to be called to do recursive type check. When the call fails, path is automatically joined to create a property path

pre_check_type(*value*)
First-step check for value. This step should not do any recursive check.

Parameters **value** – value to be checked

Returns

An object if recursive creation if needed, or None if not needed.

TypeMismatchException should be raised if there is something wrong.

class pychecktype.**MapChecker**(**args*, ***kwargs*)

Check dict type, where every key is in key_type and every value is in value_type

Examples::

```
>>> check_type([], map_(int, str))
Traceback (most recent call last):
    ...
TypeMismatchException: [] cannot match type map_(<... 'int'>, <... 'str'>):␣
↪allowed types are: <... 'dict'>
>>> check_type({}, map_(int, str))
{}
>>> check_type({1: "abc"}, map_(int, str))
{1: 'abc'}
>>> m = map_()
>>> m.bind(int, m)
>>> d = {}
>>> d[1] = d
>>> check_type(d, m)
{1: {...}}
```

__repr__()

Return repr(self).

bind(*key_type*, *value_type*, *allowed_type=<class 'dict'>*, *created_type=<class 'dict'>*)

Parameters

- **key_type** – a valid type for dict key
- **value_type** – a valid type for dict value
- **allowed_type** – allowed class of the input
- **created_type** – class of the return value

final_check_type(*value*, *current_result*, *recursive_check_type*)

Second-step check for value. This step can use recursive check.

Parameters

- **value** – value to be checked
- **current_result** – value returned by pre_check_type. If this value is not None, the return value must be the same object.
- **recursive_check_type** – a function recursive_check_type(value, type, path=None) to be called to do recursive type check. When the call fails, path is automatically joined to create a property path

pre_check_type(*value*)

First-step check for value. This step should not do any recursive check.

Parameters **value** – value to be checked

Returns

An object if recursive creation if needed, or None if not needed.

TypeMismatchException should be raised if there is something wrong.

class pychecktype.**NoMatch**

A class which never matches any value Usage:

```
>>> NoMatch()
Traceback (most recent call last):
    ...
TypeError: Cannot create 'NoMatch' instances
>>> check_type({"a":1, "b":2},         {"?a": NoMatch})
Traceback (most recent call last):
    ...
TypeMismatchException: At 'a': 1 cannot match type <class '...NoMatch'>
>>> check_type({"a": 1, "b": 2},          {"a": int, "~": NoMatch})
Traceback (most recent call last):
    ...
TypeMismatchException: At 'b': 2 cannot match type <class '...NoMatch'>
```

static __new__(*self*, **args*, ***kwargs*)

Create and return a new object. See help(type) for accurate signature.

__weakref__

list of weak references to the object (if defined)

class pychecktype.**ObjectChecker**(**args*, ***kwargs*)

Check a customized object and its properties. This checker directly operate on __dict__, so magic attributes e.g. __getattr__ does not have effects.

Examples:

```
>>> class SingleLinked(object):
...     def __init__(self, name, next = None):
...         self.name = name
...         self.next = next
...
>>> class DoubleLinked(object):
...     def __init__(self, name, next = None, prev = None):
...         self.name = name
...         self.next = next
...         self.prev = prev
...
>>> single = class_()
>>> single.bind(SingleLinked, {"next": (single, None)})
>>> s1 = SingleLinked("A", SingleLinked("B", SingleLinked("C")))
>>> s2 = SingleLinked("C", SingleLinked("B", SingleLinked("A")))
>>> s2.next.next.next = s2
>>> r = check_type(s1, single)
>>> (r.name, r.next.name, r.next.next.name, r.next.next.next) == \
... ("A", "B", "C", None)
True
>>> r = check_type(s2, single)
>>> (r.name, r.next.name, r.next.next.name, r.next.next.next.name) \
... == ("C", "B", "A", "C")
True
>>> r.next.next.next is r
True
>>> r is not s2
```

(continues on next page)

(continued from previous page)

```
True
>>>
>>> single2 = class_()
>>> single2.bind(SingleLinked, {"next": (single2, None)},
... recreate_object = False)
>>> r = check_type(s1, single2)
>>> r is s1
True
>>> r = check_type(s2, single2)
>>> r is s2
True
>>>
>>> single_to_double = class_()
>>>
>>> def _modify_node(o):
...     if o.next:
...         o.next.prev = o
...     if not hasattr(o, 'prev'):
...         o.prev = None
...
>>> def _check(x):
...     if hasattr(x, 'prev'):
...         if x.prev.name == "C" and x.name == "A":
...             return False
...     if x.name == "C" and x.next is not None and \
...             hasattr(x.next, 'name') and x.next.name == 'A':
...         return False
...     return True
...
>>> single_to_double.bind(SingleLinked,
...                       {"next": (single_to_double, None)},
...                       check_before = lambda x: x.name != "",
...                       check = _check,
...                       recreate_object =
...                         lambda: DoubleLinked.__new__(
...                             DoubleLinked),
...                       modify = _modify_node)
>>>
>>> check_type(SingleLinked(""), single_to_double)
Traceback (most recent call last):
  ...
TypeMismatchException: ... cannot match type class_(...): check_before returns␣
↪False
>>> r = check_type(s1, single_to_double)
>>> (r.prev, r.name, r.next.name, r.next.next.name, r.next.next.next) \
... == (None, "A", "B", "C", None)
True
>>> r.next.prev is r
True
>>>
>>> r = check_type(s2, single_to_double)
>>> (r.prev.name, r.name, r.next.name, r.next.next.name,
... r.next.next.next.name) == \
... ("A", "C", "B", "A", "C")
True
>>> r.next.next.next is r
True
```

(continues on next page)

(continued from previous page)

```
>>> r.prev.prev.prev is r
True
>>>
>>> def _check2(x):
...     if hasattr(x, 'prev'):
...         if x.prev.name == "A" and x.name == "C":
...             return False
...     if x.name == "A" and x.next is not None and \
...             hasattr(x.next, 'name') and x.next.name == 'C':
...         return False
...     return True
...
>>> single_to_double2 = class_()
>>> single_to_double2.bind(SingleLinked,
...                        {"next": (single_to_double2, None)},
...                        check_before = lambda x: x.name != "",
...                        check = _check2,
...                        recreate_object =
...                            lambda: DoubleLinked.__new__(
...                                        DoubleLinked),
...                        modify = _modify_node)
>>> check_type(s2, single_to_double2)
Traceback (most recent call last):
  ...
TypeMismatchException: ... cannot match type ...: check returns False
```

__repr__()
Return repr(self).

bind(*object_type*, *property_check={}*, *recreate_object=True*, *check=None*, *check_before=None*, *modify=None*, *merge=<function default_object_merger>*)

Parameters

- **object_type** – a user-defined class
- **property_check** – type check for object __dict__. The checked result will be updated to object __dict__.
- **recreate_object** – if a callable is passed in, use it to create a new object; use *object_type.__new__(object_type)* to create a new object if True; use the original object else (**WARNING: this may modify the original object**)
- **check** – run an additional check for created object
- **check_before** – run a check before property checking
- **modify** – modify the object after type check
- **merge** – customize property merge process

Sequence: check object_type -> check_before -> recreate_object -> check property -> merge -> check -> modify

final_check_type(*value*, *current_result*, *recursive_check_type*)
Second-step check for value. This step can use recursive check.

Parameters

- **value** – value to be checked

- **current_result** – value returned by pre_check_type. If this value is not None, the return value must be the same object.
- **recursive_check_type** – a function recursive_check_type(value, type, path=None) to be called to do recursive type check. When the call fails, path is automatically joined to create a property path

pre_check_type(*value*)

First-step check for value. This step should not do any recursive check.

Parameters **value** – value to be checked

Returns

An object if recursive creation if needed, or None if not needed.

TypeMismatchException should be raised if there is something wrong.

class pychecktype.**TupleChecker**(**args*, ***kwargs*)

Check a tuple type: a fix-sized tuple/list, each element may have a different type

Examples:

```
>>> tuple_((str, int))
tuple_((<... 'str'>, <... 'int'>))
>>> tuple_({})
Traceback (most recent call last):
    ...
InvalidTypeException: tuple_({}) is not a valid type: must use a tuple/list of␣
↪types
>>> check_type((1,2), tuple_((1,2), allowed_type=int))
Traceback (most recent call last):
    ...
TypeMismatchException: (1, 2) cannot match type tuple_((1, 2)): allowed types␣
↪are: <... 'int'>
>>> check_type(("abc", 123), tuple_(()))
Traceback (most recent call last):
    ...
TypeMismatchException: ('abc', 123) cannot match type tuple_(()): length mismatch
>>> check_type(("abc", 123), tuple_((str, int)))
('abc', 123)
>>> check_type(["abc", 123], tuple_((str, int)))
('abc', 123)
>>> t = []
>>> tuple_type = tuple_()
>>> t.append(tuple_type)
>>> tuple_type.bind(t)
>>> l = []
>>> l.append(l)
>>> check_type(l, tuple_type) \
... # By default, a direct recursive is not allowed
Traceback (most recent call last):
    ...
TypeMismatchException: At '0': [[...]] cannot match type tuple_([...])
>>> t = []
>>> tuple_type = tuple_()
>>> t.append([tuple_type])
>>> tuple_type.bind(t)
>>> check_type(l, tuple_type) # An indirect recursive is allowed
([[[...],)],)
```

(continues on next page)

(continued from previous page)

```
>>> t = []
>>> tuple_type = tuple_()
>>> t.append(tuple_type)
>>> t.append(int)
>>> tuple_type.bind(t, allow_recursive = True)
>>> l = []
>>> l.append(l)
>>> l.append(123)
>>> check_type(l, tuple_type) \
... # allow_recursive allows a direct recursive and return list instead of tuple
[[...], 123]
```

__repr__()
Return repr(self).

bind(*tuple_of_types*, *allowed_type=(<class 'list'>, <class 'tuple'>)*, *allow_recursive=False*)

Parameters

- **tuple_of_types** – a tuple or list, each of its element is a valid type
- **allowed_type** – allowed input types
- **allow_recursive** – if False, directly recursive struct (a tuple contains itself) is not accepted, and the result is a tuple.

  if True, recursive struct is accepted and returned as a list.

final_check_type(*value*, *current_result*, *recursive_check_type*)
Second-step check for value. This step can use recursive check.

Parameters

- **value** – value to be checked
- **current_result** – value returned by pre_check_type. If this value is not None, the return value must be the same object.
- **recursive_check_type** – a function recursive_check_type(value, type, path=None) to be called to do recursive type check. When the call fails, path is automatically joined to create a property path

pre_check_type(*value*)
First-step check for value. This step should not do any recursive check.

Parameters **value** – value to be checked

Returns

An object if recursive creation if needed, or None if not needed.

TypeMismatchException should be raised if there is something wrong.

class pychecktype.**TypeChecker**(**args*, ***kwargs*)
Check an input variable is a class, and (optionally) a subclass of *baseclass*, and (optionally) has a metaclass of *metaclass*.

Examples:

```
>>> t = type_(int)
>>> t
type_(<... 'int'>)
```

(continues on next page)

(continued from previous page)

```
>>> check_type(bool, t)
<... 'bool'>
>>> check_type(str, t)
Traceback (most recent call last):
  ...
TypeMismatchException: <... 'str'> cannot match type type_(<... 'int'>): must be␣
↪a subclass of <... 'int'>
```

__repr__()
Return repr(self).

bind(*baseclass=None*, *metaclass=<class 'type'>*)

Parameters

- **baseclass** – if not None, check the input is a subclass of *baseclass*
- **metaclass** – if not None, check the input is an instance of *metaclass*

exception pychecktype.**TypeMismatchException**(*value*, *type_*, *info=None*)

__init__(*value*, *type_*, *info=None*)
Initialize self. See help(type(self)) for accurate signature.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

pychecktype.**check_type**(*value*, *type*)
Generic type checking.

Parameters

- **type** – could be:
  - a Python type. Notice that *object* matches all types, including None. There are a few special rules: int or long type always match both int and long value; str or unicode type always match both str and unicode value; int type CANNOT match bool value.
  - a tuple of type, means that data can match any subtype. When multiple subtypes can be matched, the first matched subtype is used.
  - a empty tuple () means any data type which is not None
  - None, means None. Could be used to match nullable value e.g. *(str, None)*. Equal to types.NoneType
  - a list, means that data should be a list, or a single item which is converted to a list of length 1. Tuples are also converted to lists.
  - a list with exact one valid *type*, means a list which all items are in *type*, or an item in *type* which is converted to a list. Tuples are also converted to lists.
  - a dict, means that data should be a dict
  - a dict with keys and values. Values should be valid *type*. If a key starts with '?', it is optional and '?' is removed. If a key starts with '!', it is required, and '!' is removed. If a key starts with '~', the content after '~' should be a regular expression, and any keys in *value* which matches the regular expression (with re.search) and not matched by

other keys must match the corresponding type. The behavior is undefined when a key is matched by multiple regular expressions.

If a key does not start with '?', '!' or '~', it is required, as if '!' is prepended.

- **value** – the value to be checked. It is guaranteed that this value is not modified.

Returns the checked and converted value. An exception is raised (usually TypeMismatchException) when *value* is not in *type*. The returned result may contain objects from *value*.

Some examples:

```
>>> check_type("abc", str)
'abc'
>>> check_type([1,2,3], [int])
[1, 2, 3]
>>> check_type((1,2,3), [int])
[1, 2, 3]
>>> check_type(1, ())
1
>>> check_type([[]], ())
[[]]
>>> check_type(None, ())
Traceback (most recent call last):
    ...
TypeMismatchException: None cannot match type ()
>>> check_type([1,2,"abc"], [int])
Traceback (most recent call last):
    ...
TypeMismatchException: At '2': 'abc' cannot match type <... 'int'>
>>> check_type("abc", [str])
['abc']
>>> check_type("abc", list_([str], True))
Traceback (most recent call last):
    ...
TypeMismatchException: 'abc' cannot match type [<... 'str'>]: strict mode␣
↪disables auto-convert-to-list for single value
>>> check_type(None, str)
Traceback (most recent call last):
    ...
TypeMismatchException: None cannot match type <... 'str'>
>>> check_type(None, (str, None)) is None
True
>>> check_type([1,2,"abc",["def","ghi"]], [(int, [str])])
[1, 2, ['abc'], ['def', 'ghi']]
>>> check_type({"abc":123, "def":"ghi"}, {"abc": int, "def": str}) \
... == {"abc":123, "def":"ghi"}
True
>>> check_type({"abc": {"def": "test", "ghi": 5}, "def": 1},
... {"abc": {"def": str, "ghi": int}, "def": [int]}) == \
... {"abc": {"def": "test", "ghi": 5}, "def": [1]}
True
>>> a = []
>>> a.append(a)
>>> check_type(a, a)
[[...]]
>>> r = _
>>> r[0] is r
True
```

(continues on next page)

(continued from previous page)

```
>>> check_type(1, None)
Traceback (most recent call last):
    ...
TypeMismatchException: 1 cannot match type None
>>> check_type(a, ())
[[...]]
>>> check_type(True, int)
Traceback (most recent call last):
    ...
TypeMismatchException: True cannot match type <... 'int'>
>>> check_type(1, bool)
Traceback (most recent call last):
    ...
TypeMismatchException: 1 cannot match type <... 'bool'>
>>> check_type([1], [list])
Traceback (most recent call last):
    ...
TypeMismatchException: At '0': 1 cannot match type <... 'list'>
>>> check_type(1, 1)
Traceback (most recent call last):
    ...
InvalidTypeException: 1 is not a valid type: Unrecognized type
>>> my_type = []
>>> my_type.append(([str], my_type))
>>>
>>> my_data = ["abc"]
>>> my_data.append(my_data)
>>>
>>> check_type(my_data, my_type)
[['abc'], [...]]
>>> r = _
>>> r[1] is r
True
>>> my_type = {}
>>> my_type["abc"] = my_type
>>> my_type["def"] = [my_type]
>>> my_data = {}
>>> my_data["abc"] = my_data
>>> my_data["def"] = my_data
>>> r = check_type(my_data, my_type)
>>> r['abc'] is r
True
>>> r['def'][0] is r
True
>>> my_obj = []
>>> my_obj2 = [my_obj]
>>> my_obj.append(my_obj2)
>>> my_obj.append(1)
>>> my_type = []
>>> my_type.append(my_type)
>>> check_type(my_obj, (my_type, [(my_type, int)]))
Traceback (most recent call last):
    ...
TypeMismatchException: [[[...]], 1] cannot match type ([[...]], [([[...]], <...
↪'int'>)])...
>>> my_type = []
>>> my_type.append(my_type)
```

(continues on next page)

(continued from previous page)

```
>>> check_type(1, my_type)
Traceback (most recent call last):
    ...
TypeMismatchException: 1 cannot match type [[...]]
>>> check_type(True, bool)
True
>>> check_type(1, [[[[[[[[[[int]]]]]]]]]])
[[[[[[[[[[1]]]]]]]]]]
>>> check_type([], [int, str])
Traceback (most recent call last):
    ...
InvalidTypeException: [<... 'int'>, <... 'str'>] is not a valid type: list must␣
↪contain 0 or 1 valid inner type
>>> check_type([], [])
[]
>>> check_type([1,2,3], [])
[1, 2, 3]
>>> check_type([1,"abc"], [])
[1, 'abc']
>>> check_type((1, "abc"), [])
[1, 'abc']
>>> check_type({"a": 1}, [])
[{'a': 1}]
>>> check_type(1, {})
Traceback (most recent call last):
    ...
TypeMismatchException: 1 cannot match type {}...
>>> check_type([], {})
Traceback (most recent call last):
    ...
TypeMismatchException: [] cannot match type {}...
>>> from collections import defaultdict
>>> check_type({}, dict_({}, defaultdict, lambda: defaultdict(int)))
Traceback (most recent call last):
   ...
TypeMismatchException: {} cannot match type {}: allowed types are: <...
↪'collections.defaultdict'>
>>> check_type(defaultdict(str), dict_({}, defaultdict,
... lambda: defaultdict(int)))
defaultdict(<... 'int'>, {})
>>> from collections import OrderedDict
>>> check_type(OrderedDict((("b",1),("a",2),("def","abc"))),
... dict_({"a": int, "b": int, "def": str}, dict, OrderedDict))
OrderedDict([('b', 1), ('a', 2), ('def', 'abc')])
>>> check_type({"a":1}, {})
{'a': 1}
>>> check_type({"a":1}, {"b": int})
Traceback (most recent call last):
    ...
TypeMismatchException: {'a': 1} cannot match type {'b': <... 'int'>}: key 'b' is␣
↪required
>>> check_type({"abc": 1, "abd": 2, "abe": "abc"}, {"~a.*": int})
Traceback (most recent call last):
    ...
TypeMismatchException: At 'abe': 'abc' cannot match type <... 'int'>
>>> check_type({"abc": 1, "abd": 2, "abe": "abc"},        {"~a.*": int, "abe":␣
↪str}) == {'abc': 1, 'abd': 2, 'abe': 'abc'}
```

(continues on next page)

(continued from previous page)

```
True
>>> check_type({"abc": 1, "abd": 2, "abe": "abc"},         {"~a.*": int, "?abe":␣
↪str}) == {'abc': 1, 'abd': 2, 'abe': 'abc'}
True
>>> check_type({"abc": 1, "def": "abc"}, {"abc": int}) ==         {'abc': 1, 'def
↪': 'abc'}
True
>>> check_type({"abc": 1, "abc": 2, "bcd": "abc", "bce": "abd"},
... {"~^a.*": int, "~^b.*": str}) == \
... {"abc": 1, "abc": 2, "bcd": "abc", "bce": "abd"}
True
>>> my_type = (str, [])
>>> my_type[1].append(my_type)
>>> check_type(1, my_type)
Traceback (most recent call last):
    ...
TypeMismatchException: 1 cannot match type (<... 'str'>, [(...)])...
>>> my_obj = []
>>> my_obj.append(my_obj)
>>> my_obj.append(1)
>>> check_type(my_obj, my_type)
Traceback (most recent call last):
    ...
TypeMismatchException: [[...], 1] cannot match type (<... 'str'>, [(...)])...
>>> my_obj = []
>>> my_obj.append(my_obj)
>>> my_obj.append("abc")
>>> check_type(my_obj, my_type)
[[...], 'abc']
>>> my_type = []
>>> my_type2 = {"a": my_type, "b": my_type}
>>> my_type.append(my_type2)
>>> my_obj = {}
>>> my_obj['a'] = my_obj
>>> my_obj['b'] = my_obj
>>> r = check_type(my_obj, my_type)
>>> r[0]['a'][0] is r[0]['b'][0]
True
>>> r[0]['a'][0] is r[0]
True
>>> r = check_type(my_obj, my_type2)
>>> r['a'][0] is r['b'][0]
True
>>> r['a'][0] is r
True
>>> my_obj2 = []
>>> my_obj2.append(my_obj2)
>>> my_obj2.append(1)
>>> my_obj = [my_obj2, my_obj2]
>>> my_type = []
>>> my_type.append((int, my_type))
>>> check_type(my_obj, my_type)
[[[...], 1], [[...], 1]]
>>> r = _
>>> r[0] is r[1]
True
>>> my_type = []
```

(continues on next page)

(continued from previous page)

```
>>> my_type.append(([int], my_type))
>>> check_type(my_obj, my_type)
[[[...], [1]], [[...], [1]]]
>>> r = _
>>> r[0] is r[1]
True
>>> check_type({"abc": {"def": "123"}}, {"abc": {"def": int}})
Traceback (most recent call last):
   ...
TypeMismatchException: At 'abc.def': '123' cannot match type <... 'int'>
>>> check_type({"abc": [{"def": 123}, {"def": "123"}]}, {"abc": [{"def": int}]})
Traceback (most recent call last):
   ...
TypeMismatchException: At 'abc.1.def': '123' cannot match type <... 'int'>
>>> check_type({"abc": [{"def": 123}, {"def": "123"}]}, ({"abc": [{"def": int}]},
↪{"abc": [{"def": str}]}))
Traceback (most recent call last):
   ...
TypeMismatchException: {'abc': [{'def': 123}, {'def': '123'}]} cannot match type (
↪{'abc': [{'def': <... 'int'>}]}, {'abc': [{'def': <... 'str'>}]}): Not matched␣
↪by any of the sub types:
  At 'abc.1.def': '123' cannot match type <... 'int'>
  At 'abc.0.def': 123 cannot match type <... 'str'>
>>> check_type({"abc": 123, "def": "abc"}, map_(str, str))
Traceback (most recent call last):
   ...
TypeMismatchException: At 'abc': 123 cannot match type <... 'str'>
>>> check_type({"abc": {"abc": 123, 123: "abc"}}, {"abc": map_(int, str)})
Traceback (most recent call last):
   ...
TypeMismatchException: At 'abc.<Key>': 'abc' cannot match type <... 'int'>
```

pychecktype.**class_**
alias of *pychecktype.ObjectChecker*

pychecktype.**dict_**
alias of *pychecktype.DictChecker*

pychecktype.**extra**
alias of *pychecktype.ExtraChecker*

pychecktype.**list_**
alias of *pychecktype.ListChecker*

pychecktype.**map_**
alias of *pychecktype.MapChecker*

pychecktype.**tuple_**
alias of *pychecktype.TupleChecker*

pychecktype.**type_**(*baseclass=None*, *metaclass=<class 'type'>*)
Create a TypeChecker

Python 3 annotation based type-check

pychecktype.checked.**checked**(*f*)
Check input types with annotations

Examples:

```
>>> @checked
... def test(a: (str,int), b: (str,int))->str:
...     return a + b
...
>>> test('a','b')
'ab'
>>> test(1,2)
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At '<return>': 3 cannot match type <class 'str
↪'>
>>> test(1.0,2)
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At 'a': 1.0 cannot match type (<class 'str'>,
↪<class 'int'>)...
>>> import asyncio
>>> @checked
... async def test2(a: (str,int), b: (str,int))->str:
...     return a + b
...
>>> asyncio.get_event_loop().run_until_complete(test2(1,2))
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At '<return>': 3 cannot match type <class 'str
↪'>
>>> @checked
... def test3(a: str, *args: [int], **kwargs: {'?join': bool}):
...     if kwargs.get('join'):
...         return a.join(str(v) for v in args)
...     else:
...         return a + str(sum(args))
...
>>> test3('a',2,3)
'a5'
>>> test3('a','b',2)
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At 'args.0': 'b' cannot match type <class 'int
↪'>
>>> test3('a',5,join=True)
'5'
>>> test3('a',5,join=1)
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At 'kwargs.join': 1 cannot match type <class
↪'bool'>
>>> @checked
... async def test3(a: str, *args: [int], **kwargs: {'?join': bool}):
...     if kwargs.get('join'):
...         return a.join(str(v) for v in args)
...     else:
...         return a + str(sum(args))
...
>>> asyncio.get_event_loop().run_until_complete(test3('a',2,3))
'a5'
>>> asyncio.get_event_loop().run_until_complete(test3('a','b',2))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At 'args.0': 'b' cannot match type <class 'int
↪'>
>>> asyncio.get_event_loop().run_until_complete(test3('a',5,join=True))
'5'
>>> asyncio.get_event_loop().run_until_complete(test3('a',5,join=1))
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At 'kwargs.join': 1 cannot match type <class
↪'bool'>
>>> @checked
... def f(a, b: int):
...     return a + b
...
>>> f('a','b')
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At 'b': 'b' cannot match type <class 'int'>
>>> f(1,2)
3
>>> from functools import wraps
>>> def testdecorator(f):
...     @wraps(f)
...     def _f(*args, **kwargs):
...         print("Wrapped")
...         return f(*args, **kwargs)
...     return _f
...
>>> @checked
... @testdecorator
... def f2(a: int):
...     return a
...
>>> f2(1)
Wrapped
1
>>> f2('a')
Traceback (most recent call last):
  ...
pychecktype.TypeMismatchException: At 'a': 'a' cannot match type <class 'int'>
```

CHAPTER 2

pychecktype

A type-checker which can process recursive types and data

2.1 Install

```
pip install pychecktype
```

2.2 Basic Usage

```
from pychecktype import check_type

check_type({"abc": [1,2,3], "def": {"test": "abc"}}, {"abc": [int], "def": {"test":␣
↪[str]}})

# Returns: {"abc": [1,2,3], "def": {"test": ["abc"]}}
```

2.3 Highlight

The most intersting thing of this implementation is that it fully support recursive types and data, for example:

```
from pychecktype import check_type


my_type = []
my_type.append((int, my_type))

# my_type accepts: recursive lists with only sub-list and integers with any depth -␣
↪even infinite
```

(continues on next page)

(continued from previous page)

```
check_type([], my_type) # []

check_type([1,2,3,[1,2],[1,2,[3,4]]], my_type) # [1,2,3,[1,2],[1,2,[3,4]]]

check_type([1,2,3,[1,2],[1,2,["3",4]]], my_type) # failed

my_obj = []
my_obj.append(my_obj)
my_obj.append(2)
check_type(my_obj, my_type) # [[...], 2]
```

2.4 Rules

This type-checker has some specialized rules suitable for YAML. For example, this type-checker accepts a single value against a list type, and convert the value to `[value]`.

This type-checker uses a slightly simpler and more readable DSL rules than other libraries like `typing` and `trafaret`, most of them are Python builtin objects.

The `check_type` method not only checks that the value is matched with the given type; it returns a *corrected* version of that object.

Generally:

1. A Python type matches any object in that type (e.g str, int) except:

1. `str` and `unicode` always match both str and unicode objects both in Python 2 and Python 3
2. `int` and `long` always match both int and long objects both in Python 2 and Python 3
3. bool objects are never matched with `int` or `long`, they are only matched with `bool` (though `bool` is a subclass of `int`)

Specially, `object` matches any value including `None`. A helper class `NoMatch` is provided to do not match any instances, it can be embedded in other types to create assertions.

2. `None` matches `None` only (equivalent to `NoneType`)
3. Tuple as a type:

1. `()` matches any object *EXCEPT* `None`
2. A tuple of multiple valid types `(type1, type2, ...)` tries to match the object with each sub-type from left to right. For example, `(str, int)` matches a str object or an int object; `(str, None)` matches a str object or None
4. List as a type:

1. `[]` matches any list, or convert the object to a list contains the object
2. `[type]` matches a list of items which all match the inner type, or convert an object which matches with the inner type to a list contains it
3. By default, list types matches both *list* objects and *tuple* objects, and convert them to lists. For example, `[int]` matches `(1,2,3)` and returns `[1,2,3]`. Use `list_` factory method to create a customized list type which accepts only types that are specified. You may also use it to accept more iterable types e.g. `set`

4. By default, list types can convert non-list objects to a list contains only that object, e.g. `1` to `[1]`, `{"a":1}` to `[{"a":1}]`. This conversion cannot happen when the object itself is a list/tuple, e.g. `[list]` cannot match `[1]`, because it is not allowed to be converted to `[[1]]`.

   You may disable the conversion by creating a customized list type with `list_` factory method with `strict=True`

5. List types return a shallow copy of the input list.

5. Dist as a type:

1. `{}` matches any dict

2. When dict contains key-value pairs, they become restricts to the input dict:

   1). Keys start with '!' are required keys, and the corresponding value is a type. The value of the specified key in the input dict must match the specified type in the type dict.

   2). Keys start with '?' are optional keys, they are not needed to appear in the input dict, but if they appear they must be matched with the value in the type dict.

   3). Keys start with '~' are regular expressions. For all keys in the input dict that are matched by the regular expression followed by the '~', the corresponding value must match with the specified type. Regular expressions only match the keys that are not required or optional keys.

   4). Other keys are regarded as required keys (as if they are prepended by '!')

   5). Extra keys in the input dict do not affect the match. You may use `'~': NoMatch` to disable extra keys.

   Examples:

   {"abc": int} matches *{"abc": 1}* and *{"abc": 1, "d": 2}* but not *{"d": 2}*

   {"!abc": int} matches *{"abc": 1}* and *{"abc": 1, "d": 2}* but not *{"d": 2}*

   {"?abc": int} matches *{"abc": 1}*, *{"abc": 1, "d": 2}* and *{"d": 1}*, but not *{"abc": "a"}*

   {"~a.b": int} matches *{"acb": 1}* but not *{"facbg": "a"}* because "facbg" is matched by 'a.b'

   {"~a.b": int, "adb": str} matches *{"adb": "abc"}* but not *{"adb": 1}*

6. `tuple_((type1, type2, type3, ...))` creates a customized type (tuple type) which matches any tuple/list that contains exactly the same number of items, each matches the corresponding sub type.

7. `map_(key_type, value_type)` creates a customized type (map type) which matches any dict, in which each key matches the *key_type*, and each value matches the *value_type*

8. `extra_` and `class_` are advanced customized types, they do customized additional checks for the input object e.g. check against a regular expression etc.

See docstring in pychecktype.py for details.

2.5 Python 3 Annotation Checks

You may use *pychecktype.checked.checked* decorator to check input parameters and return values of a function

```
from pychecktype.checked import checked
@checked
def f(a: str, b: int)->str:
    """
    check `a` is str, `b` is int, and returns str
    """
```

(continues on next page)

(continued from previous page)

```
    return a + str(b)

@checked
def f2(a, b: int):
    """
    You may check only part of the parameters.
    """
    return str(a) + str(b)

@checked
async def f3(a: str, *args: [int], **kwargs: {'?join': bool})->str:
    """
    Async functions are decorated to async functions

    *args , keyword-only arguments and **kwargs can also be checked
    """
    if kwargs.get('join'):
        return a.join(str(v) for v in args)
    else:
        return a + str(sum(args))

from functools import wraps
def testdecorator(f):
    @wraps
    def _f(*args, **kwargs):
        print("Wrapped")
        return f(*args, **kwargs)

@checked
@testdecorator
def f4(a: int):
    """
    Works well with decorators that are correctly using `functools.wraps`
    and not modifying the argument list
    """
    return a + 1
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

B

C

D

E

F

I

L

M

N

O

P

T