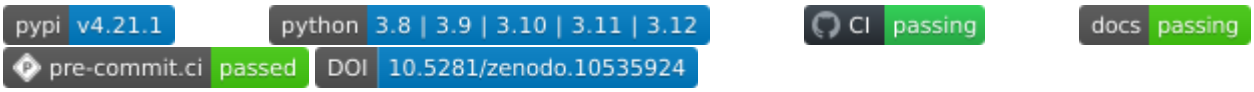
# jsonschema

***Release 4.21.2.dev43+gc3729db***

**Julian Berman**

**Apr 26, 2024**

# CONTENTS




`jsonschema` is an implementation of the JSON Schema specification for Python.

```
>>> from jsonschema import validate

>>> # A sample schema, like what we'd get from json.load()
>>> schema = {
...     "type" : "object",
...     "properties" : {
...         "price" : {"type" : "number"},
...         "name" : {"type" : "string"},
...     },
... }

>>> # If no exception is raised by validate(), the instance is valid.
>>> validate(instance={"name" : "Eggs", "price" : 34.99}, schema=schema)

>>> validate(
...     instance={"name" : "Eggs", "price" : "Invalid"}, schema=schema,
... )
Traceback (most recent call last):
    ...
ValidationError: 'Invalid' is not of type 'number'
```

It can also be used from the command line by installing check-jsonschema.

CHAPTER

ONE

# FEATURES

- Full support for Draft 2020-12, Draft 2019-09, Draft 7, Draft 6, Draft 4 and Draft 3
- Lazy validation that can iteratively report *all* validation errors.
- Programmatic querying of which properties or items failed validation.

CHAPTER

TWO

# INSTALLATION

`jsonschema` is available on PyPI. You can install using pip:

```
$ pip install jsonschema
```

## 2.1 Extras

Two extras are available when installing the package, both currently related to `format` validation:

- `format`
- `format-nongpl`

They can be used when installing in order to include additional dependencies, e.g.:

```
$ pip install jsonschema'[format]'
```

Be aware that the mere presence of these dependencies – or even the specification of `format` checks in a schema – do *not* activate format checks (as per the specification). Please read the format validation documentation for further details.

CHAPTER

THREE

# RUNNING THE TEST SUITE

If you have `nox` installed (perhaps via `pipx install nox` or your package manager), running `nox` in the directory of your source checkout will run `jsonschema`'s test suite on all of the versions of Python `jsonschema` supports. If you don't have all of the versions that `jsonschema` is tested under, you'll likely want to run using `nox`'s `--no-error-on-missing-interpreters` option.

Of course you're also free to just run the tests on a single version with your favorite test runner. The tests live in the `jsonschema.tests` package.

CHAPTER

FOUR

# BENCHMARKS

`jsonschema`'s benchmarks make use of pyperf. Running them can be done via:

```
$ nox -s perf
```

CHAPTER

FIVE

# COMMUNITY

The JSON Schema specification has a Slack, with an invite link on its home page. Many folks knowledgeable on authoring schemas can be found there.

Otherwise, opening a GitHub discussion or asking questions on Stack Overflow are other means of getting help if you're stuck.

CHAPTER

SIX

# ABOUT

I'm Julian Berman.

`jsonschema` is on GitHub.

Get in touch, via GitHub or otherwise, if you've got something to contribute, it'd be most welcome!

You can also generally find me on Libera (nick: `Julian`) in various channels, including `#python`.

If you feel overwhelmingly grateful, you can also sponsor me.

And for companies who appreciate `jsonschema` and its continued support and growth, `jsonschema` is also now supportable via TideLift.

CHAPTER

SEVEN

# CONTENTS

## 7.1 Schema Validation

**Tip:** Most of the documentation for this package assumes you're familiar with the fundamentals of writing JSON schemas themselves, and focuses on how this library helps you validate with them in Python.

If you aren't already comfortable with writing schemas and need an introduction which teaches about JSON Schema the specification, you may find Understanding JSON Schema to be a good read!

### 7.1.1 The Basics

The simplest way to validate an instance under a given schema is to use the *validate* function.

jsonschema.**validate**(*instance*, *schema*, *cls=None*, **args*, ***kwargs*)

Validate an instance under the given schema.

```
>>> validate([2, 3, 4], {"maxItems": 2})
Traceback (most recent call last):
    ...
ValidationError: [2, 3, 4] is too long
```

*validate()* will first verify that the provided schema is itself valid, since not doing so can lead to less obvious error messages and fail in less obvious or consistent ways.

If you know you have a valid schema already, especially if you intend to validate multiple instances with the same schema, you likely would prefer using the *jsonschema.protocols.Validator.validate* method directly on a specific validator (e.g. `Draft202012Validator.validate`).

**Parameters**

- **instance** – The instance to validate
- **schema** – The schema to validate with
- **cls** (jsonschema.protocols.Validator) – The class that will be used to validate the instance.

If the `cls` argument is not provided, two things will happen in accordance with the specification. First, if the schema has a $schema keyword containing a known meta-schema[1] then the proper validator will be used. The specification recommends that all schemas contain $schema properties for this reason. If no $schema property is found, the default validator class is the latest released draft.

[1] known by a validator registered with *jsonschema.validators.validates*

Any other provided positional and keyword arguments will be passed on when instantiating the `cls`.

**Raises**

- *jsonschema.exceptions.ValidationError* – if the instance is invalid
- *jsonschema.exceptions.SchemaError* – if the schema itself is invalid

### 7.1.2 The Validator Protocol

*jsonschema* defines a `protocol` that all validator classes adhere to.

---

**Hint:** If you are unfamiliar with protocols, either as a general notion or as specifically implemented by `typing.Protocol`, you can think of them as a set of attributes and methods that all objects satisfying the protocol have.

Here, in the context of *jsonschema*, the *Validator.iter_errors* method can be called on *jsonschema.validators.Draft202012Validator*, or *jsonschema.validators.Draft7Validator*, or indeed any validator class, as all of them have it, along with all of the other methods described below.

---

**class** jsonschema.protocols.**Validator**(*schema: Mapping | bool*, *registry: referencing.jsonschema.SchemaRegistry*, *format_checker:* jsonschema.FormatChecker | *None* = *None*)

The protocol to which all validator classes adhere.

**Parameters**

- **schema** – The schema that the validator object will validate with. It is assumed to be valid, and providing an invalid schema can lead to undefined behavior. See *Validator.check_schema* to validate a schema first.
- **registry** – a schema registry that will be used for looking up JSON references
- **resolver** – a resolver that will be used to resolve $ref properties (JSON references). If unprovided, one will be created.

  Deprecated since version v4.18.0: *RefResolver* has been deprecated in favor of *JSON (Schema) Referencing*, and with it, this argument.
- **format_checker** – if provided, a checker which will be used to assert about format properties present in the schema. If unprovided, *no* format validation is done, and the presence of format within schemas is strictly informational. Certain formats require additional packages to be installed in order to assert against instances. Ensure you've installed *jsonschema* with its *extra (optional) dependencies* when invoking `pip`.

Deprecated since version v4.12.0: Subclassing validator classes now explicitly warns this is not part of their public API.

**FORMAT_CHECKER: ClassVar[*jsonschema.FormatChecker*]**

A *jsonschema.FormatChecker* that will be used when validating format keywords in JSON schemas.

**ID_OF: *_typing.id_of***

A function which given a schema returns its ID.

**META_SCHEMA: ClassVar[Mapping]**

An object representing the validator's meta schema (the schema that describes valid schemas in the given version).

**TYPE_CHECKER: ClassVar[*jsonschema.TypeChecker*]**

A *jsonschema.TypeChecker* that will be used when validating type keywords in JSON schemas.

**VALIDATORS: ClassVar[Mapping]**

A mapping of validation keywords (strs) to functions that validate the keyword with that name. For more information see *Creating or Extending Validator Classes*.

**classmethod check_schema**(*schema: Mapping | bool*) → None

Validate the given schema against the validator's *META_SCHEMA*.

**Raises**

***jsonschema.exceptions.SchemaError*** – if the schema is invalid

**evolve**(***kwargs*) → *Validator*

Create a new validator like this one, but with given changes.

Preserves all other attributes, so can be used to e.g. create a validator with a different schema but with the same $ref resolution behavior.

```
>>> validator = Draft202012Validator({})
>>> validator.evolve(schema={"type": "number"})
Draft202012Validator(schema={'type': 'number'}, format_checker=None)
```

The returned object satisfies the validator protocol, but may not be of the same concrete class! In particular this occurs when a $ref occurs to a schema with a different $schema than this one (i.e. for a different draft).

```
>>> validator.evolve(
...     schema={"$schema": Draft7Validator.META_SCHEMA["$id"]}
... )
Draft7Validator(schema=..., format_checker=None)
```

**is_type**(*instance: Any*, *type: str*) → bool

Check if the instance is of the given (JSON Schema) type.

**Parameters**

- **instance** – the value to check
- **type** – the name of a known (JSON Schema) type

**Returns**

whether the instance is of the given type

**Raises**

***jsonschema.exceptions.UnknownType*** – if `type` is not a known type

**is_valid**(*instance: Any*) → bool

Check if the instance is valid under the current *schema*.

**Returns**

whether the instance is valid or not

```
>>> schema = {"maxItems" : 2}
>>> Draft202012Validator(schema).is_valid([2, 3, 4])
False
```

**iter_errors**(*instance: Any*) → Iterable[*ValidationError*]

Lazily yield each of the validation errors in the given instance.

```
>>> schema = {
...     "type" : "array",
...     "items" : {"enum" : [1, 2, 3]},
...     "maxItems" : 2,
... }
>>> v = Draft202012Validator(schema)
>>> for error in sorted(v.iter_errors([2, 3, 4]), key=str):
...     print(error.message)
4 is not one of [1, 2, 3]
[2, 3, 4] is too long
```

Deprecated since version v4.0.0: Calling this function with a second schema argument is deprecated. Use *Validator.evolve* instead.

**schema:** Mapping | bool

The schema that will be used to validate instances

**validate**(*instance: Any*) → None

Check if the instance is valid under the current *schema*.

**Raises**

***jsonschema.exceptions.ValidationError*** – if the instance is invalid

```
>>> schema = {"maxItems" : 2}
>>> Draft202012Validator(schema).validate([2, 3, 4])
Traceback (most recent call last):
    ...
ValidationError: [2, 3, 4] is too long
```

All of the *versioned validators* that are included with *jsonschema* adhere to the protocol, and any *extensions of these validators* will as well. For more information on *creating* or *extending* validators see *Creating or Extending Validator Classes*.

### 7.1.3 Type Checking

To handle JSON Schema's type keyword, a *Validator* uses an associated *TypeChecker*. The type checker provides an immutable mapping between names of types and functions that can test if an instance is of that type. The defaults are suitable for most users - each of the *versioned validators* that are included with *jsonschema* have a *TypeChecker* that can correctly handle their respective versions.

**See also:**

*Validating With Additional Types*

For an example of providing a custom type check.

**class** jsonschema.**TypeChecker**(*type_checkers: Mapping[str, Callable[[TypeChecker, Any], bool]] = HashTrieMap({})*)

A type property checker.

A *TypeChecker* performs type checking for a *Validator*, converting between the defined JSON Schema types and some associated Python types or objects.

Modifying the behavior just mentioned by redefining which Python objects are considered to be of which JSON Schema types can be done using *TypeChecker.redefine* or *TypeChecker.redefine_many*, and types can be removed via *TypeChecker.remove*. Each of these return a new *TypeChecker*.

**Parameters**
**type_checkers** – The initial mapping of types to their checking functions.

**is_type**(*instance*, *type: str*) → bool

Check if the instance is of the appropriate type.

**Parameters**

- **instance** – The instance to check
- **type** – The name of the type that is expected.

**Raises**
*jsonschema.exceptions.UndefinedTypeCheck* – if `type` is unknown to this object.

**redefine**(*type: str*, *fn*) → *TypeChecker*

Produce a new checker with the given type redefined.

**Parameters**

- **type** – The name of the type to check.
- **fn** (*collections.abc.Callable*) – A callable taking exactly two parameters - the type checker calling the function and the instance to check. The function should return true if instance is of this type and false otherwise.

**redefine_many**(*definitions=()*) → *TypeChecker*

Produce a new checker with the given types redefined.

**Parameters**
**definitions** (*dict*) – A dictionary mapping types to their checking functions.

**remove**(**types*) → *TypeChecker*

Produce a new checker with the given types forgotten.

**Parameters**
**types** – the names of the types to remove.

**Raises**
*jsonschema.exceptions.UndefinedTypeCheck* – if any given type is unknown to this object

**exception** jsonschema.exceptions.**UndefinedTypeCheck**(*type*)

A type checker was asked to check a type it did not have registered.

Raised when trying to remove a type check that is not known to this TypeChecker, or when calling *jsonschema.TypeChecker.is_type* directly.

### Validating With Additional Types

Occasionally it can be useful to provide additional or alternate types when validating JSON Schema's type keyword.

*jsonschema* tries to strike a balance between performance in the common case and generality. For instance, JSON Schema defines a `number` type, which can be validated with a schema such as `{"type" : "number"}`. By default, this will accept instances of Python `numbers.Number`. This includes in particular `int`s and `float`s, along with `decimal.Decimal` objects, `complex` numbers etc. For `integer` and `object`, however, rather than checking for `numbers.Integral` and `collections.abc.Mapping`, *jsonschema* simply checks for `int` and `dict`, since the more general instance checks can introduce significant slowdown, especially given how common validating these types are.

If you *do* want the generality, or just want to add a few specific additional types as being acceptable for a validator object, then you should update an existing *jsonschema.TypeChecker* or create a new one. You may then create a new *Validator* via *jsonschema.validators.extend*.

```
from jsonschema import validators

class MyInteger:
    pass

def is_my_int(checker, instance):
    return (
        Draft202012Validator.TYPE_CHECKER.is_type(instance, "number") or
        isinstance(instance, MyInteger)
    )

type_checker = Draft202012Validator.TYPE_CHECKER.redefine(
    "number", is_my_int,
)

CustomValidator = validators.extend(
    Draft202012Validator,
    type_checker=type_checker,
)
validator = CustomValidator(schema={"type" : "number"})
```

**exception** jsonschema.exceptions.**UnknownType**(*type*, *instance*, *schema*)

A validator was asked to validate an instance against an unknown type.

### 7.1.4 Versioned Validators

*jsonschema* ships with validator classes for various versions of the JSON Schema specification. For details on the methods and attributes that each validator class provides see the *Validator* protocol, which each included validator class implements.

Each of the below cover a specific release of the JSON Schema specification.

**class** jsonschema.**Draft202012Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**class** jsonschema.**Draft201909Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**class** jsonschema.**Draft7Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**class** jsonschema.**Draft6Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**class** jsonschema.**Draft4Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**class** jsonschema.**Draft3Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

For example, if you wanted to validate a schema you created against the Draft 2020-12 meta-schema, you could use:

```
from jsonschema import Draft202012Validator

schema = {
    "$schema": Draft202012Validator.META_SCHEMA["$id"],

    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "email": {"type": "string"},
    },
    "required": ["email"]
}
Draft202012Validator.check_schema(schema)
```

### 7.1.5 Validating Formats

JSON Schema defines the format keyword which can be used to check if primitive types (`strings`, `numbers`, `booleans`) conform to well-defined formats. By default, as per the specification, no validation is enforced. Optionally however, validation can be enabled by hooking a *format-checking object* into a *Validator*.

```
>>> validate("127.0.0.1", {"format" : "ipv4"})
>>> validate(
...     instance="-12",
...     schema={"format" : "ipv4"},
...     format_checker=Draft202012Validator.FORMAT_CHECKER,
... )
Traceback (most recent call last):
    ...
ValidationError: "-12" is not a "ipv4"
```

Some formats require additional dependencies to be installed.

The easiest way to ensure you have what is needed is to install `jsonschema` using the `format` or `format-nongpl` extras.

For example:

```
$ pip install jsonschema[format]
```

Or if you want to avoid GPL dependencies, a second extra is available:

```
$ pip install jsonschema[format-nongpl]
```

At the moment, it supports all the available checkers except for `iri` and `iri-reference`.

**Warning:** It is your own responsibility ultimately to ensure you are license-compliant, so you should be double checking your own dependencies if you rely on this extra.

The more specific list of formats along with any additional dependencies they have is shown below.

**Warning:** If a dependency is not installed when using a checker that requires it, validation will succeed without throwing an error, as also specified by the specification.

| Checker | Notes |
|---|---|
| `color` | requires webcolors |
| `date` | |
| `date-time` | requires rfc3339-validator |
| `duration` | requires isoduration |
| `email` | |
| `hostname` | requires fqdn |
| `idn-hostname` | requires idna |
| `ipv4` | |
| `ipv6` | OS must have `socket.inet_pton` function |
| `iri` | requires rfc3987 |
| `iri-reference` | requires rfc3987 |
| `json-pointer` | requires jsonpointer |
| `regex` | |
| `relative-json-pointer` | requires jsonpointer |
| `time` | requires rfc3339-validator |
| `uri` | requires rfc3987 or rfc3986-validator |
| `uri-reference` | requires rfc3987 or rfc3986-validator |
| `uri-template` | requires uri-template |

The supported mechanism for ensuring these dependencies are present is again as shown above, not by directly installing the packages.

**class** jsonschema.**FormatChecker**(*formats: Iterable[str] | None = None*)

A `format` property checker.

JSON Schema does not mandate that the `format` property actually do any validation. If validation is desired however, instances of this class can be hooked into validators to enable format validation.

*FormatChecker* objects always return `True` when asked about formats that they do not know how to validate.

To add a check for a custom format use the *FormatChecker.checks* decorator.

**Parameters**
**formats** – The known formats to validate. This argument can be used to limit which formats will be used during validation.

**checkers**

A mapping of currently known formats to tuple of functions that validate them and errors that should be caught. New checkers can be added and removed either per-instance or globally for all checkers using the *FormatChecker.checks* decorator.

**classmethod cls_checks**(*format*, *raises=()*)

Register a decorated function as *globally* validating a new format.

Any instance created after this function is called will pick up the supplied checker.

**Parameters**

- **format** (*str*) – the format that the decorated function will check
- **raises** (*Exception*) – the exception(s) raised by the decorated function when an invalid instance is found. The exception object will be accessible as the *jsonschema.exceptions.ValidationError.cause* attribute of the resulting validation error.

Deprecated since version v4.14.0: Use *FormatChecker.checks* on an instance instead.

**check**(*instance: object*, *format: str*) → None

Check whether the instance conforms to the given format.

**Parameters**

- **instance** (*any primitive type*, i.e. str, number, bool) – The instance to check
- **format** – The format that instance should conform to

**Raises**
***FormatError*** – if the instance does not conform to `format`

**checks**(*format: str*, *raises: Type[Exception] | Tuple[Type[Exception], ...] = ()*) → Callable[[_F], _F]

Register a decorated function as validating a new format.

**Parameters**

- **format** – The format that the decorated function will check.
- **raises** – The exception(s) raised by the decorated function when an invalid instance is found.

  The exception object will be accessible as the *jsonschema.exceptions.ValidationError.cause* attribute of the resulting validation error.

**conforms**(*instance: object*, *format: str*) → bool

Check whether the instance conforms to the given format.

**Parameters**

- **instance** (*any primitive type*, i.e. str, number, bool) – The instance to check
- **format** – The format that instance should conform to

**Returns**
whether it conformed

**Return type**
bool

**exception** jsonschema.exceptions.**FormatError**(*message*, *cause=None*)

Validating a format failed.

### Format-Specific Notes

#### regex

The JSON Schema specification recommends (but does not require) that implementations use ECMA 262 regular expressions.

Given that there is no current library in Python capable of supporting the ECMA 262 dialect, the `regex` format will instead validate *Python* regular expressions, which are the ones used by this implementation for other keywords like pattern or patternProperties.

#### email

Since in most cases "validating" an email address is an attempt instead to confirm that mail sent to it will deliver to a recipient, and that that recipient is the correct one the email is intended for, and since many valid email addresses are in many places incorrectly rejected, and many invalid email addresses are in many places incorrectly accepted, the `email` format keyword only provides a sanity check, not full **RFC 5322** validation.

The same applies to the `idn-email` format.

If you indeed want a particular well-specified set of emails to be considered valid, you can use *FormatChecker.checks* to provide your specific definition.

## 7.2 Handling Validation Errors

When an invalid instance is encountered, a *ValidationError* will be raised or returned, depending on which method or function is used.

**exception** jsonschema.exceptions.**ValidationError**(*message: str*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*, *type_checker=<unset>*)

An instance was invalid under a provided schema.

The information carried by an error roughly breaks down into:

| What Happened | Why Did It Happen | What Was Being Validated |
|---|---|---|
| *message* | *context*<br>*cause* | *instance*<br>*json_path*<br>*path*<br>*schema*<br>*schema_path*<br>*validator*<br>*validator_value* |

**message**

A human readable message explaining the error.

**validator**

The name of the failed keyword.

**validator_value**

The associated value for the failed keyword in the schema.

**schema**

The full schema that this error came from. This is potentially a subschema from within the schema that was passed in originally, or even an entirely different schema if a $ref was followed.

**relative_schema_path**

A `collections.deque` containing the path to the failed keyword within the schema.

**absolute_schema_path**

A `collections.deque` containing the path to the failed keyword within the schema, but always relative to the *original* schema as opposed to any subschema (i.e. the one originally passed into a validator class, *not* `schema`).

**schema_path**

Same as *`relative_schema_path`*.

**relative_path**

A `collections.deque` containing the path to the offending element within the instance. The deque can be empty if the error happened at the root of the instance.

**absolute_path**

A `collections.deque` containing the path to the offending element within the instance. The absolute path is always relative to the *original* instance that was validated (i.e. the one passed into a validation method, *not* `instance`). The deque can be empty if the error happened at the root of the instance.

**json_path**

A JSON path to the offending element within the instance.

**path**

Same as *`relative_path`*.

**instance**

The instance that was being validated. This will differ from the instance originally passed into `validate` if the validator object was in the process of validating a (possibly nested) element within the top-level instance. The path within the top-level instance (i.e. *`ValidationError.path`*) could be used to find this object, but it is provided for convenience.

**context**

If the error was caused by errors in subschemas, the list of errors from the subschemas will be available on this property. The *`schema_path`* and *`path`* of these errors will be relative to the parent error.

**cause**

If the error was caused by a *non*-validation error, the exception object will be here. Currently this is only used for the exception raised by a failed format checker in *`jsonschema.FormatChecker.check`*.

**parent**

A validation error which this error is the *`context`* of. `None` if there wasn't one.

In case an invalid schema itself is encountered, a *`SchemaError`* is raised.

**exception** jsonschema.exceptions.**SchemaError**(*message: str*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*, *type_checker=<unset>*)

A schema was invalid under its corresponding metaschema.

The same attributes are present as for *ValidationError*s.

These attributes can be clarified with a short example:

```
schema = {
    "items": {
        "anyOf": [
            {"type": "string", "maxLength": 2},
            {"type": "integer", "minimum": 5}
        ]
    }
}
instance = [{}, 3, "foo"]
v = Draft202012Validator(schema)
errors = sorted(v.iter_errors(instance), key=lambda e: e.path)
```

The error messages in this situation are not very helpful on their own.

```
for error in errors:
    print(error.message)
```

outputs:

```
{} is not valid under any of the given schemas
3 is not valid under any of the given schemas
'foo' is not valid under any of the given schemas
```

If we look at *ValidationError.path* on each of the errors, we can find out which elements in the instance correspond to each of the errors. In this example, *ValidationError.path* will have only one element, which will be the index in our list.

```
for error in errors:
    print(list(error.path))
```

```
[0]
[1]
[2]
```

Since our schema contained nested subschemas, it can be helpful to look at the specific part of the instance and subschema that caused each of the errors. This can be seen with the *ValidationError.instance* and *ValidationError.schema* attributes.

With keywords like anyOf, the *ValidationError.context* attribute can be used to see the sub-errors which caused the failure. Since these errors actually came from two separate subschemas, it can be helpful to look at the *ValidationError.schema_path* attribute as well to see where exactly in the schema each of these errors come from. In the case of sub-errors from the *ValidationError.context* attribute, this path will be relative to the *ValidationError.schema_path* of the parent error.

```
for error in errors:
    for suberror in sorted(error.context, key=lambda e: e.schema_path):
        print(list(suberror.schema_path), suberror.message, sep=", ")
```

```
[0, 'type'], {} is not of type 'string'
[1, 'type'], {} is not of type 'integer'
[0, 'type'], 3 is not of type 'string'
[1, 'minimum'], 3 is less than the minimum of 5
[0, 'maxLength'], 'foo' is too long
[1, 'type'], 'foo' is not of type 'integer'
```

The string representation of an error combines some of these attributes for easier debugging.

```
print(errors[1])
```

```
3 is not valid under any of the given schemas

Failed validating 'anyOf' in schema['items']:
    {'anyOf': [{'maxLength': 2, 'type': 'string'},
               {'minimum': 5, 'type': 'integer'}]}

On instance[1]:
    3
```

### 7.2.1 ErrorTrees

If you want to programmatically query which validation keywords failed when validating a given instance, you may want to do so using *jsonschema.exceptions.ErrorTree* objects.

**class** jsonschema.exceptions.**ErrorTree**(*errors: Iterable[*ValidationError*] = ()*)

ErrorTrees make it easier to check which validations failed.

**errors**

The mapping of validation keywords to the error objects (usually *jsonschema.exceptions.ValidationError*s) at this level of the tree.

**__contains__**(*index: str | int*)

Check whether `instance[index]` has any errors.

**__getitem__**(*index*)

Retrieve the child tree one level down at the given `index`.

If the index is not in the instance that this tree corresponds to and is not known by this tree, whatever error would be raised by `instance.__getitem__` will be propagated (usually this is some subclass of LookupError.

**__init__**(*errors: Iterable[*ValidationError*] = ()*)

**__iter__**()

Iterate (non-recursively) over the indices in the instance with errors.

**__len__**()

Return the *total_errors*.

**__repr__**()

Return repr(self).

**__setitem__**(*index: str | int*, *value:* ErrorTree)

Add an error to the tree at the given `index`.

Deprecated since version v4.20.0: Setting items on an *ErrorTree* is deprecated without replacement. To populate a tree, provide all of its sub-errors when you construct the tree.

**property total_errors**

The total number of errors in the entire tree, including children.

Consider the following example:

```
schema = {
    "type" : "array",
    "items" : {"type" : "number", "enum" : [1, 2, 3]},
    "minItems" : 3,
}
instance = ["spam", 2]
```

For clarity's sake, the given instance has three errors under this schema:

```
v = Draft202012Validator(schema)
for error in sorted(v.iter_errors(["spam", 2]), key=str):
    print(error.message)
```

```
'spam' is not of type 'number'
'spam' is not one of [1, 2, 3]
['spam', 2] is too short
```

Let's construct an *jsonschema.exceptions.ErrorTree* so that we can query the errors a bit more easily than by just iterating over the error objects.

```
from jsonschema.exceptions import ErrorTree
tree = ErrorTree(v.iter_errors(instance))
```

As you can see, *jsonschema.exceptions.ErrorTree* takes an iterable of *ValidationError*s when constructing a tree so you can directly pass it the return value of a validator's *jsonschema.protocols.Validator.iter_errors* method.

*ErrorTree*s support a number of useful operations. The first one we might want to perform is to check whether a given element in our instance failed validation. We do so using the `in` operator:

```
>>> 0 in tree
True

>>> 1 in tree
False
```

The interpretation here is that the 0th index into the instance (`"spam"`) did have an error (in fact it had 2), while the 1th index (`2`) did not (i.e. it was valid).

If we want to see which errors a child had, we index into the tree and look at the *ErrorTree.errors* attribute.

```
>>> sorted(tree[0].errors)
['enum', 'type']
```

Here we see that the enum and type keywords failed for index `0`. In fact *ErrorTree.errors* is a dict, whose values are the *ValidationError*s, so we can get at those directly if we want them.

```
>>> print(tree[0].errors["type"].message)
'spam' is not of type 'number'
```

Of course this means that if we want to know if a given validation keyword failed for a given index, we check for its presence in *ErrorTree.errors*:

```
>>> "enum" in tree[0].errors
True

>>> "minimum" in tree[0].errors
False
```

Finally, if you were paying close enough attention, you'll notice that we haven't seen our minItems error appear anywhere yet. This is because minItems is an error that applies globally to the instance itself. So it appears in the root node of the tree.

```
>>> "minItems" in tree.errors
True
```

That's all you need to know to use error trees.

To summarize, each tree contains child trees that can be accessed by indexing the tree to get the corresponding child tree for a given index into the instance. Each tree and child has a *ErrorTree.errors* attribute, a dict, that maps the failed validation keyword to the corresponding validation error.

### 7.2.2 best_match and relevance

The *best_match* function is a simple but useful function for attempting to guess the most relevant error in a given bunch.

```
>>> from jsonschema import Draft202012Validator
>>> from jsonschema.exceptions import best_match

>>> schema = {
...     "type": "array",
...     "minItems": 3,
... }
>>> print(best_match(Draft202012Validator(schema).iter_errors(11)).message)
11 is not of type 'array'
```

jsonschema.exceptions.**best_match**(*errors*, *key=<function by_relevance.<locals>.relevance>*)

Try to find an error that appears to be the best match among given errors.

In general, errors that are higher up in the instance (i.e. for which *ValidationError.path* is shorter) are considered better matches, since they indicate "more" is wrong with the instance.

If the resulting match is either oneOf or anyOf, the *opposite* assumption is made – i.e. the deepest error is picked, since these keywords only need to match once, and any other errors may not be relevant.

**Parameters**

- **errors** (*collections.abc.Iterable*) – the errors to select from. Do not provide a mixture of errors from different validation attempts (i.e. from different instances or schemas), since it won't produce sensical output.
- **key** (*collections.abc.Callable*) – the key to use when sorting errors. See *relevance* and transitively *by_relevance* for more details (the default is to sort with the defaults of that function). Changing the default is only useful if you want to change the function that rates errors but still want the error context descent done by this function.

**Returns**

the best matching error, or `None` if the iterable was empty

---

**Note:** This function is a heuristic. Its return value may change for a given set of inputs from version to version if better heuristics are added.

---

jsonschema.exceptions.**relevance**(*validation_error*)

A key function that sorts errors based on heuristic relevance.

If you want to sort a bunch of errors entirely, you can use this function to do so. Using this function as a key to e.g. `sorted` or `max` will cause more relevant errors to be considered greater than less relevant ones.

Within the different validation keywords that can fail, this function considers anyOf and oneOf to be *weak* validation errors, and will sort them lower than other errors at the same level in the instance.

If you want to change the set of weak [or strong] validation keywords you can create a custom version of this function with *by_relevance* and provide a different set of each.

```
>>> schema = {
...     "properties": {
...         "name": {"type": "string"},
...         "phones": {
...             "properties": {
...                 "home": {"type": "string"}
...             },
...         },
...     },
... }
>>> instance = {"name": 123, "phones": {"home": [123]}}
>>> errors = Draft202012Validator(schema).iter_errors(instance)
>>> [
...     e.path[-1]
...     for e in sorted(errors, key=exceptions.relevance)
... ]
['home', 'name']
```

jsonschema.exceptions.**by_relevance**(*weak=frozenset({'anyOf', 'oneOf'})*, *strong=frozenset({})*)

Create a key function that can be used to sort errors by relevance.

**Parameters**

- **weak** (*set*) – a collection of validation keywords to consider to be "weak". If there are two errors at the same level of the instance and one is in the set of weak validation keywords, the other error will take priority. By default, anyOf and oneOf are considered weak keywords and will be superseded by other same-level validation errors.
- **strong** (*set*) – a collection of validation keywords to consider to be "strong"

## 7.3 JSON (Schema) Referencing

The JSON Schema $ref and $dynamicRef keywords allow schema authors to combine multiple schemas (or sub-schemas) together for reuse or deduplication.

The `referencing` library was written in order to provide a simple, well-behaved and well-tested implementation of this kind of reference resolution[1]. It has its `own documentation which is worth reviewing`, but this page serves as an introduction which is tailored specifically to JSON Schema, and even more specifically to how to configure `referencing` for use with *`Validator`* objects in order to customize the behavior of the $ref keyword and friends in your schemas.

Configuring *`jsonschema`* for custom referencing behavior is essentially a two step process:

- Create a `referencing.Registry` object that behaves the way you wish
- Pass the `referencing.Registry` to your *`Validator`* when instantiating it

The examples below essentially follow these two steps.

### 7.3.1 Introduction to the `referencing` API

There are 3 main objects to be aware of in the *JSON (Schema) Referencing* API:

- `referencing.Registry`, which represents a specific immutable set of JSON Schemas (either in-memory or retrievable)
- `referencing.Specification`, which represents a specific *version* of the JSON Schema specification, which can have differing referencing behavior. JSON Schema-specific specifications live in the `referencing.jsonschema` module and are named like `referencing.jsonschema.DRAFT202012`.
- `referencing.Resource`, which represents a specific JSON Schema (often a Python `dict`) *along* with a specific `referencing.Specification` it is to be interpreted under.

As a concrete example, the simple schema `{"type": "integer"}` may be interpreted as a schema under either Draft 2020-12 or Draft 4 of the JSON Schema specification (amongst others); in draft 2020-12, the float `2.0` must be considered an integer, whereas in draft 4, it potentially is not. If you mean the former (i.e. to associate this schema with draft 2020-12), you'd use `referencing.Resource(contents={"type": "integer"}, specification=referencing.jsonschema.DRAFT202012)`, whereas for the latter you'd use `referencing.jsonschema.DRAFT4`.

**See also:**

the JSON Schema $schema keyword

Which should generally be used to remove all ambiguity and identify *internally* to the schema what version it is written for.

A schema may be identified via one or more URIs, either because they contain an $id keyword (in suitable versions of the JSON Schema specification) which indicates their canonical URI, or simply because you wish to externally associate a URI with the schema, regardless of whether it contains an `$id` keyword. You could add the aforementioned simple schema to a `referencing.Registry` by creating an empty registry and then identifying it via some URI:

```
from referencing import Registry, Resource
from referencing.jsonschema import DRAFT202012
schema = Resource(contents={"type": "integer"}, specification=DRAFT202012)
registry = Registry().with_resource(uri="http://example.com/my/schema", resource=schema)
print(registry)
```

[1] One that in fact is independent of this *`jsonschema`* library itself, and may some day be used by other tools or implementations.

```
<Registry (1 uncrawled resource)>
```

**Note:** `referencing.Registry` is an entirely immutable object. All of its methods which add schemas (resources) to itself return *new* registry objects containing the added schemas.

You could also confirm your schema is in the registry if you'd like, via `referencing.Registry.contents`, which will show you the contents of a resource at a given URI:

```
print(registry.contents("http://example.com/my/schema"))
```

```
{'type': 'integer'}
```

For further details, see the `referencing documentation`.

### 7.3.2 Common Scenarios

#### Making Additional In-Memory Schemas Available

The most common scenario one is likely to encounter is the desire to include a small number of additional in-memory schemas, making them available for use during validation.

For instance, imagine the below schema for non-negative integers:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "integer",
  "minimum": 0
}
```

We may wish to have other schemas we write be able to make use of this schema, and refer to it as `http://example.com/nonneg-int-schema` and/or as `urn:nonneg-integer-schema`.

To do so we make use of APIs from the referencing library to create a `referencing.Registry` which maps the URIs above to this schema:

```
from referencing import Registry, Resource
schema = Resource.from_contents(
    {
        "$schema": "https://json-schema.org/draft/2020-12/schema",
        "type": "integer",
        "minimum": 0,
    },
)
registry = Registry().with_resources(
    [
        ("http://example.com/nonneg-int-schema", schema),
        ("urn:nonneg-integer-schema", schema),
    ],
)
```

What's above is likely mostly self-explanatory, other than the presence of the `referencing.Resource.from_contents` function. Its purpose is to convert a piece of "opaque" JSON (or really a Python `dict` containing

deserialized JSON) into an object which indicates what *version* of JSON Schema the schema is meant to be interpreted under. Calling it will inspect a $schema keyword present in the given schema and use that to associate the JSON with an appropriate `specification`. If your schemas do not contain `$schema` dialect identifiers, and you intend for them to be interpreted always under a specific dialect – say Draft 2020-12 of JSON Schema – you may instead use e.g.:

```
from referencing import Registry, Resource
from referencing.jsonschema import DRAFT202012
schema = DRAFT202012.create_resource({"type": "integer", "minimum": 0})
registry = Registry().with_resources(
    [
        ("http://example.com/nonneg-int-schema", schema),
        ("urn:nonneg-integer-schema", schema),
    ],
)
```

which has the same functional effect.

You can now pass this registry to your *Validator*, which allows a schema passed to it to make use of the aforementioned URIs to refer to our non-negative integer schema. Here for instance is an example which validates that instances are JSON objects with non-negative integral values:

```
from jsonschema import Draft202012Validator
validator = Draft202012Validator(
    {
        "type": "object",
        "additionalProperties": {"$ref": "urn:nonneg-integer-schema"},
    },
    registry=registry,  # the critical argument, our registry from above
)
validator.validate({"foo": 37})
assert not validator.is_valid({"foo": -37})  # Uh oh!
```

### Resolving References from the File System

Another common request from schema authors is to be able to map URIs to the file system, perhaps while developing a set of schemas in different local files. If you have a set of *fixed* or *static* schemas in a few files, you still likely will want to follow the *above in-memory instructions*, and simply load all of your files by reading them in-memory from your program. If however you wish to *dynamically* read files off of the file system, perhaps because they may change during the lifetime of your process, then the referencing library supports doing so fully dynamically by configuring a callable which can be used to retrieve any schema which is *not* already pre-loaded in-memory.

Here we resolve any schema beginning with `http://localhost` to a directory `/tmp/schemas` on the local filesystem (note of course that this will not work if run directly unless you have populated that directory with some schemas):

```
from pathlib import Path
import json

from referencing import Registry, Resource
from referencing.exceptions import NoSuchResource

SCHEMAS = Path("/tmp/schemas")

def retrieve_from_filesystem(uri: str):
    if not uri.startswith("http://localhost/"):
```

(continues on next page)

(continued from previous page)

```
        raise NoSuchResource(ref=uri)
    path = SCHEMAS / Path(uri.removeprefix("http://localhost/"))
    contents = json.loads(path.read_text())
    return Resource.from_contents(contents)

registry = Registry(retrieve=retrieve_from_filesystem)
```

Such a registry can then be used with *Validator* objects in the same way shown above, and any such references to URIs which are not already in-memory will be retrieved from the configured directory.

We can mix the two examples above if we wish for some in-memory schemas to be available in addition to the filesystem schemas, e.g.:

```
from referencing.jsonschema import DRAFT7
registry = Registry(retrieve=retrieve_from_filesystem).with_resource(
    "urn:non-empty-array", DRAFT7.create_resource({"type": "array", "minItems": 1}),
)
```

where we've made use of the similar `referencing.Registry.with_resource` function to add a single additional resource.

### Resolving References to Schemas Written in YAML

Generalizing slightly, the retrieval function provided need not even assume that it is retrieving JSON. As long as you deserialize what you have retrieved into Python objects, you may equally be retrieving references to YAML documents or any other format.

Here for instance we retrieve YAML documents in a way similar to the *above* using PyYAML:

```
from pathlib import Path
import yaml

from referencing import Registry, Resource
from referencing.exceptions import NoSuchResource

SCHEMAS = Path("/tmp/yaml-schemas")

def retrieve_yaml(uri: str):
    if not uri.startswith("http://localhost/"):
        raise NoSuchResource(ref=uri)
    path = SCHEMAS / Path(uri.removeprefix("http://localhost/"))
    contents = yaml.safe_load(path.read_text())
    return Resource.from_contents(contents)

registry = Registry(retrieve=retrieve_yaml)
```

---

**Note:** Not all YAML fits within the JSON data model.

JSON Schema is defined specifically for JSON, and has well-defined behavior strictly for Python objects which could have possibly existed as JSON.

If you stick to the subset of YAML for which this is the case then you shouldn't have issue, but if you pass schemas (or instances) around whose structure could never have possibly existed as JSON (e.g. a mapping whose keys are not

strings), all bets are off.

One could similarly imagine a retrieval function which switches on whether to call `yaml.safe_load` or `json.loads` by file extension (or some more reliable mechanism) and thereby support retrieving references of various different file formats.

### Automatically Retrieving Resources Over HTTP

In the general case, the JSON Schema specifications tend to discourage implementations (like this one) from automatically retrieving references over the network, or even assuming such a thing is feasible (as schemas may be identified by URIs which are strictly identifiers, and not necessarily downloadable from the URI even when such a thing is sensical).

However, if you as a schema author are in a situation where you indeed do wish to do so for convenience (and understand the implications of doing so), you may do so by making use of the `retrieve` argument to `referencing.Registry`.

Here is how one would configure a registry to automatically retrieve schemas from the JSON Schema Store on the fly using the httpx:

```
from referencing import Registry, Resource
import httpx

def retrieve_via_httpx(uri: str):
    response = httpx.get(uri)
    return Resource.from_contents(response.json())

registry = Registry(retrieve=retrieve_via_httpx)
```

Given such a registry, we can now, for instance, validate instances against schemas from the schema store by passing the `registry` we configured to our *`Validator`* as in previous examples:

```
from jsonschema import Draft202012Validator
Draft202012Validator(
    {"$ref": "https://json.schemastore.org/pyproject.json"},
    registry=registry,
).validate({"project": {"name": 12}})
```

which should in this case indicate the example data is invalid:

```
Traceback (most recent call last):
    ...
jsonschema.exceptions.ValidationError: 12 is not of type 'string'

Failed validating 'type' in schema['properties']['project']['properties']['name']:
    {'pattern': '^([a-zA-Z\\d]|[a-zA-Z\\d][\\w.-]*[a-zA-Z\\d])$',
    'title': 'Project name',
    'type': 'string'}

On instance['project']['name']:
    12
```

Retrieving resources from a SQLite database or some other network-accessible resource should be more or less similar, replacing the HTTP client with one for your database of course.

**Warning:** Be sure you understand the security implications of the reference resolution you configure. And if you accept untrusted schemas, doubly sure!

You wouldn't want a user causing your machine to go off and retrieve giant files off the network by passing it a `$ref` to some huge blob, or exploiting similar vulnerabilities in your setup.

### 7.3.3 Migrating From `RefResolver`

Older versions of *jsonschema* used a different object – *_RefResolver* – for reference resolution, which you a schema author may already be configuring for your own use.

*_RefResolver* is now fully deprecated and replaced by the use of `referencing.Registry` as shown in examples above.

If you are not already constructing your own *_RefResolver*, this change should be transparent to you (or even recognizably improved, as the point of the migration was to improve the quality of the referencing implementation and enable some new functionality).

Table 1: Rough equivalence between `_RefResolver` and `referencing.Registry` APIs

| Old API | New API |
|---|---|
| `RefResolver.from_schema({"$id": "urn:example:foo", ...}` | `Registry().with_resource(uri="urn:example:foo", resource=Resource.from_contents({"$id": "urn:example:foo", ...}))` |
| Overriding `RefResolver.resolve_from_url` | Passing a callable to `referencing.Registry`'s `retrieve` argument |
| `DraftNValidator(..., resolver=_RefResolver(... )) `` | DraftNValidator(..., registry=Registry().with_resources(...))`` |

Here are some more specifics on how to migrate to the newer APIs:

#### The `store` argument

*_RefResolver*'s `store` argument was essentially the equivalent of `referencing.Registry`'s in-memory schema storage.

If you currently pass a set of schemas via e.g.:

```
from jsonschema import Draft202012Validator, RefResolver
resolver = RefResolver.from_schema(
    schema={"title": "my schema"},
    store={"http://example.com": {"type": "integer"}},
)
validator = Draft202012Validator(
    {"$ref": "http://example.com"},
    resolver=resolver,
)
validator.validate("foo")
```

you should be able to simply move to something like:

```
from referencing import Registry
from referencing.jsonschema import DRAFT202012

from jsonschema import Draft202012Validator

registry = Registry().with_resource(
    "http://example.com",
    DRAFT202012.create_resource({"type": "integer"}),
)
validator = Draft202012Validator(
    {"$ref": "http://example.com"},
    registry=registry,
)
assert not validator.is_valid("foo")
```

#### Handlers

The `handlers` functionality from *`_RefResolver`* was a way to support additional HTTP schemes for schema retrieval.

Here you should move to a custom `retrieve` function which does whatever you'd like. E.g. in pseudocode:

```
from urllib.parse import urlsplit

def retrieve(uri: str):
    parsed = urlsplit(uri)
    if parsed.scheme == "file":
        ...
    elif parsed.scheme == "custom":
        ...

registry = Registry(retrieve=retrieve)
```

#### Other Key Functional Differences

Whilst *`_RefResolver`* *did* automatically retrieve remote references (against the recommendation of the spec, and in a way which therefore could lead to questionable security concerns when combined with untrusted schemas), `referencing.Registry` does *not* do so. If you rely on this behavior, you should follow the *above example of retrieving resources over HTTP*.

## 7.4 Creating or Extending Validator Classes

jsonschema.validators.**create**(*meta_schema: ~collections.abc.Mapping[str, ~typing.Any], validators: ~collections.abc.Mapping[str, ~jsonschema._typing.SchemaKeywordValidator] | ~collections.abc.Iterable[tuple[str, ~jsonschema._typing.SchemaKeywordValidator]] = (), version: str | None = None, type_checker: ~jsonschema._types.TypeChecker = <TypeChecker types={'array', 'boolean', 'integer', 'null', 'number', 'object', 'string'}>, format_checker: ~jsonschema._format.FormatChecker = <FormatChecker checkers=['date', 'email', 'idn-email', 'idn-hostname', 'ipv4', 'ipv6', 'regex', 'uuid']>, id_of: ~typing.Callable[[bool | ~collections.abc.Mapping[str, ~typing.Any]], str | None] = <function _dollar_id>, applicable_validators: ~typing.Callable[[bool | ~collections.abc.Mapping[str, ~typing.Any]], ~typing.Iterable[~typing.Tuple[str, ~typing.Any]]] = operator.methodcaller('items')*)

Create a new validator class.

**Parameters**

- **meta_schema** – the meta schema for the new validator class
- **validators** – a mapping from names to callables, where each callable will validate the schema property with the given name.

  Each callable should take 4 arguments:

  1. a validator instance,
  2. the value of the property being validated within the instance
  3. the instance
  4. the schema
- **version** – an identifier for the version that this validator class will validate. If provided, the returned validator class will have its `__name__` set to include the version, and also will have *jsonschema.validators.validates* automatically called for the given version.
- **type_checker** – a type checker, used when applying the type keyword.

  If unprovided, a *jsonschema.TypeChecker* will be created with a set of default types typical of JSON Schema drafts.
- **format_checker** – a format checker, used when applying the format keyword.

  If unprovided, a *jsonschema.FormatChecker* will be created with a set of default formats typical of JSON Schema drafts.
- **id_of** – A function that given a schema, returns its ID.
- **applicable_validators** – A function that, given a schema, returns the list of applicable schema keywords and associated values which will be used to validate the instance. This is mostly used to support pre-draft 7 versions of JSON Schema which specified behavior around ignoring keywords if they were siblings of a `$ref` keyword. If you're not attempting to implement similar behavior, you can typically ignore this argument and leave it at its default.

**Returns**

a new *jsonschema.protocols.Validator* class

jsonschema.validators.**extend**(*validator*, *validators=()*, *version=None*, *type_checker=None*, *format_checker=None*)

Create a new validator class by extending an existing one.

**Parameters**

- **validator** (*jsonschema.protocols.Validator*) – an existing validator class
- **validators** (*collections.abc.Mapping*) – a mapping of new validator callables to extend with, whose structure is as in *create*.

  **Note:** Any validator callables with the same name as an existing one will (silently) replace the old validator callable entirely, effectively overriding any validation done in the "parent" validator class.

  If you wish to instead extend the behavior of a parent's validator callable, delegate and call it directly in the new validator function by retrieving it using `OldValidator.VALIDATORS["validation_keyword_name"]`.

- **version** (*str*) – a version for the new validator class
- **type_checker** (*jsonschema.TypeChecker*) – a type checker, used when applying the type keyword.

  If unprovided, the type checker of the extended *jsonschema.protocols.Validator* will be carried along.

- **format_checker** (*jsonschema.FormatChecker*) – a format checker, used when applying the format keyword.

  If unprovided, the format checker of the extended *jsonschema.protocols.Validator* will be carried along.

**Returns**

a new *jsonschema.protocols.Validator* class extending the one provided

**Note:** Meta Schemas

The new validator class will have its parent's meta schema.

If you wish to change or extend the meta schema in the new validator class, modify `META_SCHEMA` directly on the returned class. Note that no implicit copying is done, so a copy should likely be made before modifying it, in order to not affect the old validator.

jsonschema.validators.**validator_for**(*schema*, *default: Validator | _utils.Unset = <unset>*) → type[*Validator*]

Retrieve the validator class appropriate for validating the given schema.

Uses the $schema keyword that should be present in the given schema to look up the appropriate validator class.

**Parameters**

- **schema** (*collections.abc.Mapping or bool*) – the schema to look at
- **default** – the default to return if the appropriate validator class cannot be determined.

  If unprovided, the default is to return the latest supported draft.

#### Examples

The $schema JSON Schema keyword will control which validator class is returned:

```
>>> schema = {
...     "$schema": "https://json-schema.org/draft/2020-12/schema",
...     "type": "integer",
... }
>>> jsonschema.validators.validator_for(schema)
<class 'jsonschema.validators.Draft202012Validator'>
```

Here, a draft 7 schema instead will return the draft 7 validator:

```
>>> schema = {
...     "$schema": "http://json-schema.org/draft-07/schema#",
...     "type": "integer",
... }
>>> jsonschema.validators.validator_for(schema)
<class 'jsonschema.validators.Draft7Validator'>
```

Schemas with no `$schema` keyword will fallback to the default argument:

```
>>> schema = {"type": "integer"}
>>> jsonschema.validators.validator_for(
...     schema, default=Draft7Validator,
... )
<class 'jsonschema.validators.Draft7Validator'>
```

or if none is provided, to the latest version supported. Always including the keyword when authoring schemas is highly recommended.

jsonschema.validators.**validates**(*version*)

Register the decorated validator for a `version` of the specification.

Registered validators and their meta schemas will be considered when parsing $schema keywords' URIs.

**Parameters**
**version** (*str*) – An identifier to use as the version's name

**Returns**
a class decorator to decorate the validator with the version

**Return type**
collections.abc.Callable

### 7.4.1 Creating Validation Errors

Any validating function that validates against a subschema should call `descend`, rather than `iter_errors`. If it recurses into the instance, or schema, it should pass one or both of the `path` or `schema_path` arguments to `descend` in order to properly maintain where in the instance or schema respectively the error occurred.

### 7.4.2 The Validator Protocol

`jsonschema` defines a `protocol`, *`jsonschema.protocols.Validator`* which can be used in type annotations to describe the type of a validator.

For full details, see *The Validator Protocol*.

## 7.5 Frequently Asked Questions

### 7.5.1 My schema specifies format validation. Why do invalid instances seem valid?

The format keyword can be a bit of a stumbling block for new users working with JSON Schema.

In a schema such as:

```
{"type": "string", "format": "date"}
```

JSON Schema specifications have historically differentiated between the format keyword and other keywords. In particular, the format keyword was specified to be *informational* as much as it may be used for validation.

In other words, for many use cases, schema authors may wish to use values for the format keyword but have no expectation they be validated alongside other required assertions in a schema.

Of course this does not represent all or even most use cases – many schema authors *do* wish to assert that instances conform fully, even to the specific format mentioned.

In drafts prior to `draft2019-09`, the decision on whether to automatically enable format validation was left up to validation implementations such as this one.

This library made the choice to leave it off by default, for two reasons:

- for forward compatibility and implementation complexity reasons – if format validation were on by default, and a future draft of JSON Schema introduced a hard-to-implement format, either the implementation of that format would block releases of this library until it were implemented, or the behavior surrounding format would need to be even more complex than simply defaulting to be on. It therefore was safer to start with it off, and defend against the expectation that a given format would always automatically work.
- given that a common use of JSON Schema is for portability across languages (and therefore implementations of JSON Schema), so that users be aware of this point itself regarding format validation, and therefore remember to check any *other* implementations they were using to ensure they too were explicitly enabled for format validation.

As of `draft2019-09` however, the opt-out by default behavior mentioned here is now *required* for all implementations of JSON Schema.

Difficult as this may sound for new users, at this point it at least means they should expect the same behavior that has always been implemented here, across any other implementation they encounter.

**See also:**

Draft 2019-09's release notes on format

> for upstream details on the behavior of format and how it has changed in `draft2019-09`

*Validating Formats*

> for details on how to enable format validation

*`jsonschema.FormatChecker`*

> the object which implements format validation

### 7.5.2 Can jsonschema be used to validate YAML, TOML, etc.?

Like most JSON Schema implementations, *jsonschema* doesn't actually deal directly with JSON at all (other than in relation to the $ref keyword, elaborated on below).

In other words as far as this library is concerned, schemas and instances are simply runtime Python objects. The JSON object `{}` is simply the Python `dict` `{}`, and a JSON Schema like `{"type": "object", {"properties": {}}}` is really an assertion about particular Python objects and their keys.

---

**Note:** The $ref keyword is a single notable exception.

Specifically, in the case where *jsonschema* is asked to resolve a remote reference, it has no choice but to assume that the remote reference is serialized as JSON, and to deserialize it using the `json` module.

One cannot today therefore reference some remote piece of YAML and have it deserialized into Python objects by this library without doing some additional work. See *Resolving References to Schemas Written in YAML* for details.

---

In practice what this means for JSON-like formats like YAML and TOML is that indeed one can generally schematize and then validate them exactly as if they were JSON by simply first deserializing them using libraries like `PyYAML` or the like, and passing the resulting Python objects into functions within this library.

Beware however that there are cases where the behavior of the JSON Schema specification itself is only well-defined within the data model of JSON itself, and therefore only for Python objects that could have "in theory" come from JSON. As an example, JSON supports only string-valued keys, whereas YAML supports additional types. The JSON Schema specification does not deal with how to apply the patternProperties keyword to non-string properties. The behavior of this library is therefore similarly not defined when presented with Python objects of this form, which could never have come from JSON. In such cases one is recommended to first pre-process the data such that the resulting behavior is well-defined. In the previous example, if the desired behavior is to transparently coerce numeric properties to strings, as Javascript might, then do the conversion explicitly before passing data to this library.

### 7.5.3 Why doesn't my schema's default property set the default on my instance?

The basic answer is that the specification does not require that default actually do anything.

For an inkling as to *why* it doesn't actually do anything, consider that none of the other keywords modify the instance either. More importantly, having default modify the instance can produce quite peculiar things. It's perfectly valid (and perhaps even useful) to have a default that is not valid under the schema it lives in! So an instance modified by the default would pass validation the first time, but fail the second!

Still, filling in defaults is a thing that is useful. *jsonschema* allows you to *define your own validator classes and callables*, so you can easily create an *jsonschema.protocols.Validator* that does do default setting. Here's some code to get you started. (In this code, we add the default properties to each object *before* the properties are validated, so the default values themselves will need to be valid under the schema.)

```
from jsonschema import Draft202012Validator, validators


def extend_with_default(validator_class):
    validate_properties = validator_class.VALIDATORS["properties"]

    def set_defaults(validator, properties, instance, schema):
        for property, subschema in properties.items():
            if "default" in subschema:
                instance.setdefault(property, subschema["default"])
```

(continues on next page)

(continued from previous page)

```
        for error in validate_properties(
            validator, properties, instance, schema,
        ):
            yield error

    return validators.extend(
        validator_class, {"properties" : set_defaults},
    )


DefaultValidatingValidator = extend_with_default(Draft202012Validator)


# Example usage:
obj = {}
schema = {'properties': {'foo': {'default': 'bar'}}}
# Note jsonschema.validate(obj, schema, cls=DefaultValidatingValidator)
# will not work because the metaschema contains `default` keywords.
DefaultValidatingValidator(schema).validate(obj)
assert obj == {'foo': 'bar'}
```

See the above-linked document for more info on how this works, but basically, it just extends the properties keyword on a *jsonschema.validators.Draft202012Validator* to then go ahead and update all the defaults.

---

**Note:** If you're interested in a more interesting solution to a larger class of these types of transformations, keep an eye on Seep, which is an experimental data transformation and extraction library written on top of *jsonschema*.

---

**Hint:** The above code can provide default values for an entire object and all of its properties, but only if your schema provides a default value for the object itself, like so:

```
schema = {
    "type": "object",
    "properties": {
        "outer-object": {
            "type": "object",
            "properties" : {
                "inner-object": {
                    "type": "string",
                    "default": "INNER-DEFAULT"
                }
            },
            "default": {} # <-- MUST PROVIDE DEFAULT OBJECT
        }
    }
}

obj = {}
DefaultValidatingValidator(schema).validate(obj)
assert obj == {'outer-object': {'inner-object': 'INNER-DEFAULT'}}
```

...but if you don't provide a default value for your object, then it won't be instantiated at all, much less populated with default properties.

```
del schema["properties"]["outer-object"]["default"]
obj2 = {}
DefaultValidatingValidator(schema).validate(obj2)
assert obj2 == {} # whoops
```

### 7.5.4 How do jsonschema version numbers work?

`jsonschema` tries to follow the Semantic Versioning specification.

This means broadly that no backwards-incompatible changes should be made in minor releases (and certainly not in dot releases).

The full picture requires defining what constitutes a backwards-incompatible change.

The following are simple examples of things considered public API, and therefore should *not* be changed without bumping a major version number:

- module names and contents, when not marked private by Python convention (a single leading underscore)
- function and object signature (parameter order and name)

The following are *not* considered public API and may change without notice:

- the exact wording and contents of error messages; typical reasons to rely on this seem to involve downstream tests in packages using *`jsonschema`*. These use cases are encouraged to use the extensive introspection provided in *`jsonschema.exceptions.ValidationError`*s instead to make meaningful assertions about what failed rather than relying on *how* what failed is explained to a human.
- the order in which validation errors are returned or raised
- the contents of the `jsonschema.tests` package
- the contents of the `jsonschema.benchmarks` package
- the specific non-zero error codes presented by the command line interface
- the exact representation of errors presented by the command line interface, other than that errors represented by the plain outputter will be reported one per line
- anything marked private

With the exception of the last two of those, flippant changes are avoided, but changes can and will be made if there is improvement to be had. Feel free to open an issue ticket if there is a specific issue or question worth raising.

## 7.6 API Reference

### 7.6.1 Submodules

#### jsonschema.validators

Creation and extension of validators, with implementations for existing drafts.

class jsonschema.validators.**Draft201909Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**FORMAT_CHECKER** = <FormatChecker checkers=['date', 'email', 'idn-email', 'idn-hostname', 'ipv4', 'ipv6', 'regex', 'uuid']>

static **ID_OF**(*contents: bool | Mapping[str, Any]*) → str | None

**META_SCHEMA** = {'$id': 'https://json-schema.org/draft/2019-09/schema', '$recursiveAnchor': True, '$schema': 'https://json-schema.org/draft/2019-09/schema', '$vocabulary': {'https://json-schema.org/draft/2019-09/vocab/applicator': True, 'https://json-schema.org/draft/2019-09/vocab/content': True, 'https://json-schema.org/draft/2019-09/vocab/core': True, 'https://json-schema.org/draft/2019-09/vocab/format': False, 'https://json-schema.org/draft/2019-09/vocab/meta-data': True, 'https://json-schema.org/draft/2019-09/vocab/validation': True}, 'allOf': [{'$ref': 'meta/core'}, {'$ref': 'meta/applicator'}, {'$ref': 'meta/validation'}, {'$ref': 'meta/meta-data'}, {'$ref': 'meta/format'}, {'$ref': 'meta/content'}], 'properties': {'definitions': {'$comment': 'While no longer an official keyword as it is replaced by $defs, this keyword is retained in the meta-schema to prevent incompatible extensions as it remains in common use.', 'additionalProperties': {'$recursiveRef': '#'}, 'default': {}, 'type': 'object'}, 'dependencies': {'$comment': '"dependencies" is no longer a keyword, but schema authors should avoid redefining it to facilitate a smooth transition to "dependentSchemas" and "dependentRequired"', 'additionalProperties': {'anyOf': [{'$recursiveRef': '#'}, {'$ref': 'meta/validation#/$defs/stringArray'}]}, 'type': 'object'}}, 'title': 'Core and Validation specifications meta-schema', 'type': ['object', 'boolean']}

**TYPE_CHECKER** = <TypeChecker types={'array', 'boolean', 'integer', 'null', 'number', 'object', 'string'}>

**VALIDATORS** = {'$recursiveRef': <function recursiveRef>, '$ref': <function ref>, 'additionalItems': <function additionalItems>, 'additionalProperties': <function additionalProperties>, 'allOf': <function allOf>, 'anyOf': <function anyOf>, 'const': <function const>, 'contains': <function contains>, 'dependentRequired': <function dependentRequired>, 'dependentSchemas': <function dependentSchemas>, 'enum': <function enum>, 'exclusiveMaximum': <function exclusiveMaximum>, 'exclusiveMinimum': <function exclusiveMinimum>, 'format': <function format>, 'if': <function if_>, 'items': <function items_draft6_draft7_draft201909>, 'maxItems': <function maxItems>, 'maxLength': <function maxLength>, 'maxProperties': <function maxProperties>, 'maximum': <function maximum>, 'minItems': <function minItems>, 'minLength': <function minLength>, 'minProperties': <function minProperties>, 'minimum': <function minimum>, 'multipleOf': <function multipleOf>, 'not': <function not_>, 'oneOf': <function oneOf>, 'pattern': <function pattern>, 'patternProperties': <function patternProperties>, 'properties': <function properties>, 'propertyNames': <function propertyNames>, 'required': <function required>, 'type': <function type>, 'unevaluatedItems': <function unevaluatedItems_draft2019>, 'unevaluatedProperties': <function unevaluatedProperties_draft2019>, 'uniqueItems': <function uniqueItems>}

**classmethod check_schema**(*schema*, *format_checker=<unset>*)

**descend**(*instance*, *schema*, *path=None*, *schema_path=None*, *resolver=None*)

**evolve**(***changes*)

**format_checker**: *FormatChecker* | None

**is_type**(*instance*, *type*)

**is_valid**(*instance*, *_schema=None*)

**iter_errors**(*instance*, *_schema=None*)

**property resolver**

**schema**: bool | Mapping[str, Any]

**validate**(**args*, ***kwargs*)

class jsonschema.validators.**Draft202012Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**FORMAT_CHECKER** = <FormatChecker checkers=['date', 'email', 'idn-email', 'idn-hostname', 'ipv4', 'ipv6', 'regex', 'uuid']>

**static ID_OF**(*contents: bool | Mapping[str, Any]*) → str | None

**META_SCHEMA** = {'$comment': 'This meta-schema also defines keywords that have appeared in previous drafts in order to prevent incompatible extensions as they remain in common use.', '$dynamicAnchor': 'meta', '$id': 'https://json-schema.org/draft/2020-12/schema', '$schema': 'https://json-schema.org/draft/2020-12/schema', '$vocabulary': {'https://json-schema.org/draft/2020-12/vocab/applicator': True, 'https://json-schema.org/draft/2020-12/vocab/content': True, 'https://json-schema.org/draft/2020-12/vocab/core': True, 'https://json-schema.org/draft/2020-12/vocab/format-annotation': True, 'https://json-schema.org/draft/2020-12/vocab/meta-data': True, 'https://json-schema.org/draft/2020-12/vocab/unevaluated': True, 'https://json-schema.org/draft/2020-12/vocab/validation': True}, 'allOf': [{'$ref': 'meta/core'}, {'$ref': 'meta/applicator'}, {'$ref': 'meta/unevaluated'}, {'$ref': 'meta/validation'}, {'$ref': 'meta/meta-data'}, {'$ref': 'meta/format-annotation'}, {'$ref': 'meta/content'}], 'properties': {'$recursiveAnchor': {'$comment': '"$recursiveAnchor" has been replaced by "$dynamicAnchor".', '$ref': 'meta/core#/$defs/anchorString', 'deprecated': True}, '$recursiveRef': {'$comment': '"$recursiveRef" has been replaced by "$dynamicRef".', '$ref': 'meta/core#/$defs/uriReferenceString', 'deprecated': True}, 'definitions': {'$comment': '"definitions" has been replaced by "$defs".', 'additionalProperties': {'$dynamicRef': '#meta'}, 'default': {}, 'deprecated': True, 'type': 'object'}, 'dependencies': {'$comment': '"dependencies" has been split and replaced by "dependentSchemas" and "dependentRequired" in order to serve their differing semantics.', 'additionalProperties': {'anyOf': [{'$dynamicRef': '#meta'}, {'$ref': 'meta/validation#/$defs/stringArray'}]}, 'default': {}, 'deprecated': True, 'type': 'object'}}, 'title': 'Core and Validation specifications meta-schema', 'type': ['object', 'boolean']}

TYPE_CHECKER = <TypeChecker types={'array', 'boolean', 'integer', 'null', 'number', 'object', 'string'}>

VALIDATORS = {'$dynamicRef': <function dynamicRef>, '$ref': <function ref>, 'additionalProperties': <function additionalProperties>, 'allOf': <function allOf>, 'anyOf': <function anyOf>, 'const': <function const>, 'contains': <function contains>, 'dependentRequired': <function dependentRequired>, 'dependentSchemas': <function dependentSchemas>, 'enum': <function enum>, 'exclusiveMaximum': <function exclusiveMaximum>, 'exclusiveMinimum': <function exclusiveMinimum>, 'format': <function format>, 'if': <function if_>, 'items': <function items>, 'maxItems': <function maxItems>, 'maxLength': <function maxLength>, 'maxProperties': <function maxProperties>, 'maximum': <function maximum>, 'minItems': <function minItems>, 'minLength': <function minLength>, 'minProperties': <function minProperties>, 'minimum': <function minimum>, 'multipleOf': <function multipleOf>, 'not': <function not_>, 'oneOf': <function oneOf>, 'pattern': <function pattern>, 'patternProperties': <function patternProperties>, 'prefixItems': <function prefixItems>, 'properties': <function properties>, 'propertyNames': <function propertyNames>, 'required': <function required>, 'type': <function type>, 'unevaluatedItems': <function unevaluatedItems>, 'unevaluatedProperties': <function unevaluatedProperties>, 'uniqueItems': <function uniqueItems>}

classmethod **check_schema**(*schema*, *format_checker=<unset>*)

**descend**(*instance*, *schema*, *path=None*, *schema_path=None*, *resolver=None*)

**evolve**(***changes*)

**format_checker**: *FormatChecker* | None

**is_type**(*instance*, *type*)

**is_valid**(*instance*, *_schema=None*)

**iter_errors**(*instance*, *_schema=None*)

property **resolver**

**schema**: bool | Mapping[str, Any]

**validate**(**args*, ***kwargs*)

class jsonschema.validators.**Draft3Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

FORMAT_CHECKER = <FormatChecker checkers=['date', 'email', 'idn-email', 'ip-address', 'ipv6', 'regex', 'time']>

static **ID_OF**(*contents: Mapping[str, Any]*) → str | None

META_SCHEMA = {'$schema': 'http://json-schema.org/draft-03/schema#', 'default': {}, 'dependencies': {'exclusiveMaximum': 'maximum', 'exclusiveMinimum': 'minimum'}, 'id': 'http://json-schema.org/draft-03/schema#', 'properties': {'$ref': {'type': 'string'}, '$schema': {'format': 'uri', 'type': 'string'}, 'additionalItems': {'default': {}, 'type': [{'$ref': '#'}, 'boolean']}, 'additionalProperties': {'default': {}, 'type': [{'$ref': '#'}, 'boolean']}, 'default': {'type': 'any'}, 'dependencies': {'additionalProperties': {'items': {'type': 'string'}, 'type': ['string', 'array', {'$ref': '#'}]}, 'default': {}, 'type': 'object'}, 'description': {'type': 'string'}, 'disallow': {'items': {'type': ['string', {'$ref': '#'}]}, 'type': ['string', 'array'], 'uniqueItems': True}, 'divisibleBy': {'default': 1, 'exclusiveMinimum': True, 'minimum': 0, 'type': 'number'}, 'enum': {'minItems': 1, 'type': 'array', 'uniqueItems': True}, 'exclusiveMaximum': {'default': False, 'type': 'boolean'}, 'exclusiveMinimum': {'default': False, 'type': 'boolean'}, 'extends': {'default': {}, 'items': {'$ref': '#'}, 'type': [{'$ref': '#'}, 'array']}, 'format': {'type': 'string'}, 'id': {'type': 'string'}, 'items': {'default': {}, 'items': {'$ref': '#'}, 'type': [{'$ref': '#'}, 'array']}, 'maxItems': {'minimum': 0, 'type': 'integer'}, 'maxLength': {'type': 'integer'}, 'maximum': {'type': 'number'}, 'minItems': {'default': 0, 'minimum': 0, 'type': 'integer'}, 'minLength': {'default': 0, 'minimum': 0, 'type': 'integer'}, 'minimum': {'type': 'number'}, 'pattern': {'format': 'regex', 'type': 'string'}, 'patternProperties': {'additionalProperties': {'$ref': '#'}, 'default': {}, 'type': 'object'}, 'properties': {'additionalProperties': {'$ref': '#'}, 'default': {}, 'type': 'object'}, 'required': {'default': False, 'type': 'boolean'}, 'title': {'type': 'string'}, 'type': {'default': 'any', 'items': {'type': ['string', {'$ref': '#'}]}, 'type': ['string', 'array'], 'uniqueItems': True}, 'uniqueItems': {'default': False, 'type': 'boolean'}}, 'type': 'object'}

TYPE_CHECKER = <TypeChecker types={'any', 'array', 'boolean', 'integer', 'null', 'number', 'object', 'string'}>

VALIDATORS = {'$ref': <function ref>, 'additionalItems': <function additionalItems>, 'additionalProperties': <function additionalProperties>, 'dependencies': <function dependencies_draft3>, 'disallow': <function disallow_draft3>, 'divisibleBy': <function multipleOf>, 'enum': <function enum>, 'extends': <function extends_draft3>, 'format': <function format>, 'items': <function items_draft3_draft4>, 'maxItems': <function maxItems>, 'maxLength': <function maxLength>, 'maximum': <function maximum_draft3_draft4>, 'minItems': <function minItems>, 'minLength': <function minLength>, 'minimum': <function minimum_draft3_draft4>, 'pattern': <function pattern>, 'patternProperties': <function patternProperties>, 'properties': <function properties_draft3>, 'type': <function type_draft3>, 'uniqueItems': <function uniqueItems>}

**classmethod check_schema**(*schema*, *format_checker=<unset>*)

**descend**(*instance*, *schema*, *path=None*, *schema_path=None*, *resolver=None*)

**evolve**(***changes*)

**format_checker:** *FormatChecker* | None

**is_type**(*instance*, *type*)

**is_valid**(*instance*, *_schema=None*)

**iter_errors**(*instance*, *_schema=None*)

**property resolver**

**schema: bool | Mapping[str, Any]**

**validate**(**args*, ***kwargs*)

**class** jsonschema.validators.**Draft4Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**FORMAT_CHECKER = <FormatChecker checkers=['email', 'idn-email', 'ipv4', 'ipv6', 'regex']>**

**static ID_OF**(*contents: Mapping[str, Any]*) → str | None

```
META_SCHEMA = {'$schema': 'http://json-schema.org/draft-04/schema#', 'default':
{}, 'definitions': {'positiveInteger': {'minimum': 0, 'type': 'integer'},
'positiveIntegerDefault0': {'allOf': [{'$ref': '#/definitions/positiveInteger'},
{'default': 0}]}, 'schemaArray': {'items': {'$ref': '#'}, 'minItems': 1,
'type': 'array'}, 'simpleTypes': {'enum': ['array', 'boolean', 'integer', 'null',
'number', 'object', 'string']}, 'stringArray': {'items': {'type': 'string'},
'minItems': 1, 'type': 'array', 'uniqueItems': True}}, 'dependencies':
{'exclusiveMaximum': ['maximum'], 'exclusiveMinimum': ['minimum']}, 'description':
'Core schema meta-schema', 'id': 'http://json-schema.org/draft-04/schema#',
'properties': {'$schema': {'type': 'string'}, 'additionalItems': {'anyOf':
[{'type': 'boolean'}, {'$ref': '#'}], 'default': {}}, 'additionalProperties':
{'anyOf': [{'type': 'boolean'}, {'$ref': '#'}], 'default': {}}, 'allOf':
{'$ref': '#/definitions/schemaArray'}, 'anyOf': {'$ref':
'#/definitions/schemaArray'}, 'default': {}, 'definitions':
{'additionalProperties': {'$ref': '#'}, 'default': {}, 'type': 'object'},
'dependencies': {'additionalProperties': {'anyOf': [{'$ref': '#'}, {'$ref':
'#/definitions/stringArray'}]}, 'type': 'object'}, 'description': {'type':
'string'}, 'enum': {'minItems': 1, 'type': 'array', 'uniqueItems': True},
'exclusiveMaximum': {'default': False, 'type': 'boolean'}, 'exclusiveMinimum':
{'default': False, 'type': 'boolean'}, 'format': {'type': 'string'}, 'id':
{'type': 'string'}, 'items': {'anyOf': [{'$ref': '#'}, {'$ref':
'#/definitions/schemaArray'}], 'default': {}}, 'maxItems': {'$ref':
'#/definitions/positiveInteger'}, 'maxLength': {'$ref':
'#/definitions/positiveInteger'}, 'maxProperties': {'$ref':
'#/definitions/positiveInteger'}, 'maximum': {'type': 'number'}, 'minItems':
{'$ref': '#/definitions/positiveIntegerDefault0'}, 'minLength': {'$ref':
'#/definitions/positiveIntegerDefault0'}, 'minProperties': {'$ref':
'#/definitions/positiveIntegerDefault0'}, 'minimum': {'type': 'number'},
'multipleOf': {'exclusiveMinimum': True, 'minimum': 0, 'type': 'number'}, 'not':
{'$ref': '#'}, 'oneOf': {'$ref': '#/definitions/schemaArray'}, 'pattern':
{'format': 'regex', 'type': 'string'}, 'patternProperties':
{'additionalProperties': {'$ref': '#'}, 'default': {}, 'type': 'object'},
'properties': {'additionalProperties': {'$ref': '#'}, 'default': {}, 'type':
'object'}, 'required': {'$ref': '#/definitions/stringArray'}, 'title': {'type':
'string'}, 'type': {'anyOf': [{'$ref': '#/definitions/simpleTypes'}, {'items':
{'$ref': '#/definitions/simpleTypes'}, 'minItems': 1, 'type': 'array',
'uniqueItems': True}]}, 'uniqueItems': {'default': False, 'type': 'boolean'}},
'type': 'object'}

TYPE_CHECKER = <TypeChecker types={'array', 'boolean', 'integer', 'null', 'number',
'object', 'string'}>

VALIDATORS = {'$ref': <function ref>, 'additionalItems': <function
additionalItems>, 'additionalProperties': <function additionalProperties>, 'allOf':
<function allOf>, 'anyOf': <function anyOf>, 'dependencies': <function
dependencies_draft4_draft6_draft7>, 'enum': <function enum>, 'format': <function
format>, 'items': <function items_draft3_draft4>, 'maxItems': <function maxItems>,
'maxLength': <function maxLength>, 'maxProperties': <function maxProperties>,
'maximum': <function maximum_draft3_draft4>, 'minItems': <function minItems>,
'minLength': <function minLength>, 'minProperties': <function minProperties>,
'minimum': <function minimum_draft3_draft4>, 'multipleOf': <function multipleOf>,
'not': <function not_>, 'oneOf': <function oneOf>, 'pattern': <function pattern>,
'patternProperties': <function patternProperties>, 'properties': <function
properties>, 'required': <function required>, 'type': <function type>,
'uniqueItems': <function uniqueItems>}
```

**classmethod check_schema**(*schema*, *format_checker=<unset>*)

**descend**(*instance*, *schema*, *path=None*, *schema_path=None*, *resolver=None*)

**evolve**(***changes*)

**format_checker**: *FormatChecker* | None

**is_type**(*instance*, *type*)

**is_valid**(*instance*, *_schema=None*)

**iter_errors**(*instance*, *_schema=None*)

**property resolver**

**schema**: bool | Mapping[str, Any]

**validate**(**args*, ***kwargs*)

**class** jsonschema.validators.**Draft6Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

**FORMAT_CHECKER = <FormatChecker checkers=['email', 'idn-email', 'ipv4', 'ipv6', 'regex']>**

**static ID_OF**(*contents: bool | Mapping[str, Any]*) → str | None

```
META_SCHEMA = {'$id': 'http://json-schema.org/draft-06/schema#', '$schema':
'http://json-schema.org/draft-06/schema#', 'default': {}, 'definitions':
{'nonNegativeInteger': {'minimum': 0, 'type': 'integer'},
'nonNegativeIntegerDefault0': {'allOf': [{'$ref':
'#/definitions/nonNegativeInteger'}, {'default': 0}]}, 'schemaArray': {'items':
{'$ref': '#'}, 'minItems': 1, 'type': 'array'}, 'simpleTypes': {'enum':
['array', 'boolean', 'integer', 'null', 'number', 'object', 'string']},
'stringArray': {'default': [], 'items': {'type': 'string'}, 'type': 'array',
'uniqueItems': True}}, 'properties': {'$id': {'format': 'uri-reference', 'type':
'string'}, '$ref': {'format': 'uri-reference', 'type': 'string'}, '$schema':
{'format': 'uri', 'type': 'string'}, 'additionalItems': {'$ref': '#'},
'additionalProperties': {'$ref': '#'}, 'allOf': {'$ref':
'#/definitions/schemaArray'}, 'anyOf': {'$ref': '#/definitions/schemaArray'},
'const': {}, 'contains': {'$ref': '#'}, 'default': {}, 'definitions':
{'additionalProperties': {'$ref': '#'}, 'default': {}, 'type': 'object'},
'dependencies': {'additionalProperties': {'anyOf': [{'$ref': '#'}, {'$ref':
'#/definitions/stringArray'}]}, 'type': 'object'}, 'description': {'type':
'string'}, 'enum': {'type': 'array'}, 'examples': {'items': {}, 'type':
'array'}, 'exclusiveMaximum': {'type': 'number'}, 'exclusiveMinimum': {'type':
'number'}, 'format': {'type': 'string'}, 'items': {'anyOf': [{'$ref': '#'},
{'$ref': '#/definitions/schemaArray'}], 'default': {}}, 'maxItems': {'$ref':
'#/definitions/nonNegativeInteger'}, 'maxLength': {'$ref':
'#/definitions/nonNegativeInteger'}, 'maxProperties': {'$ref':
'#/definitions/nonNegativeInteger'}, 'maximum': {'type': 'number'}, 'minItems':
{'$ref': '#/definitions/nonNegativeIntegerDefault0'}, 'minLength': {'$ref':
'#/definitions/nonNegativeIntegerDefault0'}, 'minProperties': {'$ref':
'#/definitions/nonNegativeIntegerDefault0'}, 'minimum': {'type': 'number'},
'multipleOf': {'exclusiveMinimum': 0, 'type': 'number'}, 'not': {'$ref': '#'},
'oneOf': {'$ref': '#/definitions/schemaArray'}, 'pattern': {'format': 'regex',
'type': 'string'}, 'patternProperties': {'additionalProperties': {'$ref': '#'},
'default': {}, 'propertyNames': {'format': 'regex'}, 'type': 'object'},
'properties': {'additionalProperties': {'$ref': '#'}, 'default': {}, 'type':
'object'}, 'propertyNames': {'$ref': '#'}, 'required': {'$ref':
'#/definitions/stringArray'}, 'title': {'type': 'string'}, 'type': {'anyOf':
[{'$ref': '#/definitions/simpleTypes'}, {'items': {'$ref':
'#/definitions/simpleTypes'}, 'minItems': 1, 'type': 'array', 'uniqueItems':
True}]}, 'uniqueItems': {'default': False, 'type': 'boolean'}}, 'title': 'Core
schema meta-schema', 'type': ['object', 'boolean']}

TYPE_CHECKER = <TypeChecker types={'array', 'boolean', 'integer', 'null', 'number',
'object', 'string'}>
```

VALIDATORS = {'$ref': <function ref>, 'additionalItems': <function additionalItems>, 'additionalProperties': <function additionalProperties>, 'allOf': <function allOf>, 'anyOf': <function anyOf>, 'const': <function const>, 'contains': <function contains_draft6_draft7>, 'dependencies': <function dependencies_draft4_draft6_draft7>, 'enum': <function enum>, 'exclusiveMaximum': <function exclusiveMaximum>, 'exclusiveMinimum': <function exclusiveMinimum>, 'format': <function format>, 'items': <function items_draft6_draft7_draft201909>, 'maxItems': <function maxItems>, 'maxLength': <function maxLength>, 'maxProperties': <function maxProperties>, 'maximum': <function maximum>, 'minItems': <function minItems>, 'minLength': <function minLength>, 'minProperties': <function minProperties>, 'minimum': <function minimum>, 'multipleOf': <function multipleOf>, 'not': <function not_>, 'oneOf': <function oneOf>, 'pattern': <function pattern>, 'patternProperties': <function patternProperties>, 'properties': <function properties>, 'propertyNames': <function propertyNames>, 'required': <function required>, 'type': <function type>, 'uniqueItems': <function uniqueItems>}

classmethod check_schema(*schema*, *format_checker=<unset>*)

descend(*instance*, *schema*, *path=None*, *schema_path=None*, *resolver=None*)

evolve(***changes*)

format_checker: *FormatChecker* | None

is_type(*instance*, *type*)

is_valid(*instance*, *_schema=None*)

iter_errors(*instance*, *_schema=None*)

property resolver

schema: bool | Mapping[str, Any]

validate(**args*, ***kwargs*)

class jsonschema.validators.**Draft7Validator**(*schema: bool | ~collections.abc.Mapping[str, ~typing.Any], resolver=None, format_checker: ~jsonschema._format.FormatChecker | None = None, *, registry: ~referencing._core.Registry[bool | ~collections.abc.Mapping[str, ~typing.Any]] = <Registry (20 resources)>, _resolver=None*)

FORMAT_CHECKER = <FormatChecker checkers=['date', 'email', 'idn-email', 'idn-hostname', 'ipv4', 'ipv6', 'regex']>

static ID_OF(*contents: bool | Mapping[str, Any]*) → str | None

```
META_SCHEMA = {'$id': 'http://json-schema.org/draft-07/schema#', '$schema':
'http://json-schema.org/draft-07/schema#', 'default': True, 'definitions':
{'nonNegativeInteger': {'minimum': 0, 'type': 'integer'},
'nonNegativeIntegerDefault0': {'allOf': [{'$ref':
'#/definitions/nonNegativeInteger'}, {'default': 0}]}, 'schemaArray': {'items':
{'$ref': '#'}, 'minItems': 1, 'type': 'array'}, 'simpleTypes': {'enum':
['array', 'boolean', 'integer', 'null', 'number', 'object', 'string']},
'stringArray': {'default': [], 'items': {'type': 'string'}, 'type': 'array',
'uniqueItems': True}}, 'properties': {'$comment': {'type': 'string'}, '$id':
{'format': 'uri-reference', 'type': 'string'}, '$ref': {'format':
'uri-reference', 'type': 'string'}, '$schema': {'format': 'uri', 'type':
'string'}, 'additionalItems': {'$ref': '#'}, 'additionalProperties': {'$ref':
'#'}, 'allOf': {'$ref': '#/definitions/schemaArray'}, 'anyOf': {'$ref':
'#/definitions/schemaArray'}, 'const': True, 'contains': {'$ref': '#'},
'contentEncoding': {'type': 'string'}, 'contentMediaType': {'type': 'string'},
'default': True, 'definitions': {'additionalProperties': {'$ref': '#'},
'default': {}, 'type': 'object'}, 'dependencies': {'additionalProperties':
{'anyOf': [{'$ref': '#'}, {'$ref': '#/definitions/stringArray'}]}, 'type':
'object'}, 'description': {'type': 'string'}, 'else': {'$ref': '#'}, 'enum':
{'items': True, 'type': 'array'}, 'examples': {'items': True, 'type': 'array'},
'exclusiveMaximum': {'type': 'number'}, 'exclusiveMinimum': {'type': 'number'},
'format': {'type': 'string'}, 'if': {'$ref': '#'}, 'items': {'anyOf':
[{'$ref': '#'}, {'$ref': '#/definitions/schemaArray'}], 'default': True},
'maxItems': {'$ref': '#/definitions/nonNegativeInteger'}, 'maxLength': {'$ref':
'#/definitions/nonNegativeInteger'}, 'maxProperties': {'$ref':
'#/definitions/nonNegativeInteger'}, 'maximum': {'type': 'number'}, 'minItems':
{'$ref': '#/definitions/nonNegativeIntegerDefault0'}, 'minLength': {'$ref':
'#/definitions/nonNegativeIntegerDefault0'}, 'minProperties': {'$ref':
'#/definitions/nonNegativeIntegerDefault0'}, 'minimum': {'type': 'number'},
'multipleOf': {'exclusiveMinimum': 0, 'type': 'number'}, 'not': {'$ref': '#'},
'oneOf': {'$ref': '#/definitions/schemaArray'}, 'pattern': {'format': 'regex',
'type': 'string'}, 'patternProperties': {'additionalProperties': {'$ref': '#'},
'default': {}, 'propertyNames': {'format': 'regex'}, 'type': 'object'},
'properties': {'additionalProperties': {'$ref': '#'}, 'default': {}, 'type':
'object'}, 'propertyNames': {'$ref': '#'}, 'readOnly': {'default': False,
'type': 'boolean'}, 'required': {'$ref': '#/definitions/stringArray'}, 'then':
{'$ref': '#'}, 'title': {'type': 'string'}, 'type': {'anyOf': [{'$ref':
'#/definitions/simpleTypes'}, {'items': {'$ref': '#/definitions/simpleTypes'},
'minItems': 1, 'type': 'array', 'uniqueItems': True}]}, 'uniqueItems':
{'default': False, 'type': 'boolean'}}, 'title': 'Core schema meta-schema',
'type': ['object', 'boolean']}

TYPE_CHECKER = <TypeChecker types={'array', 'boolean', 'integer', 'null', 'number',
'object', 'string'}>
```

VALIDATORS = {'$ref': <function ref>, 'additionalItems': <function additionalItems>, 'additionalProperties': <function additionalProperties>, 'allOf': <function allOf>, 'anyOf': <function anyOf>, 'const': <function const>, 'contains': <function contains_draft6_draft7>, 'dependencies': <function dependencies_draft4_draft6_draft7>, 'enum': <function enum>, 'exclusiveMaximum': <function exclusiveMaximum>, 'exclusiveMinimum': <function exclusiveMinimum>, 'format': <function format>, 'if': <function if_>, 'items': <function items_draft6_draft7_draft201909>, 'maxItems': <function maxItems>, 'maxLength': <function maxLength>, 'maxProperties': <function maxProperties>, 'maximum': <function maximum>, 'minItems': <function minItems>, 'minLength': <function minLength>, 'minProperties': <function minProperties>, 'minimum': <function minimum>, 'multipleOf': <function multipleOf>, 'not': <function not_>, 'oneOf': <function oneOf>, 'pattern': <function pattern>, 'patternProperties': <function patternProperties>, 'properties': <function properties>, 'propertyNames': <function propertyNames>, 'required': <function required>, 'type': <function type>, 'uniqueItems': <function uniqueItems>}

classmethod check_schema(*schema*, *format_checker=<unset>*)

descend(*instance*, *schema*, *path=None*, *schema_path=None*, *resolver=None*)

evolve(***changes*)

format_checker: *FormatChecker* | None

is_type(*instance*, *type*)

is_valid(*instance*, *_schema=None*)

iter_errors(*instance*, *_schema=None*)

property resolver

schema: bool | Mapping[str, Any]

validate(**args*, ***kwargs*)

class jsonschema.validators._RefResolver(*base_uri*, *referrer*, *store=HashTrieMap({})*, *cache_remote=True*, *handlers=()*, *urljoin_cache=None*, *remote_cache=None*)

Resolve JSON References.

**Parameters**

- **base_uri** (*str*) – The URI of the referring document
- **referrer** – The actual referring document
- **store** (*dict*) – A mapping from URIs to documents to cache
- **cache_remote** (*bool*) – Whether remote refs should be cached after first resolution
- **handlers** (*dict*) – A mapping from URI schemes to functions that should be used to retrieve them
- **urljoin_cache** (functools.lru_cache()) – A cache that will be used for caching the results of joining the resolution scope to subscopes.
- **remote_cache** (functools.lru_cache()) – A cache that will be used for caching the results of resolved remote URLs.

**cache_remote**

Whether remote refs should be cached after first resolution

**Type**
bool

Deprecated since version v4.18.0: `RefResolver` has been deprecated in favor of `referencing`.

**property base_uri**

Retrieve the current base URI, not including any fragment.

**classmethod from_schema**(*schema*, *id_of=<function _dollar_id>*, **args*, ***kwargs*)

Construct a resolver from a JSON schema object.

**Parameters**
**schema** – the referring schema

**Returns**
*_RefResolver*

**in_scope**(*scope*)

Temporarily enter the given scope for the duration of the context.

Deprecated since version v4.0.0.

**pop_scope**()

Exit the most recent entered scope.

Treats further dereferences as being performed underneath the original scope.

Don't call this method more times than *push_scope* has been called.

**push_scope**(*scope*)

Enter a given sub-scope.

Treats further dereferences as being performed underneath the given scope.

**property resolution_scope**

Retrieve the current resolution scope.

**resolve**(*ref*)

Resolve the given reference.

**resolve_fragment**(*document*, *fragment*)

Resolve a `fragment` within the referenced `document`.

**Parameters**

- **document** – The referent document
- **fragment** (*str*) – a URI fragment to resolve within it

**resolve_from_url**(*url*)

Resolve the given URL.

**resolve_remote**(*uri*)

Resolve a remote `uri`.

If called directly, does not check the store first, but after retrieving the document at the specified URI it will be saved in the store if *cache_remote* is True.

**Note:** If the requests library is present, `jsonschema` will use it to request the remote `uri`, so that the correct encoding is detected and used.

If it isn't, or if the scheme of the `uri` is not `http` or `https`, UTF-8 is assumed.

**Parameters**
**uri** (*str*) – The URI to resolve

**Returns**
The retrieved document

**resolving**(*ref*)

Resolve the given `ref` and enter its resolution scope.

Exits the scope on exit of this context manager.

**Parameters**
**ref** (*str*) – The reference to resolve

jsonschema.validators.**create**(*meta_schema: ~collections.abc.Mapping[str, ~typing.Any], validators: ~collections.abc.Mapping[str, ~jsonschema._typing.SchemaKeywordValidator] | ~collections.abc.Iterable[tuple[str, ~jsonschema._typing.SchemaKeywordValidator]] = (), version: str | None = None, type_checker: ~jsonschema._types.TypeChecker = <TypeChecker types={'array', 'boolean', 'integer', 'null', 'number', 'object', 'string'}>, format_checker: ~jsonschema._format.FormatChecker = <FormatChecker checkers=['date', 'email', 'idn-email', 'idn-hostname', 'ipv4', 'ipv6', 'regex', 'uuid']>, id_of: ~typing.Callable[[bool | ~collections.abc.Mapping[str, ~typing.Any]], str | None] = <function _dollar_id>, applicable_validators: ~typing.Callable[[bool | ~collections.abc.Mapping[str, ~typing.Any]], ~typing.Iterable[~typing.Tuple[str, ~typing.Any]]] = operator.methodcaller('items')*)

Create a new validator class.

**Parameters**

- **meta_schema** – the meta schema for the new validator class
- **validators** – a mapping from names to callables, where each callable will validate the schema property with the given name.

  Each callable should take 4 arguments:

  1. a validator instance,
  2. the value of the property being validated within the instance
  3. the instance
  4. the schema
- **version** – an identifier for the version that this validator class will validate. If provided, the returned validator class will have its `__name__` set to include the version, and also will have *jsonschema.validators.validates* automatically called for the given version.
- **type_checker** – a type checker, used when applying the type keyword.

  If unprovided, a *jsonschema.TypeChecker* will be created with a set of default types typical of JSON Schema drafts.

- **format_checker** – a format checker, used when applying the format keyword.

  If unprovided, a *jsonschema.FormatChecker* will be created with a set of default formats typical of JSON Schema drafts.

- **id_of** – A function that given a schema, returns its ID.

- **applicable_validators** – A function that, given a schema, returns the list of applicable schema keywords and associated values which will be used to validate the instance. This is mostly used to support pre-draft 7 versions of JSON Schema which specified behavior around ignoring keywords if they were siblings of a `$ref` keyword. If you're not attempting to implement similar behavior, you can typically ignore this argument and leave it at its default.

**Returns**

a new *jsonschema.protocols.Validator* class

jsonschema.validators.**extend**(*validator*, *validators=()*, *version=None*, *type_checker=None*, *format_checker=None*)

Create a new validator class by extending an existing one.

**Parameters**

- **validator** (jsonschema.protocols.Validator) – an existing validator class

- **validators** (*collections.abc.Mapping*) – a mapping of new validator callables to extend with, whose structure is as in *create*.

  ---

  **Note:** Any validator callables with the same name as an existing one will (silently) replace the old validator callable entirely, effectively overriding any validation done in the "parent" validator class.

  If you wish to instead extend the behavior of a parent's validator callable, delegate and call it directly in the new validator function by retrieving it using `OldValidator.VALIDATORS["validation_keyword_name"]`.

  ---

- **version** (*str*) – a version for the new validator class

- **type_checker** (jsonschema.TypeChecker) – a type checker, used when applying the type keyword.

  If unprovided, the type checker of the extended *jsonschema.protocols.Validator* will be carried along.

- **format_checker** (jsonschema.FormatChecker) – a format checker, used when applying the format keyword.

  If unprovided, the format checker of the extended *jsonschema.protocols.Validator* will be carried along.

**Returns**

a new *jsonschema.protocols.Validator* class extending the one provided

---

**Note:** Meta Schemas

The new validator class will have its parent's meta schema.

If you wish to change or extend the meta schema in the new validator class, modify `META_SCHEMA` directly on the returned class. Note that no implicit copying is done, so a copy should likely be made before modifying it,

in order to not affect the old validator.

---

jsonschema.validators.**validate**(*instance*, *schema*, *cls=None*, **args*, ***kwargs*)

Validate an instance under the given schema.

```
>>> validate([2, 3, 4], {"maxItems": 2})
Traceback (most recent call last):
    ...
ValidationError: [2, 3, 4] is too long
```

*validate()* will first verify that the provided schema is itself valid, since not doing so can lead to less obvious error messages and fail in less obvious or consistent ways.

If you know you have a valid schema already, especially if you intend to validate multiple instances with the same schema, you likely would prefer using the *jsonschema.protocols.Validator.validate* method directly on a specific validator (e.g. `Draft202012Validator.validate`).

**Parameters**

- **instance** – The instance to validate
- **schema** – The schema to validate with
- **cls** (jsonschema.protocols.Validator) – The class that will be used to validate the instance.

If the `cls` argument is not provided, two things will happen in accordance with the specification. First, if the schema has a $schema keyword containing a known meta-schema[1] then the proper validator will be used. The specification recommends that all schemas contain $schema properties for this reason. If no $schema property is found, the default validator class is the latest released draft.

Any other provided positional and keyword arguments will be passed on when instantiating the `cls`.

**Raises**

- ***jsonschema.exceptions.ValidationError*** – if the instance is invalid
- ***jsonschema.exceptions.SchemaError*** – if the schema itself is invalid

jsonschema.validators.**validates**(*version*)

Register the decorated validator for a `version` of the specification.

Registered validators and their meta schemas will be considered when parsing $schema keywords' URIs.

**Parameters**

**version** (*str*) – An identifier to use as the version's name

**Returns**

a class decorator to decorate the validator with the version

**Return type**

collections.abc.Callable

jsonschema.validators.**validator_for**(*schema*, *default: Validator | _utils.Unset = <unset>*) → type[*Validator*]

Retrieve the validator class appropriate for validating the given schema.

Uses the $schema keyword that should be present in the given schema to look up the appropriate validator class.

**Parameters**

[1] known by a validator registered with *jsonschema.validators.validates*

- **schema** (*collections.abc.Mapping or bool*) – the schema to look at
- **default** – the default to return if the appropriate validator class cannot be determined.

  If unprovided, the default is to return the latest supported draft.

### Examples

The $schema JSON Schema keyword will control which validator class is returned:

```
>>> schema = {
...     "$schema": "https://json-schema.org/draft/2020-12/schema",
...     "type": "integer",
... }
>>> jsonschema.validators.validator_for(schema)
<class 'jsonschema.validators.Draft202012Validator'>
```

Here, a draft 7 schema instead will return the draft 7 validator:

```
>>> schema = {
...     "$schema": "http://json-schema.org/draft-07/schema#",
...     "type": "integer",
... }
>>> jsonschema.validators.validator_for(schema)
<class 'jsonschema.validators.Draft7Validator'>
```

Schemas with no `$schema` keyword will fallback to the default argument:

```
>>> schema = {"type": "integer"}
>>> jsonschema.validators.validator_for(
...     schema, default=Draft7Validator,
... )
<class 'jsonschema.validators.Draft7Validator'>
```

or if none is provided, to the latest version supported. Always including the keyword when authoring schemas is highly recommended.

## jsonschema.exceptions

Validation errors, and some surrounding helpers.

**class** jsonschema.exceptions.**ErrorTree**(*errors: Iterable[ValidationError] = ()*)

ErrorTrees make it easier to check which validations failed.

**property total_errors**

The total number of errors in the entire tree, including children.

**exception** jsonschema.exceptions.**FormatError**(*message*, *cause=None*)

Validating a format failed.

**exception** jsonschema.exceptions.**SchemaError**(*message: str*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*, *type_checker=<unset>*)

A schema was invalid under its corresponding metaschema.

**exception** jsonschema.exceptions.**UndefinedTypeCheck**(*type*)

A type checker was asked to check a type it did not have registered.

**exception** jsonschema.exceptions.**UnknownType**(*type*, *instance*, *schema*)

A validator was asked to validate an instance against an unknown type.

**exception** jsonschema.exceptions.**ValidationError**(*message: str*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*, *type_checker=<unset>*)

An instance was invalid under a provided schema.

jsonschema.exceptions.**best_match**(*errors*, *key=<function by_relevance.<locals>.relevance>*)

Try to find an error that appears to be the best match among given errors.

In general, errors that are higher up in the instance (i.e. for which *ValidationError.path* is shorter) are considered better matches, since they indicate "more" is wrong with the instance.

If the resulting match is either oneOf or anyOf, the *opposite* assumption is made – i.e. the deepest error is picked, since these keywords only need to match once, and any other errors may not be relevant.

**Parameters**

- **errors** (*collections.abc.Iterable*) – the errors to select from. Do not provide a mixture of errors from different validation attempts (i.e. from different instances or schemas), since it won't produce sensical output.
- **key** (*collections.abc.Callable*) – the key to use when sorting errors. See *relevance* and transitively *by_relevance* for more details (the default is to sort with the defaults of that function). Changing the default is only useful if you want to change the function that rates errors but still want the error context descent done by this function.

**Returns**

the best matching error, or `None` if the iterable was empty

---

**Note:** This function is a heuristic. Its return value may change for a given set of inputs from version to version if better heuristics are added.

---

jsonschema.exceptions.**by_relevance**(*weak=frozenset({'anyOf', 'oneOf'})*, *strong=frozenset({})*)

Create a key function that can be used to sort errors by relevance.

**Parameters**

- **weak** (*set*) – a collection of validation keywords to consider to be "weak". If there are two errors at the same level of the instance and one is in the set of weak validation keywords, the other error will take priority. By default, anyOf and oneOf are considered weak keywords and will be superseded by other same-level validation errors.
- **strong** (*set*) – a collection of validation keywords to consider to be "strong"

jsonschema.exceptions.**relevance**(*error*)

A key function (e.g. to use with `sorted`) which sorts errors by relevance.

Example:

```
sorted(validator.iter_errors(12), key=jsonschema.exceptions.relevance)
```

### jsonschema.protocols

typing.Protocol classes for jsonschema interfaces.

**class** jsonschema.protocols.**Validator**(*schema: Mapping | bool*, *registry: referencing.jsonschema.SchemaRegistry*, *format_checker: jsonschema.FormatChecker | None = None*)

The protocol to which all validator classes adhere.

**Parameters**

- **schema** – The schema that the validator object will validate with. It is assumed to be valid, and providing an invalid schema can lead to undefined behavior. See *Validator.check_schema* to validate a schema first.
- **registry** – a schema registry that will be used for looking up JSON references
- **resolver** – a resolver that will be used to resolve $ref properties (JSON references). If unprovided, one will be created.

  Deprecated since version v4.18.0: *RefResolver* has been deprecated in favor of `referencing`, and with it, this argument.
- **format_checker** – if provided, a checker which will be used to assert about format properties present in the schema. If unprovided, *no* format validation is done, and the presence of format within schemas is strictly informational. Certain formats require additional packages to be installed in order to assert against instances. Ensure you've installed *jsonschema* with its *extra (optional) dependencies* when invoking `pip`.

Deprecated since version v4.12.0: Subclassing validator classes now explicitly warns this is not part of their public API.

**FORMAT_CHECKER: ClassVar[*jsonschema.FormatChecker*]**

A *jsonschema.FormatChecker* that will be used when validating format keywords in JSON schemas.

**ID_OF: *_typing.id_of***

A function which given a schema returns its ID.

**META_SCHEMA: ClassVar[Mapping]**

An object representing the validator's meta schema (the schema that describes valid schemas in the given version).

**TYPE_CHECKER: ClassVar[*jsonschema.TypeChecker*]**

A *jsonschema.TypeChecker* that will be used when validating type keywords in JSON schemas.

**VALIDATORS: ClassVar[Mapping]**

A mapping of validation keywords (`str`s) to functions that validate the keyword with that name. For more information see *Creating or Extending Validator Classes*.

**classmethod check_schema**(*schema: Mapping | bool*) → None

Validate the given schema against the validator's *META_SCHEMA*.

**Raises**

***jsonschema.exceptions.SchemaError*** – if the schema is invalid

**evolve**(***kwargs*) → *Validator*

Create a new validator like this one, but with given changes.

Preserves all other attributes, so can be used to e.g. create a validator with a different schema but with the same $ref resolution behavior.

```
>>> validator = Draft202012Validator({})
>>> validator.evolve(schema={"type": "number"})
Draft202012Validator(schema={'type': 'number'}, format_checker=None)
```

The returned object satisfies the validator protocol, but may not be of the same concrete class! In particular this occurs when a $ref occurs to a schema with a different $schema than this one (i.e. for a different draft).

```
>>> validator.evolve(
...     schema={"$schema": Draft7Validator.META_SCHEMA["$id"]}
... )
Draft7Validator(schema=..., format_checker=None)
```

**is_type**(*instance: Any*, *type: str*) → bool

Check if the instance is of the given (JSON Schema) type.

**Parameters**

- **instance** – the value to check
- **type** – the name of a known (JSON Schema) type

**Returns**
whether the instance is of the given type

**Raises**
*jsonschema.exceptions.UnknownType* – if `type` is not a known type

**is_valid**(*instance: Any*) → bool

Check if the instance is valid under the current *schema*.

**Returns**
whether the instance is valid or not

```
>>> schema = {"maxItems" : 2}
>>> Draft202012Validator(schema).is_valid([2, 3, 4])
False
```

**iter_errors**(*instance: Any*) → Iterable[*ValidationError*]

Lazily yield each of the validation errors in the given instance.

```
>>> schema = {
...     "type" : "array",
...     "items" : {"enum" : [1, 2, 3]},
...     "maxItems" : 2,
... }
>>> v = Draft202012Validator(schema)
>>> for error in sorted(v.iter_errors([2, 3, 4]), key=str):
...     print(error.message)
4 is not one of [1, 2, 3]
[2, 3, 4] is too long
```

Deprecated since version v4.0.0: Calling this function with a second schema argument is deprecated. Use *Validator.evolve* instead.

**schema: Mapping | bool**

The schema that will be used to validate instances

**validate**(*instance: Any*) → None

Check if the instance is valid under the current *schema*.

**Raises**

***jsonschema.exceptions.ValidationError*** – if the instance is invalid

```
>>> schema = {"maxItems" : 2}
>>> Draft202012Validator(schema).validate([2, 3, 4])
Traceback (most recent call last):
    ...
ValidationError: [2, 3, 4] is too long
```

### 7.6.2 jsonschema

An implementation of JSON Schema for Python.

The main functionality is provided by the validator classes for each of the supported JSON Schema versions.

Most commonly, *jsonschema.validators.validate* is the quickest way to simply validate a given instance under a schema, and will create a validator for you.

**class** jsonschema.**FormatChecker**(*formats: Iterable[str] | None = None*)

A `format` property checker.

JSON Schema does not mandate that the `format` property actually do any validation. If validation is desired however, instances of this class can be hooked into validators to enable format validation.

*FormatChecker* objects always return `True` when asked about formats that they do not know how to validate.

To add a check for a custom format use the *FormatChecker.checks* decorator.

**Parameters**

**formats** – The known formats to validate. This argument can be used to limit which formats will be used during validation.

**check**(*instance: object*, *format: str*) → None

Check whether the instance conforms to the given format.

**Parameters**

- **instance** (*any primitive type*, i.e. str, number, bool) – The instance to check
- **format** – The format that instance should conform to

**Raises**

***FormatError*** – if the instance does not conform to `format`

**checks**(*format: str*, *raises: Type[Exception] | Tuple[Type[Exception], ...] = ()*) → Callable[[_F], _F]

Register a decorated function as validating a new format.

**Parameters**

- **format** – The format that the decorated function will check.
- **raises** – The exception(s) raised by the decorated function when an invalid instance is found.

  The exception object will be accessible as the *jsonschema.exceptions.ValidationError.cause* attribute of the resulting validation error.

**conforms**(*instance: object*, *format: str*) → bool

Check whether the instance conforms to the given format.

**Parameters**

- **instance** (*any primitive type*, i.e. str, number, bool) – The instance to check
- **format** – The format that instance should conform to

**Returns**
whether it conformed

**Return type**
bool

**exception** jsonschema.**SchemaError**(*message: str*, *validator=<unset>*, *path=()*, *cause=None*, *context=()*, *validator_value=<unset>*, *instance=<unset>*, *schema=<unset>*, *schema_path=()*, *parent=None*, *type_checker=<unset>*)

A schema was invalid under its corresponding metaschema.

**class** jsonschema.**TypeChecker**(*type_checkers: Mapping[str, Callable[[TypeChecker, Any], bool]] = HashTrieMap({})*)

A type property checker.

A *TypeChecker* performs type checking for a *Validator*, converting between the defined JSON Schema types and some associated Python types or objects.

Modifying the behavior just mentioned by redefining which Python objects are considered to be of which JSON Schema types can be done using *TypeChecker.redefine* or *TypeChecker.redefine_many*, and types can be removed via *TypeChecker.remove*. Each of these return a new *TypeChecker*.

**Parameters**
**type_checkers** – The initial mapping of types to their checking functions.

**is_type**(*instance*, *type: str*) → bool

Check if the instance is of the appropriate type.

**Parameters**

- **instance** – The instance to check
- **type** – The name of the type that is expected.

**Raises**
*jsonschema.exceptions.UndefinedTypeCheck* – if `type` is unknown to this object.

**redefine**(*type: str*, *fn*) → *TypeChecker*

Produce a new checker with the given type redefined.

**Parameters**

- **type** – The name of the type to check.
- **fn** (*collections.abc.Callable*) – A callable taking exactly two parameters - the type checker calling the function and the instance to check. The function should return true if instance is of this type and false otherwise.

**redefine_many**(*definitions=()*) → *TypeChecker*

Produce a new checker with the given types redefined.

**Parameters**
**definitions** (*dict*) – A dictionary mapping types to their checking functions.

**remove**(**types*) → *TypeChecker*

Produce a new checker with the given types forgotten.

**Parameters**

**types** – the names of the types to remove.

**Raises**

***jsonschema.exceptions.UndefinedTypeCheck*** – if any given type is unknown to this object

jsonschema.**validate**(*instance*, *schema*, *cls=None*, **args*, ***kwargs*)

Validate an instance under the given schema.

```
>>> validate([2, 3, 4], {"maxItems": 2})
Traceback (most recent call last):
    ...
ValidationError: [2, 3, 4] is too long
```

*validate()* will first verify that the provided schema is itself valid, since not doing so can lead to less obvious error messages and fail in less obvious or consistent ways.

If you know you have a valid schema already, especially if you intend to validate multiple instances with the same schema, you likely would prefer using the *jsonschema.protocols.Validator.validate* method directly on a specific validator (e.g. `Draft202012Validator.validate`).

**Parameters**

- **instance** – The instance to validate
- **schema** – The schema to validate with
- **cls** (jsonschema.protocols.Validator) – The class that will be used to validate the instance.

If the `cls` argument is not provided, two things will happen in accordance with the specification. First, if the schema has a $schema keyword containing a known meta-schema[1] then the proper validator will be used. The specification recommends that all schemas contain $schema properties for this reason. If no $schema property is found, the default validator class is the latest released draft.

Any other provided positional and keyword arguments will be passed on when instantiating the `cls`.

**Raises**

- ***jsonschema.exceptions.ValidationError*** – if the instance is invalid
- ***jsonschema.exceptions.SchemaError*** – if the schema itself is invalid

jsonschema._format.**_F = ~_F**

A format checker callable.

jsonschema._typing.**id_of**

alias of `Callable[[bool | Mapping[str, Any]], str | None]`

[1] known by a validator registered with *jsonschema.validators.validates*

## 7.7 Indices and tables

- genindex

# PYTHON MODULE INDEX

## j

# INDEX

## Symbols



## A



## B



## C



## D



## E

## F



## I

## J



## M



## P



## R



## S

## T



## U



## V