dataspec

May 11, 2020

Contents:

Dataspec is a data specification and normalization toolkit written in pure Python. With Dataspec, you can create Specs to validate and normalize data of almost any shape. Dataspec is inspired by Clojure's spec library.

CHAPTER 1

Getting Started

Contents

1.1 What are Specs?

Specs are declarative data specifications written in pure Python code. Specs can be created using the generic Spec constructor function `dataspec.s()`. Specs provide two useful and related functions. The first is to evaluate whether an arbitrary data structure satisfies the specification. The second function is to conform (or normalize) valid data structures into a canonical format.

The simplest Specs are based on common predicate functions, such as `lambda x: isinstance(x, str)` which asks "Is the object x an instance of `str`?". Fortunately, Specs are not limited to being created from single predicates. Specs can also be created from groups of predicates, composed in a variety of useful ways, and even defined for complex data structures. Because Specs are ultimately backed by pure Python code, any question that you can answer about your data in code can be encoded in a Spec.

1.2 Features

- Simple API using primarily native Python types and data structures

- Stateless, immutable Spec objects are designed to be created once, reused, and composed
- Rich error objects point to the exact location of the error in the input value
- Builtin factories for many common validations

1.3 Installation

Dataspec is developed on GitHub and hosted on PyPI. You can fetch Dataspec using `pip`:

```
pip install dataspec
```

To enable support for phone number specs or arbitrary date strings, you can choose the extras when you install:

```
pip install dataspec[dates]
pip install dataspec[phonenumbers]
```

1.4 First Steps

To begin using the `dataspec` library, you can simply import the `s` object:

```
from dataspec import s
```

`s` is a generic constructor for creating new Specs. Many useful Specs can be composed from basic Python objects like types, functions, and data structures. The "Hello, world!" equivalent for creating new Specs might be a simple Spec that validates that an input is a string (a Python `str`). We can do this by simply passing the Python `str` type directly to `s`. When `s` receives an instance of a `type` object, it assumes you want to create a Spec that validates input values are of that type:

```
spec = s(str)
spec.is_valid("a string")   # True
spec.is_valid(3)            # False
```

Often you want to assert more than one condition on an input value. After all, it's fairly trivial to assert type checks on a value. In fact, this may even be done by a deserialization library on your behalf. Perhaps you're interested in checking that your input is a string and that it contains only numbers and hyphens. `dataspec` lets you define Specs with boolean logic, which can be useful for asserting multiple conditions on your input:

```
spec = s.all(str, lambda s: all(c.isdecimal() or c == "-" for c in s))
spec.is_valid("212-867-5309")      # True
spec.is_valid("Philip Jennings")   # False
```

Composition is at the heart of `dataspec` 's design. In the previous example, we learned a few useful things. First, `s` is actually a callable object with static methods which help produce other sorts of Specs. Second, we can see that when we pass objects understood to `s` into various Spec constructors, they are automatically coerced into the appropriate Spec type. Here, we passed a `type`, which we used previously. We also passed in a function of one argument returning a boolean; in `dataspec`, these are called predicates and they are turned into Specs which validate input values if the function returns `True` and fail otherwise. Finally, we learned that `s.all` can be used to produce `and` -type boolean logic between different Specs. (You can produce `or` Specs using `s.any`).

In the previous example, we used the `and` logic to check for our conditions to show various different features of `dataspec`. However, in real code you'd likely take advantage of `dataspec` 's builtin `s.str` factory, which can assert several useful properties of strings (in addition to the basic `isinstance` check). In the case above, perhaps

we really wanted to check for a US ZIP code (with the trailing 4 digits). We can perform that check using a simple regex string validator:

```
spec = s.str("us_zip_plus_4", regex=r"\d{5}\-\d{4}")
spec.is_valid("10001-3093")  # True
spec.is_valid("10001")       # False
spec.is_valid("N0L 1E0")     # False
```

Scalar Specs like the one above trivially different from the same checks you could write in raw Python. The real power of `dataspec` comes from its ability to compose Specs for larger, nested data structures. Suppose you were accepting a physician profile object via a JSON API and you wanted to validate that the physician licenses were valid in all of the states you operate in:

```
operating_states = s("operating_states", {"CA", "GA", "NY"})
license_states = s("license_states", [operating_states, {"kind": list}])
license_states.is_valid(["CA", "NY"])  # True
license_states.is_valid(["SD", "GA"])  # False, you do not operate in South Dakota
license_states.is_valid({"CA"})        # False, as the input collection is a set
```

In the previous example, we learned a bit more about `dataspec`. First, we can see that Spec objects are designed to be reused. We declared `operating_states` as a separate Spec from `license_states` with the intent that we could use it as a component of other Specs. Specs are immutable and stateless, so they can be reused in other Specs without issue. Next, we can see that we're expecting a collection, indicated by the Python `list` wrapping `operating_states` in the `license_states` Spec. In particular, we are expecting exactly a `list`, not a `set` or `tuple`. Third, we are expecting a limited set of enumerated values, indicated by `operating_states` being a `set`. Values not in the set are rejected. `dataspec` also supports using Python's `Enum` objects for defining enumerated types.

We did declare two separate Specs and pass both to `s` directly. However, we could have declared the entire Spec inline and `s` would have converted each child value into a Spec automatically: `s([{"CA", "GA", "NY"}, {"kind": list}])`.

Building on the previous example, let's suppose we want to validate a simplified version of that physician profile object. Spec is great for validating data at your application boundaries. You can pass it your deserialized input values and it will help you ensure that you're receiving data in the shape your internal services expect:

```
spec = s(
    "user-profile",
    {
        "id": s.str("id", format_="uuid"),
        "first_name": s.str("first_name"),
        "last_name": s.str("last_name"),
        "date_of_birth": s.str("date_of_birth", format_="iso-date"),
        s.opt("gender"): s("gender", {"M", "F"}),
        "license_states": license_states,  # using the previously defined Spec
    }
)
spec.is_valid(  # True
    {
        "id": "e1bc9fb2-a4d3-4683-bfef-3acc61b0edcc",
        "first_name": "Carl",
        "last_name": "Sagan",
        "date_of_birth": "1996-12-20",
        "license_states": ["CA"],
    }
)
spec.is_valid(  # False; the optional "gender" key included an invalid value
```

(continues on next page)

(continued from previous page)

```
    {
        "id": "e1bc9fb2-a4d3-4683-bfef-3acc61b0edcc",
        "first_name": "Carl",
        "last_name": "Sagan",
        "date_of_birth": "1996-12-20",
        "gender": "O",
        "license_states": ["CA"],
    }
)
spec.is_valid(  # True; note that extra keys _are ignored_
    {
        "id": "958e2f55-5fdf-4b84-a522-a0765299ba4b",
        "first_name": "Marie",
        "last_name": "Curie",
        "date_of_birth": "1867-11-07",
        "gender": "F",
        "license_states": ["NY", "GA"],
        "occupation": "Chemist",
    }
)
spec.is_valid(  # False; the "license_states" includes the invalid value "TX"
    {
        "id": "958e2f55-5fdf-4b84-a522-a0765299ba4b",
        "first_name": "Marie",
        "last_name": "Curie",
        "date_of_birth": "1867-11-07",
        "license_states": ["TX"],
    }
)
```

1.5 Why not X?

Python's ecosystem features a rich collection of data validation and normalization tools, so a new entrant in the space naturally begs the question "why didn't you just use X instead?". Before creating Dataspec, we surveyed a wide variety of different tools and had even used one or two in our production service. All of these tools are generally successful at validating data, but each had some issue that caused us to pass.

- Many of the libraries in this space primarily help validate data, but do not always help you normalize or conform that data after it has been validated. Dataspec provides validation and conformation out of the box.
- Libraries which do feature validation and normalization often complect these two steps. Dataspec validation is a discrete step that occurs before conformation, so it is easy to reason about failures in validation.
- Some of the libraries we tried were stateful or leaned too heavily on mutability. We tend to prefer immutable and stateless objects where mutability and state is not required. Specs in Dataspec are completely stateless and conformation always produces a new value. This is certainly more costly than mutating inputs, but mutating code is harder to reason about and is a major source of bugs, so we prefer to avoid it.
- Many libraries we surveyed focused on defining validations from the top-down, rather than encouraging composition. Specs in Dataspec are designed to be created once, reused, and composed, rather than requiring a separate definition for each usage.

CHAPTER 2

Usage

Contents

2.1 Constructing Specs

To begin using the `dataspec` library, you can simply import the *`dataspec.s`* object:

```
from dataspec import s
```

`s()` is a generic Spec constructor, which can be called to construct new Specs from a variety of sources. It is a singleton instance of *`dataspec.SpecAPI`* and nearly all of the factory or convenience methods below are available as static methods on *`s()`*.

Specs are designed to be composed, so each of the spec types below can serve as the base for more complex data definitions. For collection, mapping, and tuple type Specs, Specs will be recursively created for child elements if they are types understood by *`s()`*.

Specs may also optionally be created with *Tags*, which are just string names provided in *`dataspec.ErrorDetails`* objects emitted by Spec instance *`dataspec.Spec.validate()`* methods. For *`s()`*, tags may be provided as the first positional argument. Specs are required to have tags and all builtin Spec factories will supply a default tag if one is not given.

2.2 Validation

Once you've *constructed* your Spec, you'll most likely want to begin validating data with that Spec. The *`dataspec.Spec`* interface provides several different ways to check that your data is valid given your use case.

The simplest way to validate your data is by calling *`dataspec.Spec.is_valid()`* which returns a simple boolean `True` if your data is valid and `False` otherwise. Of course, that kind of simple yes or no answer may be sufficient in some cases, but in other cases you may be more interested in knowing *exactly* why the data you provided is invalid. For more complex cases, you can turn to the generator *`dataspec.Spec.validate()`* which will emit successive *`dataspec.ErrorDetails`* instances describing the errors in your input value.

`dataspec.ErrorDetails` instances include comprehensive details about why your input data did not meet the Spec, including an error message, the predicate that validated it, and the value itself. *`via`* is a list of all Spec tags that validated your data up to (and including) the error. For nested values, the *`path`* attribute indicates the indices and keys that lead from the input value to the failing value. This detail can be used to programmatically emit useful error messages to clients.

Note: For convenience, you can fetch all of the errors at once as a list using *`dataspec.Spec.validate_all()`* or raise an exception with all of the errors using *`dataspec.Spec.validate_ex()`*.

Warning: `dataspec` will emit an exhaustive list of every instance where your input data fails to meet the Spec, so if you do not require a full list of errors, you may want to consider using *`dataspec.Spec.is_valid()`* or using the generator method *`dataspec.Spec.validate()`* to fetch errors as needed.

2.3 Conformation

Data validation is only one half of the value proposition for using `dataspec`. After you've validated that data is valid, the next step is to normalize it into a canonical format. Conformers are functions of one argument that can accept a validated value and emit a canonical representation of that value. Conformation is the component of `dataspec` that helps you normalize data.

Every Spec value comes with a default conformer. For most Specs, that conformer simply returns the value it was passed, though a few builtin Specs do provide a richer, canonicalized version of the input data. For example, *s.date()* conforms a date (possibly from a `strptime` format string) into a `date` object. Note that **none** of the builtin Spec conformers ever modify the data they are passed. `dataspec` conformers always create new data structures and return the conformed values. Custom conformers can modify their data in-flight, but that is not recommended since it will be harder reason about failures (in particular, if a mutating conformer appeared in the middle of `s.all(...)` Spec and a later Spec produced an error).

Most common Spec workflows will involve validating that your data is, in fact, valid using *dataspec.Spec.is_valid()* or *dataspec.Spec.validate()* for richer error details and then calling *dataspec.Spec.conform_valid()* if it is valid or dealing with the error if not.

2.3.1 User Provided Conformers

When you create Specs, you can always provide a conformer using the `conformer` keyword argument. This function will be called any time you call *dataspec.Spec.conform()* on your Spec or any Spec your Spec is a part of. The `conformer` keyword argument for *s()* and other builtin factories will always apply your conformer as by *dataspec.Spec.compose_conformer()* , rather than replacing the default conformer. To have your conformer *completely* replace the default conformer (if one is provided), you can use the *dataspec.Spec.with_conformer()* method on the returned Spec.

2.4 Predicate and Validators

You can define a spec using any simple predicate you may have by passing the predicate directly to the *s()* function, since not every valid state of your data can be specified using existing specs.

```
spec = s(lambda id_: uuid.UUID(id_).version == 4)
spec.is_valid("4716df50-0aa0-4b7d-98a4-1f2b2bcb1c6b")  # True
spec.is_valid("b4e9735a-ee8c-11e9-8708-4c327592fea9")  # False
```

Simple predicates make fine specs, but are unable to provide more details to the caller about exactly why the input value failed to validate. Validator specs directly yield *dataspec.ErrorDetails* objects which can indicate more precisely why the input data is failing to validate.

```
def _is_positive_int(v: Any) -> Iterable[ErrorDetails]:
    if not isinstance(v, int):
        yield ErrorDetails(
            message="Value must be an integer", pred=_is_positive_int, value=v
        )
    elif v < 1:
        yield ErrorDetails(
            message="Number must be greater than 0", pred=_is_positive_int, value=v
        )

spec = s(_is_positive_int)
spec.is_valid(5)      # True
```

(continues on next page)

(continued from previous page)

```
spec.is_valid(0.5)     # False
spec.validate_ex(-1)   # ValidationError(errors=[ErrorDetails(message="Number must be␣
↪greater than 0", ...)])
```

Simple predicates can be converted into validator functions using the builtin *dataspec.pred_to_validator()* decorator:

```
@pred_to_validator("Number must be greater than 0")
def _is_positive_num(v: Union[int, float]) -> bool:
    return v > 0

spec = s(_is_positive_num)
spec.is_valid(5)       # True
spec.is_valid(0.5)     # True
spec.validate_ex(-1)   # ValidationError(errors=[ErrorDetails(message="Number must be␣
↪greater than 0", ...)])
```

2.5 Type Specs

You can define a Spec that validates input values are instances of specific class types by simply passing a Python type directly to the *s()* constructor:

```
spec = s(str)
spec.is_valid("a string")   # True
spec.is_valid(3)            # False
```

Note: `s(None)` is a shortcut for `s(type(None))`.

2.6 Factories

The `s` API also includes several Spec factories for common Python types such as *bool*, *bytes*, *date*, *datetime* (via `s.inst()`), *float* (via `s.num()`), *int* (via `s.num()`), *str*, *time*, and *uuid*.

s also includes several pre-built Specs for basic types which are useful if you only want to verify that a value is of a specific type. All the pre-built Specs are supplied as *s.is_{type}* on `s`. You can generate a more generic type-checking spec using *Type Specs*.

2.6.1 String Specs

You can create a spec which validates strings with *s.str()*. Common string validations can be specified as keyword arguments, such as the min/max length or a matching regex. If you are only interested in validating that a value is a string without any further validations, spec features the predefined spec `s.is_str` (note no function call required).

2.6.2 Numeric Specs

Likewise, numeric specs can be created using *s.num()*, with several builtin validations available as keyword arguments such as min/max value and narrowing down the specific numeric types. If you are only interested in validating that a value is numeric, you can use the builtin `s.is_num` or `s.is_int` or `s.is_float` specs.

2.6.3 UUID Specs

In a previous section, we used a simple predicate to check that a UUID was a certain version of an RFC 4122 variant UUID. However, `dataspec` includes the builtin UUID spec factory *`s.uuid()`* which can simplify the logic here:

```
spec = s.uuid(versions={4})
spec.is_valid("4716df50-0aa0-4b7d-98a4-1f2b2bcb1c6b")  # True
spec.is_valid("b4e9735a-ee8c-11e9-8708-4c327592fea9")  # False
```

Additionally, if you are only interested in validating that a value is a UUID, the builting spec `s.is_uuid` is available.

2.6.4 Time and Date Specs

`dataspec` includes some builtin Specs for Python's `datetime`, `date`, and `time` classes. With the builtin specs, you can validate that any of these three class types are before or after a given. Suppose you want to verify that someone is 18 by checking their date of birth:

```
spec = s.date(after=date.today() - timedelta(years=18))
spec.is_valid(date.today() - timedelta(years=21))  # True
spec.is_valid(date.today() - timedelta(years=12))  # False
```

For datetimes (instants) and times, you can also use `is_aware=True` to specify that the instance be timezone-aware (e.g. not naive).

You can use the builtins `s.is_date`, `s.is_inst`, and `s.is_time` if you only want to validate that a value is an instance of any of those classes.

Note: `dataspec` supports specs for arbitrary date strings if you have `python-dateutil` installed. See *`s.inst_str()`* for info.

2.6.5 Phone Number Specs

`dataspec` supports creating Specs for validating telephone numbers from strings using *`s.phone()` if you have the* phonenumbers *library installed*. Telephone number Specs can validate that a telephone number is merely formatted correctly or they can validate that a telephone number is both possible and valid (via `phonenumbers`).

```
spec = s.phone(region="US")
spec.is_valid("(212) 867-5309")  # True
spec.conform("(212) 867-5309")   # "+12128675309"
spec.is_valid("(22) 867-5309")   # False
```

2.6.6 Email Address and URL Specs

`dataspec` features Spec factories for validating email addresses using *`s.email()`* and URLs using *`s.url()`*.

Email addresses are validated using Python's builtin `email.headerregistry.Address` class to parse email addresses into username and domain. For each of `username` and `domain` , you may validate that the value is an exact match, is one of a set of possible matches, or that it matches a regex pattern. To produce a Spec which only validates email addresses from `gmail.com` or `googlemail.com`:

```
spec = s.email(domain_in={"gmail.com", "googlemail.com"})
spec = s.email(domain_regex=r"(gmail|googlemail)\.com")
spec = s.email(domain="gmail.com")  # Don't allow "googlemail.com" email addresses
```

No more than one keyword filter may be supplied for either of `username` or `domain`.

URLs are validated using Python's builtin `urllib` module to parse URLs into their constituent components: `scheme` , `netloc` , `path` , `params` , `fragment` , `username` , `password` , `hostname`, and `port`. URL Specs may optionally provide a Spec for the `dict` created by parsing the query-string (if present) for the URL. Specs for each of the components of a URL allow the same filters as described above for email addresses. For more information, see *`s.url()`*.

2.7 Enumeration (Set) Specs

Commonly, you may be interested in validating that a value is one of a constrained set of known values. In Python code, you would use an `Enum` type to model these values. To define an enumermation spec, you can pass an existing `Enum` value into *`dataspec.s()`* :

```
class YesNo(Enum):
    YES = "Yes"
    NO = "No"

s(YesNo).is_valid("Yes")    # True
s(YesNo).is_valid("Maybe")  # False
```

Any valid representation of the `Enum` value would satisfy the spec, including the value, alias, and actual `Enum` value (like `YesNo.NO`).

Additionally, for simpler cases you can specify an enum using Python `set` s (or `frozenset` s):

```
s({"Yes", "No"}).is_valid("Yes")    # True
s({"Yes", "No"}).is_valid("Maybe")  # False
```

2.8 Collection Specs

Specs can be defined for values in homogenous collections as well. Define a spec for a homogenous collection as a list passed to *`dataspec.s()`* with the first element as the Spec for collection elements:

```
s([s.num(min_=0)]).is_valid([1, 2, 3, 4])  # True
s([s.num(min_=0)]).is_valid([-11, 2, 3])   # False
```

You may also want to assert certain conditions that apply to the collection as a whole. `dataspec` allows you to specify an *optional* dictionary as the second element of the list with a few possible rules applying to the collection as a whole, such as length and collection type.

```
s([s.num(min_=0), {"kind": list}]).is_valid([1, 2, 3, 4])  # True
s([s.num(min_=0), {"kind": list}]).is_valid({1, 2, 3, 4})  # False
```

Collection specs conform input collections by applying the element conformer(s) to each element of the input collection. Callers can specify an `"into"` key in the collection options dictionary as part of the spec to specify which type of collection is emitted by the collection spec default conformer. Collection specs which do not specify the `"into"` collection type will conform collections into the same type as the input collection.

2.9 Mapping Specs

Specs can be defined for mapping/associative types and objects. To define a spec for a mapping type, pass a dictionary of specs to `s`. The keys should be the expected key value (most often a string) and the value should be the spec for values located in that key. If a mapping spec contains a key, the spec considers that key *required*. To specify an *optional* key in the spec, wrap the key in *`s.opt()`*. Optional keys will be validated if they are present, but allow the map to exclude those keys without being considered invalid.

```
s(
    {
        "id": s.str("id", format_="uuid"),
        "first_name": s.str("first_name"),
        "last_name": s.str("last_name"),
        "date_of_birth": s.str("date_of_birth", format_="iso-date"),
        "gender": s("gender", {"M", "F"}),
        s.opt("state"): s("state", {"CA", "GA", "NY"}),
    }
)
```

Above the key `"state"` is optional in tested values, but if it is provided it must be one of `"CA"`, `"GA"`, or `"NY"`.

Note: Mapping specs do not validate that input values *only* contain the expected set of keys. Extra keys will be ignored. This is intentional behavior.

Note: To apply the mapping Spec key as the tag of the value Spec, use *`s.dict_tag()`* to construct your mapping Spec. For more precise control over the value Spec tags, prefer *`s()`*.

Mapping specs conform input dictionaries by applying each field's conformer(s) to the fields of the input map to return a new dictionary. As a consequence, the value returned by the mapping spec default conformer will not include any extra keys included in the input. Optional keys will be included in the conformed value if they appear in the input map.

2.9.1 Merging Mapping Specs

Occasionally, you may wish to declare your mapping Specs across two or more different Specs. It may be convenient to do so for composition of common keys across multiple Specs. In such cases, you may naturally turn to one of the builtin *Combination Specs* to return a union of the input Specs. However, combination Specs composed of mapping Specs with disjoint or only partially intersecting key sets will end up producing unexpected results. Recall mapping Specs have a default conformer which drops keys not declared in the input Spec, so the chained conformation of *`s.all()`* will drop keys potentially expected by later Specs.

To merge mapping Specs, use *`s.merge()`* instead.

```
s.merge(
    {"id": int},
    {
        "id": lambda v: v > 0,
        "first_name": str,
        s.opt("middle_initial"): str,
        "last_name": str,
    },
)
```

In the above Spec, `id` would be a required key, which must be an integer greater than zero. Specs for the remaining keys would match the Spec defined in the second input Spec.

Note: Only mapping Specs may be merged. `s.merge` will throw a `ValueError` if you attempt to merge non-mapping type Specs. To combine mapping and non-mapping Spec types, you should wrap the mapping Specs with `s.merge` and pass that to `s.all`.

2.9.2 Key/Value Specs

Mapping Specs are useful for heterogeneous associative data structures for which the keys are known *a priori*. However, you may often wish to validate a homogeneous mapping with unknown keys. For such cases, you can turn to *`s.kv()`*.

```
spec = s.kv(s.str(regex=r"[A-Z]{2}"), s.str(regex=r"[A-Z][\w ]+"))
spec.is_valid({"GA": "Georgia", "NM": "New Mexico"})  # True
spec.is_valid({"ga": "Georgia", "NM": "New Mexico"})  # False
spec.is_valid({"ga": "Georgia", "NM": "new mexico"})  # False
```

Note: By default *`s.kv`* will not conform keys on input values, to avoid potential creating potentially duplicate keys from the key conformer. You can override this behavior with the `conform_keys` keyword argument.

2.10 Tuple Specs

Specs can be defined for heterogenous collections of elements, which is often the use case for Python's `tuple` type. To define a spec for a tuple, pass a tuple of specs for each element in the collection at the corresponding tuple index:

```
s(
    (
        s.str("id", format_="uuid"),
        s.str("first_name"),
        s.str("last_name"),
        s.str("date_of_birth", format_="iso-date"),
        s("gender", {"M", "F"}),
    )
)
```

Tuple specs conform input tuples by applying each field's conformer(s) to the fields of the input tuple to return a new tuple. If each field in the tuple spec has a unique tag and the tuple has a custom tag specified, the default conformer will yield a `namedtuple` with the tuple spec tag as the type name and the field spec tags as each field name. The type name and field names will be munged to be valid Python identifiers.

2.11 Combination Specs

In most of the previous examples, we used basic builtin Specs. However, real world data often more nuanced specifications for data. Fortunately, Specs were designed to be composed. In particular, Specs can be composed using standard boolean logic. To specify an `or` spec, you can use *`s.any()`* with any `n` specs.

```
spec = s.any(s.str(format_="uuid"), s.str(maxlength=0))
spec.is_valid("4716df50-0aa0-4b7d-98a4-1f2b2bcb1c6b")  # True
spec.is_valid("")            # True
spec.is_valid("3837273723")  # False
```

Similarly, to specify an `and` spec, you can use *`s.all()`* with any `n` specs:

```
spec = s.all(s.str(format_="uuid"), s(lambda id_: uuid.UUID(id_).version == 4))
spec.is_valid("4716df50-0aa0-4b7d-98a4-1f2b2bcb1c6b")  # True
spec.is_valid("b4e9735a-ee8c-11e9-8708-4c327592fea9")  # False
```

Note: `and` Specs apply each child Spec's conformer to the value during validation, so you may assume the output of the previous Spec's conformer in subsequent Specs.

Note: The names `any` and `all` were chosen because `or` and `and` are not valid Python since they are reserved keywords.

Warning: Using a *`s.all()`* Spec to combine mapping Specs for maps with disjoint or only partially intersecting keys will result in maps losing keys during conformation and failing validation in later Specs. Use *`s.merge()`* to combine mapping Specs. Read more in *Merging Mapping Specs*.

2.12 Utility Specs

Often when dealing with real world data, you may wish to allow certain values to be blank or `None`. We *could* handle these cases with *Combination Specs*, but since they occur so commonly, `dataspec` features a couple of utility Specs for quickly defining these cases. For cases where `None` is a valid value, you can wrap your Spec with *`s.nilable()`*. If you are dealing with strings and need to allow a blank value (as is often the case when handling CSVs), you can wrap your Spec with *`s.blankable`*.

```
spec = s.nilable("birth_date", s.str(format_="iso-date"))
spec.is_valid(None)          # True
spec.is_valid("1980-09-14")  # True
spec.is_valid("")            # False
spec.is_valid("09/14/1980")  # False, because the string is not ISO formatted

spec = s.blankable("birth_date", s.str(format_="iso-date"))
spec.is_valid(None)          # False
spec.is_valid("1980-09-14")  # True
spec.is_valid("")            # True
spec.is_valid("09/14/1980")  # False
```

In certain cases, you may be willing to accept invalid data and overwrite it with a default value during conformation. For such cases, you can specify a default value whenever the input value does not pass validation for another spec using *`s.default`*. The value supplied to the `default` keyword argument will be provided by the conformer if the inner Spec does not validate.

```
spec = s.default("birth_date_or_none", s.str(format=_"iso-date"), default=None)
spec.is_valid(None)          # True; conforms to None
```

(continues on next page)

(continued from previous page)

```
spec.is_valid("1980-09-14")  # True; conforms to "1980-09-14"
spec.is_valid("")            # True; conforms to None
spec.is_valid("09/14/1980")  # True; conforms to None
```

Note: As a consequence of the default value, `s.default(...)` Specs consider every value valid. If you do not want to permit all values to pass, you should not use `s.default`.

Occasionally, it may be useful to allow any value to pass validation. For these cases *`s.every()`* is perfect.

Note: You may want to combine `s.every(...)` with `s.all(...)` to perform a pre- conformation step prior to later steps. In this case, it may still be useful to provide a slightly more strict validation to ensure your conformer does not throw an exception.

CHAPTER 3

Concepts and Patterns

Contents

3.1 Concepts

3.1.1 Composition

Specs are designed to be composed, so each of the builtin spec types can serve as the base for more complex data definitions. For collection, mapping, and tuple type Specs, Specs will be recursively created for child elements if they are types understood by `s()`. Specs can be composed using boolean logic with `s.all()` and `s.any()`. Many of the builtin factories accept existing specs or values which can be coerced to specs. With Dataspec, you can easily start speccing out your code and gradually add new specs and build off of existing specs as your app evolves.

3.1.2 Predicates

Predicates are functions of one argument which return a boolean. Predicates answer questions such as "is `x` an instance of `str`?" or "is `n` greater than `0`?". Frequently in Python, predicates are simply expressions used in an `if` statement. In functional programming languages (and particularly in Lisps), it is more common to encode these predicates in functions which can be combined using lambdas or partials to be reused. Spec encourages that functional paradigm and benefits directly from it.

Predicate functions should satisfy the *`dataspec.PredicateFn`* type and will be wrapped in the `PredicateSpec` spec type.

3.1.3 Validators

Validators are like predicates in that they answer the same fundamental questions about data that predicates do. However, Validators are a Spec concept that allow us to retrieve richer error data from Spec failures than we can natively with a simple predicate. Validators are functions of one argument which return 0 or more `ErrorDetails` instances (typically `yield` -ed as a generator) describing the error.

Validator functions should satisfy the *`dataspec.ValidatorFn`* type and will be wrapped in the `ValidatorSpec` spec type.

3.1.4 Conformers

Conformers are functions of one argument, `x`, that return either a conformed value, which may be `x` itself, a new value based on `x`, or an object of type *`Invalid`* if the value cannot be conformed. Builtin specs typically return the constant `INVALID`, which allows for a quick identity check (via the `is` operator) in many cases.

All specs may include conformers. Scalar spec types such as `PredicateSpec` and `ValidatorSpec` simply return their argument if it satisfies the spec. Specs for more complex data structures supply a default conformer which produce new data structures after applying any child conformation functions to the data structure elements.

3.1.5 Tags

Tags are simple string names for specs. Tags most often appear in *`ErrorDetails`* objects when an input value cannot be validated indicating the spec or specs which failed. This is useful for both debugging and producing useful user-facing validation messages. All Specs can be created with custom tags, which are specified as a string in the first positional argument of any spec creation function. Callers are not required to provide tags, but tags are *required* on Spec instances so `dataspec` provides a default value for all builtin spec types.

3.2 Patterns

3.2.1 Factories

Often when validating documents such as a CSV or a JSON blob, you'll find yourself writing a series of similar specs again and again. In situations like these, it is recommended to create a factory function for generating specs consistently. `dataspec` uses this pattern for many of the common spec types described above. This encourages reuse of commonly used specs and should help enforce consistency across your domain.

Note: If nothing changes between definitions, then consider defining your Spec at the module level instead. Spec instances are immutable and stateless, so they only need to be defined once.

3.2.2 Reuse

Specs are designed to be immutable and stateless, so they may be reused across many different contexts. Often, the only thing that changes between uses is the tag or conformer. Specs provide a convenient API for generating copies of themselves with new tags and conformers. You can even generate new specs with a composition of the existing spec's conformer. The API for creating new copies of specs always returns new copies, leaving the existing spec unmodified, so you can safely create copies of specs with slight tweaks without fear of unexpected modification.

In an application setting, it may make sense to collocate your common specs in a single sub-module or sub-package so they can be easily referred to from other parts of the application. We typically do not recommend `CONSTANT_CASE` for module-level specs, since there tend to be quite a few of them and the all-caps names are more challenging to skim.

CHAPTER 4

Dataspec API

Contents

4.1 Creating Specs

dataspec.**s**(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

`dataspec.s` is a singleton of *`dataspec.SpecAPI`* which can be imported and used directly as a generic *`dataspec.Spec`* constructor.

For more information, see *`dataspec.SpecAPI.__call__()`*.

class dataspec.**SpecAPI**

__call__(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Create a new Spec instance from a `dataspec.base.SpecPredicate`.

Specs may be created from a variety of functions. Functions which take a single argument and return a boolean value can produce simple Specs. For more detailed error messages, callers can provide a function which takes a single argument and yields consecutive `ErrorDetails` (in particular, the return annotation should be *exactly* `Iterator[ErrorDetails]`).

Specs may be created from Python types, in which case a Spec will be produced that performs an `isinstance()` check. `None` may be provided as a shortcut for `type(None)`. To specify a nilable value, you should use *dataspec.SpecAPI.nilable()* instead.

Specs may be created for enumerated types using a Python `set` or `frozenset` or using Python `enum.Enum` types. Specs created for enumerated types based on `enum.Enum` values validate the Enum name, value, or Enum singleton and conform the input value to the corresponding `enum.Enum` value.

Specs may be created for homogeneous collections using a Python `list` type. Callers can specify a few additional parameters for collection specs by providing an optional dictionary of values in the second position of the input `list`. To validate the input collection type, provide the `"kind"` key with a collection type. To specify the output type used by the default conformer, provide the `"into"` keyword with a collection type.

Specs may be created for mapping types using a Python `dict` type. The input `dict` maps key values (most often strings) to Specs (or values which can be coerced to Specs by this function). Mapping Specs validate that an input map contains the required keys and that the value associated with the key matches the given Spec. Mapping specs can be specified with optional keys by wrapping the optional key with `s.opt`. If that key is present in the input value, it will be validated against the given Spec. However, if the input value does not contain the optional key, the map is still considered valid. Mapping Specs do not assert that input values contain *only* the keys given in the Spec – this is by design.

Specs may be created for heterogeneous collections using a Python `tuple` type. Tuple Specs will conform into `collections.NamedTuple` s, with each element in the input tuple being validated and conformed to the corresponding element in the Spec.

Specs may be be created from existing Specs. If an existing `datspec.Spec` instance is given, that Spec will be returned without modification. If a tag is given, a new Spec will be created from the existing Spec with the new tag. If a conformer is given, a new Spec will be created from the existing Spec with the new conformer (*replacing* any conformer on the existing Spec, rather than composing). If both a new tag and conformer are given, a new Spec will be returned with both the new tag and conformer.

Parameters

- **`tag_or_pred`** – an optional tag for the resulting spec or a Spec or value which can be converted into a Spec; if no tag is provided, the default depends on the input type:
  - for `frozenset` and `set` predicates, the default is `"set"`
  - for `Enum` predicates, the default is the name of the enum
  - for `tuple` predicates, the default is `"tuple"`
  - for `list` (collection) predicates, the default is `"coll"`
  - for `dict` (mapping) predicates, the default is `"map"`
  - for `type` predicates, the default is the name of the type
  - for callable predicates, the default is the name of the function
- **`preds`** – if a tag is given, exactly one spec predicate; if no tag is given, this should not be specified
- **`conformer`** – an optional *dataspec.Conformer* for the value

Returns a `dataspec.base.Spec` instance

static all(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which validates input values against all of the input Specs or spec predicates.

For each Spec for which the input value is successfully validated, the value is successively passed to the Spec's *dataspec.Spec.conform_valid()* method.

The returned Spec's *dataspec.Spec.validate()* method will emit a stream of `dataspec.ErrorDetails` from the first failing constituent Spec. *dataspec.ErrorDetails* emitted from Specs after a failing Spec will not be emitted, because the failing Spec's `dataspec.Spec.conform()` would not successfully conform the value.

The returned Spec's *dataspec.Spec.conform()* method is the composition of all of the input Spec's `conform` methods.

If no Specs or Spec predicates are given, a `ValueError` will be raised. If only one Spec or Spec predicate is provided, it will be passed to *dataspec.s()* with the given `tag` and `conformer` and the value returned without merging.

This method is not suitable for producing a union of mapping Specs. To merge mapping Specs, use *dataspec.SpecAPI.merge()* instead.

Parameters

- **tag_or_pred** – an optional tag for the resulting spec or the first Spec or value which can be converted into a Spec; if no tag is provided, the default is `"all"`
- **preds** – zero or more Specs or values which can be converted into a Spec
- **conformer** – an optional conformer which will be applied to the final conformed value produced by the input Specs conformers

Returns a Spec

static any(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, tag_conformed: bool = False, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which validates input values against any one of an arbitrary number of input Specs.

The returned Spec validates input values against the input Specs in the order they are passed into this function.

If the returned Spec fails to validate the input value, the *dataspec.Spec.validate()* method will emit a stream of *dataspec.ErrorDetails* from all of failing constituent Specs. If any of the constituent Specs successfully validates the input value, then no *dataspec.ErrorDetails* will be emitted by the *dataspec.Spec.validate()* method.

The conformer for the returned Spec will select the conformer for the first constituent Spec which successfully validates the input value. If a `conformer` is specified for this Spec, that conformer will be applied after the successful Spec's conformer. If `tag_conformed` is specified, the final conformed value from both conformers will be wrapped in a tuple, where the first element is the tag of the successful Spec and the second element is the final conformed value. If `tag_conformed` is not specified (which is the default), the conformer will emit the conformed value directly.

If no Specs or Spec predicates are given, a `ValueError` will be raised. If only one Spec or Spec predicate is provided, it will be passed to *`dataspec.s()`* with the given `tag` and `conformer` and the value returned without merging.

Parameters

- **tag_or_pred** – an optional tag for the resulting spec or the first Spec or value which can be converted into a Spec; if no tag is provided, the default is `"any"`
- **preds** – zero or more Specs or values which can be converted into a Spec
- **tag_conformed** – if `True`, the conformed value will be wrapped in a 2-tuple where the first element is the successful spec and the second element is the conformed value; if `False`, return only the conformed value
- **conformer** – an optional conformer for the value

Returns a Spec

static blankable(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*)

Return a Spec which will validate values either by the input Spec or allow the empty string.

The returned Spec is roughly equivalent to `s.any(spec, {""})`.

If no Specs or Spec predicates is given, a `ValueError` will be raised.

Parameters

- **tag_or_pred** – an optional tag for the resulting Spec *or* a Spec or value which can be converted into a Spec; if no tag is provided, the default is `"blankable"`
- **preds** – if a tag is provided for `tag_or_pred`, exactly Spec predicate as described in `tag_or_pred`; otherwise, nothing
- **conformer** – an optional conformer for the value

Returns a Spec which validates either according to `pred` or the empty string

static bool(*tag: str = 'bool', allowed_values: Optional[Set[bool]] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which will validate boolean values.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"bool"`
- **allowed_values** – if specified, a set of allowed boolean values
- **conformer** – an optional conformer for the value

Returns a Spec which validates boolean values

static bytes(*tag: str = 'bytes', type_: Tuple[Union[Type[bytes], Type[bytearray]], ...] = (<class 'bytes'>, <class 'bytearray'>), length: Optional[int] = None, minlength: Optional[int] = None, maxlength: Optional[int] = None, regex: Union[Pattern[AnyStr], bytes, None] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a spec that can validate bytes and bytearrays against common rules.

If `type_` is specified, the resulting Spec will only validate the byte type or types named by `type_`, otherwise `byte` and `bytearray` will be used.

If `length` is specified, the resulting Spec will validate that input bytes measure exactly `length` bytes by by `len()`. If `minlength` is specified, the resulting Spec will validate that input bytes measure at least `minlength` bytes by by `len()`. If `maxlength` is specified, the resulting Spec will validate that input bytes measure not more than `maxlength` bytes by by `len()`. Only one of `length`, `minlength`, or `maxlength` can be specified. If more than one is specified a `ValueError` will be raised. If any length value is specified less than 0 a `ValueError` will be raised. If any length value is not an `int` a `TypeError` will be raised.

If `regex` is specified and is a *bytes*, a Regex pattern will be created by `re.compile()`. If `regex` is specified and is a `typing.Pattern`, the supplied pattern will be used. In both cases, the `re.fullmatch()` will be used to validate input strings.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"bytes"`
- **type** – a single `type` or tuple of `type` s which will be used to type check input values by the resulting Spec
- **length** – if specified, the resulting Spec will validate that bytes are exactly `length` bytes long by `len()`
- **minlength** – if specified, the resulting Spec will validate that bytes are not fewer than `minlength` bytes long by `len()`
- **maxlength** – if specified, the resulting Spec will validate that bytes are not longer than `maxlength` bytes long by `len()`
- **regex** – if specified, the resulting Spec will validate that strings match the `regex` pattern using `re.fullmatch()`
- **conformer** – an optional conformer for the value

Returns a Spec which validates bytes and bytearrays

static date(*tag: str = 'date', format_: Optional[str] = None, before: Optional[datetime.date] = None, after: Optional[datetime.date] = None, is_aware: Optional[bool] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which validates `datetime.date` types with common rules.

If `format_` is specified, the resulting Spec will accept string values and attempt to coerce them to `datetime.date` instances first before applying the other specified validations. If the `datetime.datetime` object parsed from the `format_` string contains a portion not available in `datetime.date`, then the validator will emit an error at runtime.

If `before` is specified, the resulting Spec will validate that input values are before `before` by Python's `<` operator. If `after` is specified, the resulting Spec will validate that input values are after `after` by Python's `>` operator. If `before` and `after` are specified and `after` is before `before`, a `ValueError` will be raised.

If `is_aware` is specified, a `TypeError` will be raised as `datetime.date` values cannot be aware or naive.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"date"`
- **format** – if specified, a time format string which will be fed to `datetime.date.strptime()` to convert the input string to a `datetime.date` before applying the other validations
- **before** – if specified, the input value must come before this date or time

- **after** – if specified, the input value must come after this date or time
- **is_aware** – if `True` , validate that input objects are timezone aware; if `False` , validate that input objects are naive; if `None`, do not consider whether the input value is naive or aware
- **conformer** – an optional conformer for the value; if the `format_` parameter is supplied, the conformer will be passed a `datetime.date` value, rather than a string

Returns a Spec which validates `datetime.date` types

static default(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, default: Any = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which will validate every value, but which will conform values not meeting the Spec to a default value.

The returned Spec is equivalent to the following Spec:

```
s.any(spec, s.every(conformer=lambda _:  default))
```

This Spec **will allow any value to pass**, but will conform to the given default if the data does not satisfy the input Spec.

If no Specs or Spec predicates is given, a `ValueError` will be raised.

Parameters

- **tag_or_pred** – an optional tag for the resulting Spec *or* a Spec or value which can be converted into a Spec; if no tag is provided, the default is `"default"`
- **preds** – if a tag is provided for `tag_or_pred`, exactly Spec predicate as described in `tag_or_pred`; otherwise, nothing
- **default** – the default value to apply if the Spec does not validate a value
- **conformer** – an optional conformer for the value

Returns a Spec which validates every value, but which conforms values to a default

static dict_tag(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a mapping Spec for which the Tags for each of the `dict` values is set to the corresponding key.

This is a convenience factory for the common pattern of creating a mapping Spec with all of the key Specs' Tags bearing the same name as the corresponding key. The value Specs are created as by *dataspec.s*, so existing Specs will not be modified; instead new Specs will be created by *dataspec.Spec.with_tag()*.

For more precise tagging of mapping Spec values, use the default `s` constructor with a `dict` value.

Parameters

- **tag_or_pred** – an optional tag for the resulting spec or the first Spec or value which can be converted into a Spec; if no tag is provided, default is `"map"`

- **preds** – if a tag is given, exactly one mapping spec predicate; if no tag is given, this should not be specified
- **conformer** – an optional conformer for the value

Returns a mapping Spec

static email(*tag: str = 'email', conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None, **kwargs*) → dataspec.base.Spec

Return a spec that can validate strings containing email addresses.

Email string specs always verify that input values are strings and that they can be successfully parsed by `email.headerregistry.Address()`.

Other restrictions can be applied by passing any one of three different keyword arguments for any of the fields of `email.headerregistry.Address`. For example, to specify restrictions on the `username` field, you could use the following keywords:

- `domain` accepts any value (including `None`) and checks for an exact match of the keyword argument value
- `domain_in` takes a `set` or `frozenset` and validates that the *domain'* field is an exact match with one of the elements of the set
- `domain_regex` takes a *str*, creates a Regex pattern from that string, and validates that `domain` is a match (by `re.fullmatch()`) with the given pattern

The value `None` can be used for comparison in all cases, though the value `None` is never tolerated as a valid `username` or `domain` of an email address.

At most only one restriction can be applied to any given field for the `email.headerregistry.Address`. Specifying more than one restriction for a field will produce a `ValueError`.

Providing a keyword argument for a non-existent field of `email.headerregistry.Address` will produce a `ValueError`.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"email"`
- **username** – if specified, require an exact match for `username`
- **username_in** – if specified, require `username` to match at least one value in the set
- **username_regex** – if specified, require `username` to match the regex pattern
- **domain** – if specified, require an exact match for `domain`
- **domain_in** – if specified, require `domain` to match at least one value in the set
- **domain_regex** – if specified, require `domain` to match the regex pattern
- **conformer** – an optional conformer for the value

Returns a Spec which can validate that a string contains an email address

static every(*tag: str = 'every', conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which validates every possible value.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"every"`
- **conformer** – an optional conformer for the value

Returns a Spec which validates any value

static explain(*spec: dataspec.base.Spec*, *v*) → Optional[dataspec.base.ValidationError]
Return a ValidationError instance containing all of the errors validating `v`, if there were any; return None otherwise.

static fdef(*argpreds: Tuple[Union[Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], ...] = (), kwargpreds: Optional[Mapping[str, Union[Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec]]] = None, retpred: Union[Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec, None] = None*)
Wrap a function `f` and validate its arguments, keyword arguments, and return value with Specs, if any are given.

static inst(*tag: str = 'datetime', format_: Optional[str] = None, before: Optional[datetime.datetime] = None, after: Optional[datetime.datetime] = None, is_aware: Optional[bool] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec
Return a Spec which validates `datetime.datetime` types with common rules.

If `format_` is specified, the resulting Spec will accept string values and attempt to coerce them to `datetime.datetime` instances first before applying the other specified validations.

If `before` is specified, the resulting Spec will validate that input values are before `before` by Python's < operator. If `after` is specified, the resulting Spec will validate that input values are after `after` by Python's > operator. If `before` and `after` are specified and `after` is before `before`, a `ValueError` will be raised.

If `is_aware` is `True` , the resulting Spec will validate that input values are timezone aware. If `is_aware` is `False` , the resulting Spec will validate that inpute values are naive. If unspecified, the resulting Spec will not consider whether the input value is naive or aware.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"datetime"`
- **format** – if specified, a time format string which will be fed to `datetime.datetime.strptime()` to convert the input string to a `datetime.datetime` before applying the other validations
- **before** – if specified, the input value must come before this date or time
- **after** – if specified, the input value must come after this date or time
- **is_aware** – if `True` , validate that input objects are timezone aware; if `False` , validate that input objects are naive; if `None`, do not consider whether the input value is naive or aware
- **conformer** – an optional conformer for the value; if the `format_` parameter is supplied, the conformer will be passed a `datetime.datetime` value, rather than a string

Returns a Spec which validates `datetime.datetime` types

static inst_str(*tag: str = 'datetime_str', iso_only: bool = False, before: Optional[datetime.datetime] = None, after: Optional[datetime.datetime] = None, is_aware: Optional[bool] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec that validates strings containing date/time strings in most common formats.

The resulting Spec will validate that the input value is a string which contains a date/time using `dateutil.parser.parse()`. If the input value can be determined to contain a valid `datetime.datetime` instance, it will be validated against a datetime Spec as by a standard `dataspec` datetime Spec using the keyword options below.

`dateutil.parser.parse()` cannot produce `datetime.time` or `datetime.date` instances directly, so this method will only produce `datetime.datetime()` instances even if the input string contains only a valid time or date, but not both.

If `iso_only` keyword argument is `True`, restrict the set of allowed input values to strings which contain ISO 8601 formatted strings. This is accomplished using `dateutil.parser.isoparse()`, which does not guarantee strict adherence to the ISO 8601 standard, but accepts a wider range of valid ISO 8601 strings than Python 3.7+'s `datetime.datetime.fromisoformat()` function.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"datetime_str"`
- **iso_only** – if True, restrict the set of allowed date strings to those formatted as ISO 8601 datetime strings; default is False
- **before** – if specified, a datetime that specifies the latest instant this Spec will validate
- **after** – if specified, a datetime that specifies the earliest instant this Spec will validate
- **is_aware** (*bool*) – if specified, indicate whether the Spec will validate either aware or naive `datetime.datetime` instances.
- **conformer** – an optional conformer for the value; if one is not provided `dateutil.parser.parse()` will be used

Returns a Spec which validates strings containing date/time strings

static kv(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conform_keys: bool = False, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec that validates mapping types against a single Spec for all keys and a single Spec for all values.

If `conform_keys` is specified as `True`, the default conformer will conform keys and values. By default, `conform_keys` is `False` to avoid duplicate names produced during the conformation.

The returned Spec's *`dataspec.Spec.conform()`* method will return a `dict` with values conformed by the corresponding input Spec. If a `conformer` is provided via keyword argument, that conformer will be provided a `dict` with the conformed `dict` as described above. Otherwise, the default conformer will simply return the conformed `dict` . Note that the default conformer does **not** modify the input mapping in place.

Exactly two Specs must be provided or a `ValueError` will be raised during construction.

Parameters

- **tag_or_pred** – an optional tag for the resulting spec or the key Spec or value which can be converted into a Spec
- **preds** – if a tag is given, preds should be exactly two Specs or values which can be converted into Specs; the first shall be the Spec for the keys and the second shall be the Spec for values

- **conform_keys** – if `True`, the default conformer will also conform keys according to the input key Spec; default is `False`
- **conformer** – an optional conformer which will be composed with the default conformer

Returns a Spec

static merge(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Merge two or more mapping Specs into a single new Spec.

The returned Spec validates input values against a mapping Spec which is created from the union of input mapping Specs. Mapping Specs will be merged in the order they are provided. Individual key Specs whose keys appear more than one input Spec will be merged as via *dataspec.SpecAPI.all()* in the order they are passed into this function.

If no Specs or Spec predicates are given, a `ValueError` will be raised. If only one Spec or Spec predicate is provided, it will be passed to *dataspec.s()* with the given `tag` and `conformer` and the value returned without merging. If any Specs or Spec predicates are provided which are not mapping Specs or which cannot be coerced to mapping Specs, a `TypeError` will be raised.

The returned Spec's *dataspec.Spec.conform()* method is a standard mapping Spec default conformer. Keys not defined in the union of key sets will be dropped during conformation. Values with more than one Spec defined in the input Specs will be conformed as by *dataspec.SpecAPI.all()* applied to all of their input Specs in the order they were provided. Values with exactly one Spec will use that Spec as given.

Parameters

- **tag_or_pred** – an optional tag for the resulting spec or the first Spec or value which can be converted into a Spec; if no tag is provided, the default is computed as `"merge-of-spec1-and-spec2-..."`
- **preds** – zero or more mapping Specs or values which can be converted into a mapping Spec
- **conformer** – an optional conformer for the value

Returns a single mapping Spec which is the union of all input Specs

static nilable(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which will validate values either by the input Spec or allow the value `None`.

The returned Spec is roughly equivalent to `s.any(spec, {None})`.

If no Specs or Spec predicates is given, a `ValueError` will be raised.

Parameters

- **tag_or_pred** – an optional tag for the resulting Spec *or* a Spec or value which can be converted into a Spec; if no tag is provided, the default is `"nilable"`
- **preds** – if a tag is provided for `tag_or_pred`, exactly one Spec predicate as described in `tag_or_pred`; otherwise, nothing
- **conformer** – an optional conformer for the value

Returns a Spec which validates either according to `pred` or the value `None`

static num(*tag: str = 'num', type_: Union[Type[CT_co], Tuple[Type[CT_co], ...]] = (<class 'float'>, <class 'int'>), min_: Union[complex, float, int, None] = None, max_: Union[complex, float, int, None] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec that can validate numeric values against common rules.

If `type_` is specified, the resulting Spec will only validate the numeric type or types named by `type_`, otherwise `float` and `int` will be used.

If `min_` is specified, the resulting Spec will validate that input values are at least `min_` using Python's < operator. If `max_` is specified, the resulting Spec will validate that input values are not more than `max_` using Python's < operator. If `min_` and `max_` are specified and `max_` is less than `min_`, a `ValueError` will be raised.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"num"`
- **type** – a single `type` or tuple of `type` s which will be used to type check input values by the resulting Spec
- **min** – if specified, the resulting Spec will validate that numeric values are not less than `min_` (as by <)
- **max** – if specified, the resulting Spec will validate that numeric values are not less than `max_` (as by >)
- **conformer** – an optional conformer for the value

Returns a Spec which validates numeric values

static obj(*tag_or_pred: Union[str, Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], *preds, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec for an arbitrary object.

Object Specs are defined as a mapping Spec with only string keys. The resulting Spec will validate arbitrary objects by calling `getattr()` on the input value with the mapping key names to validate the value contained on that attribute.

Object Specs support optional keys via *dataspec.SpecAPI.opt()*. The value must be a string.

Object Specs do not perform any type checks. Type checks can be defined separately by calling *dataspec.s()* with a type.

If no Specs or Spec predicates is given, a `ValueError` will be raised.

Parameters

- **tag_or_pred** – an optional tag for the resulting Spec *or* a mapping Spec predicate with string keys (potentially wrapped by *dataspec.SpecAPI.opt()*) and Spec predicates for values; if no tag is provided, the default is `"object"`
- **preds** – if a tag is provided for `tag_or_pred`, exactly one mapping Spec predicate as described in `tag_or_pred`; otherwise, nothing
- **conformer** – an optional conformer for the value

Returns a Spec which validates generic objects by their attributes

static opt(*k: T*) → dataspec.base.OptionalKey[~T][T]
Return `k` wrapped in a marker object indicating that the key is optional in associative specs.

static phone(*tag: str = 'phonenumber_str', region: Optional[str] = None, is_possible: bool = True, is_valid: bool = True, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec
Return a Spec that validates strings containing telephone number in most common formats.

The resulting Spec will validate that the input value is a string which contains a telephone number using `phonenumbers.parse()`. If the input value can be determined to contain a valid telephone number, it will be validated against a Spec which validates properties specified by the keyword arguments of this function.

If `region` is supplied, the region will be used as a hint for `phonenumbers.parse()` and the region of the parsed telephone number will be verified. Telephone numbers can be specified with their region as a "+" prefix, which takes precedence over the `region` hint. The Spec will reject parsed telephone numbers whose region differs from the specified region in all cases.

If `is_possible` is True, the parsed telephone number will be validated as a possible telephone number for the parsed region (which may be different from the specified region).

If `is_valid` is True, the parsed telephone number will be validated as a valid telephone number (as by `phonenumbers.is_valid_number()`).

By default, the Spec supplies a conformer which conforms telephone numbers to the international E.164 format, which is globally unique.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"phonenumber_str"`
- **region** – an optional two-letter country code which, if provided, will be checked against the parsed telephone number's region
- **is_possible** – if True and the input number can be successfully parsed, validate that the number is a possible number (it has the right number of digits)
- **is_valid** – if True and the input number can be successfully parsed, validate that the number is a valid number (it is an an assigned exchange)
- **conformer** – an optional conformer for the value; the conformer will be passed a `phonenumbers.PhoneNumber` object, rather than a string

Returns a Spec which validates strings containing telephone numbers

static str(*tag: str = 'str', length: Optional[int] = None, minlength: Optional[int] = None, maxlength: Optional[int] = None, regex: Union[Pattern[AnyStr], str, None] = None, format_: Optional[str] = None, conform_format: Optional[str] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec
Return a Spec that can validate strings against common rules.

String Specs always validate that the input value is a *str* type.

If `length` is specified, the resulting Spec will validate that input strings measure exactly `length` characters by by `len()`. If `minlength` is specified, the resulting Spec will validate that input strings measure at least `minlength` characters by by `len()`. If `maxlength` is specified, the resulting Spec will validate that input strings measure not more than `maxlength` characters by by `len()`. Only one of `length`, `minlength`, or `maxlength` can be specified. If more than one is specified a `ValueError` will be raised. If any length value is specified less than 0 a `ValueError` will be raised. If any length value is not an `int` a `TypeError` will be raised.

If `regex` is specified and is a *str*, a Regex pattern will be created by `re.compile()`. If `regex` is specified and is a `typing.Pattern`, the supplied pattern will be used. In both cases, the `re.`

`fullmatch()` will be used to validate input strings. If `format_` is specified, the input string will be validated using the Spec registered to validate for the string name of the format. If `conform_format` is specified, the input string will be validated using the Spec registered to validate for the string name of the format and the default conformer registered with the format Spec will be set as the `conformer` for the resulting Spec. Only one of `regex`, `format_`, and `conform_format` may be specified when creating a string Spec; if more than one is specified, a `ValueError` will be raised.

String format Specs may be registered using the function `dataspec.register_str_format_spec`()`. Alternatively, a string format validator function may be registered using the decorator `dataspec.register_str_format`()`. String formats may include a default conformer which will be applied for `conform_format` usages of the format.

Several useful defaults are supplied as part of this library:

- *iso-date* validates that a string contains a valid ISO 8601 date string
- *iso-datetime* (Python 3.7+) validates that a string contains a valid ISO 8601 date and time stamp
- *iso-time* (Python 3.7+) validates that a string contains a valid ISO 8601 time string
- `uuid` validates that a string contains a valid UUID

Parameters

- **`tag`** – an optional tag for the resulting spec; default is `"str"`
- **`length`** – if specified, the resulting Spec will validate that strings are exactly `length` characters long by `len()`
- **`minlength`** – if specified, the resulting Spec will validate that strings are not fewer than `minlength` characters long by `len()`
- **`maxlength`** – if specified, the resulting Spec will validate that strings are not longer than `maxlength` characters long by `len()`
- **`regex`** – if specified, the resulting Spec will validate that strings match the `regex` pattern using `re.fullmatch()`
- **`format`** – if specified, the resulting Spec will validate that strings match the registered string format `format`
- **`conform_format`** – if specified, the resulting Spec will validate that strings match the registered string format `conform_format`; the resulting Spec will automatically use the default conformer supplied with the string format
- **`conformer`** – an optional conformer for the value

Returns a Spec which validates strings

static time(*tag: str = 'time', format_: Optional[str] = None, before: Optional[datetime.time] = None, after: Optional[datetime.time] = None, is_aware: Optional[bool] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec which validates `datetime.time` types with common rules.

If `format_` is specified, the resulting Spec will accept string values and attempt to coerce them to `datetime.time` instances first before applying the other specified validations. If the `datetime.datetime` object parsed from the `format_` string contains a portion not available in `datetime.time`, then the validator will emit an error at runtime.

If `before` is specified, the resulting Spec will validate that input values are before `before` by Python's `<` operator. If `after` is specified, the resulting Spec will validate that input values are after `after` by

Python's > operator. If `before` and `after` are specified and `after` is before `before`, a `ValueError` will be raised.

If `is_aware` is `True` , the resulting Spec will validate that input values are timezone aware. If `is_aware` is `False` , the resulting Spec will validate that inpute values are naive. If unspecified, the resulting Spec will not consider whether the input value is naive or aware.

Parameters

- **`tag`** – an optional tag for the resulting spec; default is `"time"`
- **`format`** – if specified, a time format string which will be fed to `datetime.time.strptime()` to convert the input string to a `datetime.time` before applying the other validations
- **`before`** – if specified, the input value must come before this date or time
- **`after`** – if specified, the input value must come after this date or time
- **`is_aware`** – if `True` , validate that input objects are timezone aware; if `False` , validate that input objects are naive; if `None`, do not consider whether the input value is naive or aware
- **`conformer`** – an optional conformer for the value; if the `format_` parameter is supplied, the conformer will be passed a `datetime.time` value, rather than a string

Returns a Spec which validates `datetime.time` types

static url(*tag: str = 'url_str', query: Union[Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec, None] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None, **kwargs*) → dataspec.base.Spec

Return a spec that can validate URLs against common rules.

URL string specs always verify that input values are strings and that they can be successfully parsed by `urllib.parse.urlparse()`.

URL specs can specify a new or existing Spec or spec predicate value to validate the query string value produced by calling `urllib.parse.parse_qs()` on the `urllib.parse.ParseResult.query` attribute of the parsed URL result.

Other restrictions can be applied by passing any one of three different keyword arguments for any of the fields (excluding `urllib.parse.ParseResult.query`) of `urllib.parse.ParseResult`. For example, to specify restrictions on the `hostname` field, you could use the following keywords:

- `hostname` accepts any value (including `None`) and checks for an exact match of the keyword argument value
- `hostname_in` takes a :py:class:`set` or :py:class:`frozenset` and validates that the *hostname*' field is an exact match with one of the elements of the set
- `hostname_regex` takes a :py:class:`str`, creates a Regex pattern from that string, and validates that `hostname` is a match (by `re.fullmatch()`) with the given pattern

The value `None` can be used for comparison in all cases. Note that default the values for fields of `urllib.parse.ParseResult` vary by field, so using None may produce unexpected results.

At most only one restriction can be applied to any given field for the `urllib.parse.ParseResult`. Specifying more than one restriction for a field will produce a `ValueError`.

At least one restriction must be specified to create a URL string Spec. Attempting to create a URL Spec without specifying a restriction will produce a `ValueError`.

Providing a keyword argument for a non-existent field of `urllib.parse.ParseResult` will produce a `ValueError`.

Parameters

- **`tag`** – an optional tag for the resulting spec; default is `"url_str"`
- **`query`** – an optional spec for the `dict` created by calling `urllib.parse.parse_qs()` on the `urllib.parse.ParseResult.query` attribute of the parsed URL
- **`scheme`** – if specified, require an exact match for `scheme`
- **`scheme_in`** – if specified, require `scheme` to match at least one value in the set
- **`schema_regex`** – if specified, require `scheme` to match the regex pattern
- **`netloc`** – if specified, require an exact match for `netloc`
- **`netloc_in`** – if specified, require `netloc` to match at least one value in the set
- **`netloc_regex`** – if specified, require `netloc` to match the regex pattern
- **`path`** – if specified, require an exact match for `path`
- **`path_in`** – if specified, require `path` to match at least one value in the set
- **`path_regex`** – if specified, require `path` to match the regex pattern
- **`params`** – if specified, require an exact match for `params`
- **`params_in`** – if specified, require `params` to match at least one value in the set
- **`params_regex`** – if specified, require `params` to match the regex pattern
- **`fragment`** – if specified, require an exact match for `fragment`
- **`fragment_in`** – if specified, require `fragment` to match at least one value in the set
- **`fragment_regex`** – if specified, require `fragment` to match the regex pattern
- **`username`** – if specified, require an exact match for `username`
- **`username_in`** – if specified, require `username` to match at least one value in the set
- **`username_regex`** – if specified, require `username` to match the regex pattern
- **`password`** – if specified, require an exact match for `password`
- **`password_in`** – if specified, require `password` to match at least one value in the set
- **`password_regex`** – if specified, require `password` to match the regex pattern
- **`hostname`** – if specified, require an exact match for `hostname`
- **`hostname_in`** – if specified, require `hostname` to match at least one value in the set
- **`hostname_regex`** – if specified, require `hostname` to match the regex pattern
- **`port`** – if specified, require an exact match for `port`
- **`port_in`** – if specified, require `port` to match at least one value in the set
- **`conformer`** – an optional conformer for the value

Returns a Spec which can validate that a string contains a URL

static uuid(*tag: str = 'uuid', versions: Optional[Set[int]] = None, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → dataspec.base.Spec

Return a Spec that can validate UUIDs against common rules.

UUID Specs always validate that the input value is a `uuid.UUID` type.

If `versions` is specified, the resulting Spec will validate that input UUIDs are the RFC 4122 variant and that they are one of the specified integer versions of RFC 4122 variant UUIDs. If `versions` specifies an invalid RFC 4122 variant UUID version, a `ValueError` will be raised.

Parameters

- **tag** – an optional tag for the resulting spec; default is `"uuid"`
- **versions** – an optional set of integers of 1, 3, 4, and 5 which the input `uuid.UUID` must match; otherwise, any version will pass the Spec
- **conformer** – an optional conformer for the value

Returns a Spec which validates UUIDs

4.2 Types

class dataspec.**Spec**

The abstract base class of all Specs.

All Specs returned by *dataspec.s* conform to this interface.

compose_conformer(*conformer: Callable[[T], Union[V, dataspec.base.Invalid]]*) → dataspec.base.Spec

Return a new Spec instance with a new conformer which is the composition of the `conformer` and the current conformer for this Spec instance.

If the current Spec instance has a custom conformer, this is equivalent to calling `spec.with_conformer(lambda v: conformer(spec.conformer(v)))`. If the current Spec instance has no custom conformer, this is equivalent to calling *dataspec.Spec.with_conformer()* with `conformer`.

To completely replace the conformer for this Spec instance, use *dataspec.Spec.with_conformer()*.

This method does not modify the current Spec instance.

Parameters **conformer** – a conformer to compose with the conformer of the current Spec instance

Returns a copy of the current Spec instance with the new composed conformer

conform(*v: Any*)

Conform `v` to the Spec, returning the possibly conformed value or an instance of *dataspec.Invalid* if the value is invalid cannot be conformed.

Exceptions arising from calling *dataspec.Spec.conformer* with `v` will be raised from this method.

Parameters **v** – a value to conform

Returns a conformed value or a *dataspec.Invalid* instance if the input value could not be conformed

conform_valid(*v: Any*)

Conform `v` to the Spec without checking if `v` is valid first and return the possibly conformed value or `INVALID` if the value cannot be conformed.

This function should be used only if `v` has already been check for validity.

Exceptions arising from calling *dataspec.Spec.conformer* with `v` will be raised from this method.

Parameters **v** – a *validated* value to conform

Returns a conformed value or a *dataspec.Invalid* instance if the input value could not be conformed

conformer
Return the custom conformer attached to this Spec, if one is defined.

is_valid(*v: Any*) → bool
Returns `True` if `v` is valid according to the Spec, otherwise returns `False`.

Parameters **v** – a value to validate

Returns `True` if the value is valid according to the Spec, otherwise `False`

tag
Return the tag used to identify this Spec.

Tags are useful for debugging and in validation messages.

validate(*v: Any*) → Iterator[dataspec.base.ErrorDetails]
Validate the value `v` against the Spec, yielding successive Spec failures as *dataspec.ErrorDetails* instances, if any.

By definition, if `next(spec.validate(v))` raises `StopIteration`, the first time it is called, the value is considered valid according to the Spec.

Parameters **v** – a value to validate

Returns an iterator of Spec failures as *dataspec.ErrorDetails* instances, if any

validate_all(*v: Any*) → List[dataspec.base.ErrorDetails]
Validate the value `v` against the Spec, returning a `list` of all Spec failures of `v` as *dataspec.ErrorDetails* instances.

This method is equivalent to `list(spec.validate(v))`. If an empty list is returned `v` is valid according to the Spec.

Parameters **v** – a value to validate

Returns a list of Spec failures as *dataspec.ErrorDetails* instances, if any

validate_ex(*v: Any*) → None
Validate the value `v` against the Spec, throwing a *dataspec.ValidationError* containing a list of all of the Spec failures for `v` , if any. Returns `None` otherwise.

Parameters **v** – a value to validate

Returns `None`

with_conformer(*conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]]*) → dataspec.base.Spec
Return a new Spec instance with the new conformer, replacing any custom conformers.

If `conformer` is `None` , the returned Spec will have no custom conformer.

To return a copy of the current Spec with a composition of the current Spec instance, use *dataspec.Spec.compose_conformer()*.

Parameters **conformer** – a conformer to replace the conformer of the current Spec instance or `None` to remove the conformer associated with this

Returns a copy of the current Spec instance with new conformer

with_tag(*tag: str*) → dataspec.base.Spec
Return a new Spec instance with the new tag applied.

This method does not modify the current Spec instance.

Parameters **tag** – a new tag to use for the new Spec

Returns a copy of the current Spec instance with the new tag applied

dataspec.**SpecPredicate**
SpecPredicates are values that can be coerced into Specs by *dataspec.s()*.

dataspec.**Tag**
Tags are string names given to *dataspec.Spec* instances which are emitted in *dataspec.ErrorDetails* instances to indicate which Spec or Specs were evaluated to produce the error.

dataspec.**Conformer**
Conformers are functions of one argument which return either a conformed value or an instance of *dataspec.Invalid* (such as *dataspec.INVALID*).

dataspec.**PredicateFn**
Predicate functions are functions of one argument which return `bool` indicating whether or not the argument is valid or not.

dataspec.**ValidatorFn**
Validator functions are functions of one argument which yield successive *dataspec.ErrorDetails* instances indicating exactly why input values do not meet the Spec.

4.3 Spec Errors

class dataspec.**ErrorDetails**(*message: str, pred: Union[Mapping[Hashable, SpecPredicate], Mapping[Union[str, OptionalKey[str]], SpecPredicate], Tuple[SpecPredicate, ...], List[SpecPredicate], FrozenSet[Any], Set[Any], Type[Any], Callable[[Any], bool], Callable[[Any], Iterable[ErrorDetails]], Spec], value: Any, via: List[str] = NOTHING, path: List[Any] = NOTHING*)
`ErrorDetails` instances encode details about values which fail Spec validation.

The `message` of an `ErrorDetails` object gives a human-readable description of why the value failed to validate. The `message` is intended for logs and debugging purposes by application developers. The `message` is *not* intended for non-technical users and dataspec makes no guarantees that builtin error messages could be read and understood by such users.

`ErrorDetails` instances may be emitted for values failing "child" Specs from within mapping, collection, or tuple Specs or they may be emitted from simple predicate failures. The `path` attribute indicates directly which nested element triggered the Spec failure.

`via` indicates the list of all Specs that were evaluated up to and include the current failure for this particular branch of logic. Tags for sibling Specs to the current Spec will not be included in `via`. Because multiple Specs may be evaluated against the same value, it is likely that the number of Tags in `via` will not match the number elements in the `path`.

Parameters

- **message** – a string message intended for developers to indicate why the input value failed to validate
- **pred** – the input Spec predicate that caused the failure
- **value** – the value that failed to validate

- **via** – a list of *dataspec.Tag* s for *dataspec.Spec* s that were evaluated up to and including the one that caused this failure
- **path** – a list of indexes or keys that indicate the path to the current value from the primary value being validated; this is most useful for nested data structures such as `Mapping` types and collections

as_map() → Mapping[str, Union[str, List[str]]]
Return a map of the fields of this instance converted to strings or a list of strings, suitable for being converted into JSON.

The `dataspec.ErrorDetails.pred` attribute will be stringified in one of three ways. If `pred` is a *dataspec.Spec* instance, `pred` will be converted to the *dataspec.Spec.tag* of that instance. If `pred` is a callable (as by `callable()`) , it will be converted to the `__name__` of the callable. Otherwise, `pred` will be passed directly to `str()`.

`message` will remain a string. `value` will be passed to `str()` directly. `via` and `path` will be returned as a list of strings.

Returns a mapping of string keys to strings or lists of strings

with_details(*tag: str*, *loc: Any* = <*object object*>) → dataspec.base.ErrorDetails
Add the given tag to the `via` list and add a key path if one is specified by the caller.

This method mutates the `via` and `path` list attributes directly rather than returning a new `ErrorDetails` instance.

class dataspec.**Invalid**
Objects of type `Invalid` should be emitted from *dataspec.Conformer* s if they are not able to conform a value or if it is not valid.

Builtin `Conformers` emit the constant value *dataspec.INVALID* if they cannot conform their input value. This allows for a fast identity check using Python's `is` operator, though for type checking `Invalid` will required.

class dataspec.**ValidationError**(*errors: Sequence[dataspec.base.ErrorDetails]*)
`ValidationErrors` are thrown by *dataspec.Spec.validate_ex()* and contain a sequence of all *dataspec.ErrorDetails* instances generated by the Spec for the input value.

Parameters **errors** – a sequence of all *dataspec.ErrorDetails* instancess generated by the Spec for the input value

dataspec.**INVALID**
`INVALID` is a singleton instance of *dataspec.Invalid* emitted by builtin conformers which can be used for a quick `is` identity check.

4.4 Utilities

dataspec.**pred_to_validator**(*message: str, complement: bool = False, convert_value: Callable[[Any], Any] = <function _identity>, **fmtkwargs*) → Callable[[Callable[[Any], bool]], Callable[[Any], Iterable[dataspec.base.ErrorDetails]]]
Decorator which converts a simple predicate function to a validator function.

If the wrapped predicate returns a truthy value, the wrapper function will emit a single `dataspec.base.ErrorDetails` object with the `message` format string interpolated with the failing value as `value` (possibly subject to conversion by the optional keyword argument `convert_value`) and any other key/value pairs from `fmtkwargs`.

If `complement` keyword argument is `True`, the return value of the decorated predicate will be converted as by Python's `not` operator and the return value will be used to determine whether or not an error has occurred. This is a convenient way to negate a predicate function without having to modify the function itself.

Parameters

- **message** – a format string which will be the base error message in the resulting `dataspec.base.ErrorDetails` object
- **complement** – if :py:obj:`True`, the boolean complement of the decorated function's return value will indicate failure
- **convert_value** – an optional function which can convert the value before interpolating it into the error message
- **fmtkwargs** – optional key/value pairs which will be interpolated into the error message

Returns a validator function which can be fed into a `dataspec.base.ValidatorSpec`

`dataspec.`**register_str_format**(*tag: str, conformer: Optional[Callable[[T], Union[V, dataspec.base.Invalid]]] = None*) → Callable[[Callable[[Any], Iterable[dataspec.base.ErrorDetails]]], Callable[[Any], Iterable[dataspec.base.ErrorDetails]]]

Register a new String format, which will be checked by the validator function `validate`. A conformer can be supplied for the string format which will be applied if desired, but may otherwise be ignored.

`dataspec.`**tag_maybe**(*maybe_tag: Union[str, T], *args*) → Tuple[Optional[str], Tuple[T, ...]]

Return the Spec tag and the remaining arguments if a tag is given, else return the arguments.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

I

K

M

N

O

P

R

S

T

U

V

W