
datatype Documentation

Release 0.9a3

Adam Wagner

March 28, 2012

CONTENTS

1	What is an anonymous datatype?	3
2	Contents	5
2.1	Defining datatypes	5
2.2	Validation	7
2.3	Coercion	8
2.4	Language Helpers	9
3	Indices and tables	11

Datatype is a collection of tools that help manage anonymous datatypes and currently has support for validation and value coercion.

WHAT IS AN ANONYMOUS DATATYPE?

An anonymous datatype is anything that can be constructed with a small set of built-ins and primitives, specifically: dictionaries, lists, tuples, strings, integers, and floats.

CONTENTS

2.1 Defining datatypes

Datatype definitions are intended to look like the values they define, and are themselves defined as anonymous datatypes.

For example, the following values:

```
{'first_name': 'Bob', 'last_name': 'Smith'}  
{'first_name': 'John', 'last_name': 'Doe'}
```

are both valid instances of this datatype:

```
{'first_name': 'str', 'last_name': 'str'}
```

2.1.1 Primitives

Primitive types are defined as strings bearing the python constructor name.

- String: `"str"`
- Integer: `"int"`
- Float: `"float"`
- Boolean: `"bool"`

Primitive datatypes are, by default, not-nullable. This can be overridden by prefixing the datatype with the "nullable" flag. Example:

```
"nullable str"
```

2.1.2 Lists

Lists-types are homogeneous, and are defined as a list of one datatype.

Example:

```
["int"]
```

Represents a list of ints. While:

```
[{'height': 'float', 'width': 'float'}]
```

Represents a list of dictionaries (or "objects").

2.1.3 Tuples

Tuples can be heterogeneous, but are fixed width and must have at least two items. Again, these are defined as lists of other datatypes:

```
["int", "str"]
```

Represents a tuple with two items. The first is an integer, and the second is a string.

2.1.4 Dictionaries

Dictionaries, or anonymous objects, are defined as dictionaries. The key of each item in the dictionary is the property-name, and the value must be another datatype definition. Example:

```
{'id': 'int', 'name': 'str', 'description': 'str'}
```

By default, all properties listed on the dictionary are required. The following value is invalid, as it lacks the required “description” property:

```
{'id': 5, 'name': 'invalid value'}
```

This behavior can be overridden by prefixing the property-name with the “optional” flag. This datatype *is* valid for the value listed above:

```
{'id': 'int', 'name': 'str', 'optional description': 'str'}
```

Arbitrary properties are supported when the wild-card key “_any_” is defined on the dictionary:

```
{'_any_': 'str'}
```

2.1.5 Literals

The ‘literal’ type can be used to require a specific value:

```
{'_type_': 'literal', 'value': 'my_literal_value'}
```

This value, and only this value, is a valid value for this datatype:

```
'my_literal_value'
```

2.1.6 Choices

To give an option between several datatypes, the choice type can be used. A choice is defined as a special type of dictionary:

```
{'_type_': 'choice', 'choices': ['int', 'str']}
```

This type allows for either an integer or a string.

Choices can be used anywhere other types can be used. This type, for example, represents a heterogeneous list of integers and booleans:

```
[['_type_': 'choice', 'choices': ['int', 'bool']]]
```

The following are all valid values of this type:

```
[5, True, False]
[1, 2, 3]
[False]
```

2.1.7 Named/Recursive Types

Recursion requires the use of two special types: *named* and *reference*:

```
{'_type_': 'named', 'name': 'mytype', 'value': {'foo': 'bar'}}
{'_type_': 'reference', 'name': 'mytype'}
```

The named type does nothing more than provide a name to a datatype. The reference type, then, is used to refer to this named type a later point for things like recursion. Here's a complete example:

```
{
  '_type_': 'named',
  'name': 'person',
  'value': {
    'name': 'str',
    'children': [{'_type_': 'reference', 'name': 'person'}]
  }
}
```

This type represents a person who can have a list of children, each of which, can have their own children, etc. The following is a valid value of this type:

```
{
  'name': 'bob',
  'children': [
    {
      'name': 'frank',
      'children': []
    },
    {
      'name': 'jane',
      'children': [
        {'name': 'alfred', 'children': []}
      ]
    }
  ]
}
```

2.2 Validation

2.2.1 datatype.validation

`datatype.validation.failures` (*datatype*, *value*)
Return list of failures (if any) validating *value* against *datatype*.

Params:

datatype: Datatype to validate value against. See [README.markdown](#) for examples.

value: Value to validate *path*: Used internally for location of failures.

Example:

```
>>> failures('int', 'foo')
['expected int, got str']
```

`datatype.validation.is_valid(datatype, value)`
 Return boolean representing validity of *value* against *datatype*.

2.2.2 datatype.decorators

The decorators module provides means for enforcing proper function return value datatypes.

`datatype.decorators.returns(dfn, strict=True)`
 Make decorators to watch return values of functions to ensure they match the given datatype definition.

Optional Arguments: `strict`: if false, unexpected values on dictionaries will not raise an exception

Example:

```
>>> @returns('int')
... def myfunction():
...     return "bad return value"
>>> myfunction()
Traceback (most recent call last):
BadReturnValueError
```

`datatype.decorators.returns_iter(dfn, strict=True)`
 Validate output of iterator/generator function.

Note: exceptions for bad return datatypes will not be raised until the bad value of the iterator is consumed.

Optional Arguments: `strict`: if false, unexpected values on dictionaries will not raise an exception

Example:

```
>>> @returns_iter('str')
... def myfunction():
...     for x in range(3):
...         yield "number %s" % x
>>> list(myfunction()) # no error
['number 0', 'number 1', 'number 2']
```

2.3 Coercion

2.3.1 datatype.coercion

`datatype.coercion.coerce_value(datatype, value)`
 Attempt to coerce value to requested datatype.

Example:

```
>>> coerce_value("int", "5")
5
```

If coercion is not possible, the incoercible portion is left changed.

Example:

```
>>> coerce_value(['int'], ['1', '2', 'c'])
[1, 2, 'c']
```

2.4 Language Helpers

2.4.1 datatype.language

These small helper functions give a higher level interface for constructing complex datatype definitions, while maintaining portable datatypes.

`datatype.language.choice` (*choices)
Returns a 'choice' dictionary of the given choices.

Example:

```
>>> choice('str', 'int')
{'_type_': 'choice', 'choices': ['str', 'int']}
```

`datatype.language.literal` (value)
Returns a 'literal' dictionary.

Example:

```
>>> literal('foo')
{'_type_': 'literal', 'value': 'foo'}
```

`datatype.language.named` (name, value)
Returns a 'named' dictionary for naming the given 'value' datatype.

Example:

```
>>> named('person', {'first_name': 'str'})
{'_type_': 'named', 'name': 'person', 'value': {'first_name': 'str'}}
```

`datatype.language.reference` (name)
Returns a 'reference' dictionary for the given named type.

Example:

```
>>> reference('person')
{'_type_': 'reference', 'name': 'person'}
```


INDICES AND TABLES

- *genindex*
- *modindex*
- *search*