Python

tonyfast

Apr 07, 2021

CONTENTS

1 `schemata` types and traits **3**

1.1 `schemata` types 3

1.2 `schemata` traits 3

2 comparison with other trait libraries **9**

`schemata` is trait system for Python and IPytohn applications that provide:

- validation with jsonschema conventions
- testing strategies `hypothesis`
- user interfaces
  - ansi with `rich`
  - a port of the react json schema form ui-schema to `ipywidgets`
- observable pattern trait patterns

`schemata`'s type system is composable and expresses the type annotations in the form of json schema. attaching the schema to the types allows different interfaces to construct defaults in aspects like testing, visualization, and verification.

Learn more from the documentation.

CHAPTER

ONE

SCHEMATA TYPES AND TRAITS

schemata is python type/trait system based on that jsonschema. the schemata types build jsonschema that describe test case, validation, observable patterns, and visualization.

1.1 schemata types

schemata is designed as a composable type system for python that allows enhanced type descriptions through common jsonschema conventions. as types, schemata is concerned with constructing json schema representation of type annotations. in other words, it is concerned with enhanced type descriptions beyond the vanilla string, integer, float, lists, and dictionaries python provides.

the enhanced type descriptions provide extra constraints on the types that are defined in the Json Schema Draft 7 specification. the schemata.base.Generic describes that abstract interfaces for the schemata types are specifies how types are build and composed.

jsonschema provides ~50 extra properties to describe the basic types. schemata can extend to other jsonschema, and apply conventions like the react json schema form ui-schema convention that defines how types should be represented.

1.2 schemata traits

the composability of schemata types, and the enhanced schema representation, make it possible to derive different contextual views of python objects.

1.2.1 schemata type validation

1.2.2 schemata user interfaces

1.2.3 schemata testing strategies

1.2.4 schemata applications

schemata types and traits

```
from schemata import *
```

the `String` type

the `String` is a normal python string that can contain extended types descriptions for enhanced validation and representation.

```
    String("abc")
```

```
'abc'
```

```
    String.text()("abc")
```

```
Text(value='abc')
```

regular expressions

one condition is a `String.Pattern` that defines a regular expression that validates the input.

```
    String.pattern("^a")("abc")
```

```
'abc'
```

another situation is a string that is a regular expression.

```
    strings.Regex("^a").match("abc")
```

```
<re.Match object; span=(0, 1), match='a'>
```

the `Enum` type

```
    Enum["a", "b"]("a")
```

```
'a'
```

the `Integer and Float` types

```
    import math
```

```
    Integer(1), Float(math.pi)
```

```
(1, 3.141592653589793)
```

```
    Integer.minimum(0).maximum(100).multipleOf(2).range()(4)
```

```
IntSlider(value=4, step=2)
```

```
    Float.minimum(0).maximum(100).text()(math.pi)
```

```
BoundedFloatText(value=3.141592653589793)
```

the `List` type

```
    List([1, "a", 2, "b"])
```

```
[1, 'a', 2, 'b']
```

```
    List[String](["a", "b"])
```

```
['a', 'b']
```

```
    List.minItems(1).maxItems(3)([1, 2])
```

```
[1, 2]
```

Composite types

AnyOf

```
    Integer | String
```

```
abc.AnyOf
```

OneOf

```
    Integer ^ String
```

```
abc.OneOf
```

AllOf

```
    String & String.Pattern["^a"]
```

```
abc.AllOf
```

Python Types

```
    Py
```

```
schemata.types.Py
```

the `Null` type

```
    assert Null() is Null(None) is Null[None]() is None
```

the `Bool` type

```
    assert Bool() is Bool[False]() is Bool[True](False) is bool() is False
```

```
    assert Bool(True) is Bool[True]() is Bool[False](True) is True
```

`schemata` ipython extension

```
from schemata import *
from IPython import get_ipython
%reload_ext schemata
```

String widgets

```
string: String = "abc"
```

```
'abc'
```

```
string_text: String.text() = "def"
```

```
Text(value='def', description='string_text')
```

```
string_textarea: String.textarea() = "hij"
```

```
Textarea(value='hij', description='string_textarea')
```

```
# NBVAL_IGNORE_OUTPUT
string_html: strings.Html = "klm"
```

```
'klm'
```

```
# string = string_textarea = string_text = string_html ="wxyz"
```

```
string_date: strings.Date = "2020-01-01" # need to fix this
```

```
datetime.datetime(2020, 1, 1, 0, 0)
```

Numeric widgets

```
integer: Integer = 1
```

```
1
```

```
integer_updown: Integer.updown() = 2
```

```
IntText(value=2, description='integer_updown')
```

```
integer_range: Integer.range() = 3
```

```
IntSlider(value=3, description='integer_range')
```

```
integer_bounded: Integer.minimum(0).maximum(10).ui() = 5
```

```
BoundedIntText(value=5, description='integer_bounded', max=10)
```

```
number: Number = 1.5
```

```
1.5
```

```
number_updown: Number.updown() = 2.14
```

```
FloatText(value=2.14, description='number_updown')
```

```
number_range: Number.range() = 3.14
```

```
FloatSlider(value=3.14, description='number_range')
```

```
number_bounded: Number.minimum(0).maximum(10).ui() = 5.6
```

```
BoundedFloatText(value=5.6, description='number_bounded', max=10.0)
```

Enum widgets

```
e = Enum[["a", "b", "c", "d", "e", "f", "g"]]
```

```
enum: e = "a"
```

```
'a'
```

```
enum_dropdown: e.dropdown() = "b"
```

```
Dropdown(description='enum_dropdown', index=1, options=('a', 'b', 'c', 'd', 'e', 'f',
↪'g'), value='b')
```

```
enum_select: e.select() = "c"
```

```
Select(description='enum_select', index=2, options=('a', 'b', 'c', 'd', 'e', 'f', 'g
↪'), value='c')
```

```
enum_slider: e.range() = "d"
```

```
SelectionSlider(description='enum_slider', index=3, options=('a', 'b', 'c', 'd', 'e',
↪'f', 'g'), value='d')
```

```
enum_toggle: e.toggle() = "e"
```

```
ToggleButtons(description='enum_toggle', index=4, options=('a', 'b', 'c', 'd', 'e', 'f
↪', 'g'), value='e')
```

```
enum_radio: e.radio() = "f"
```

```
RadioButtons(description='enum_radio', index=5, options=('a', 'b', 'c', 'd', 'e', 'f',
↪ 'g'), value='f')
```

```
# enum = enum_dropdown = enum_radio = enum_select = enum_slider = enum_toggle = "a"
```

CHAPTER

TWO

COMPARISON WITH OTHER TRAIT LIBRARIES

`schemata` is preceded by a few different trait libraries `enthought.traits`, `traitlets`, and `pydantic`. `traitlets` is a reimplementation of the `enthought.traits` by the `IPython` community; `traitlets` have been the configuration for `jupyter` and `IPython` since. `traitlets` preceeded a lot of critical web technology critical to the `jupyter` interactive computing ecosystem; `traitlets` are only concerned with Python objects and lack features of the modern. `pydantic` provides value as trait system by building off of the `jsonschema` specification to validate types. `schemata` unifies `traitlets` and `pydantic` by providing a description type interface based off of open web standards.

The desire is a trait system for interactive computing that enables more expressive design and testing of interactive computing technology.