



Ads by Google

Send feedback

Why this ad?

Updated On : May-19,2020

python, callbacks, data-types



traitlets - Eventful Classes in Python

Table of Contents

- [Introduction](#)
- [1. Classes with Typed Attributes](#)
 - [Example 1](#)
 - [Example 2](#)
 - [List of Common Data Types in `traitlets`](#)
- [2. Callbacks using `observe\(\)`](#)
 - [Example 1: Observing State Change of Single Attribute using `observe\(\)`](#)
 - [Example 2: Observing State Change of Single Attribute using `@observe`](#)
 - [Example 3: Observing State Change of Multiple Attributes](#)
- [3. Validation using `validate\(\)`](#)
 - [Example 1: Validation of Single Class Attribute](#)
 - [Example 2: Validation of Multiple Class Attributes](#)
 - [Partial Asynchronous Update](#)
 - [Synchronous Update of All Attributes & Validation](#)
- [4. Linking Attributes of Different Classes using `link\(\)` & `dlink\(\)` to Reflect](#)
 - [Example 1 : Bidirectional Link using `link\(\)`](#)
 - [Example 2 Unidirectional Link using `dlink\(\)`](#)
- [References](#)

Introduction

The variables in python do not have any data type by default. The variable in python can hold any type of data (`int`, `float`, `bool`, `list`, `dict`, etc). The attributes of python classes also can hold any type of data unlike other programming languages (`java`, `C++`, etc). As there is not data type enforced in python, we generally assign a default value of `None` to variables in python to indicate that they do not have any default values as well for attributes. Apart from that python classes do not have a way to handle a particular event happens like change in any attribute values.

Python has a library called `traitlets` which can solve all the above-mentioned functionalities:

- It lets python classes have `attribute (trait) type checking` as well as
- It lets python classes define `callbacks` which gets called when attributes are changed
- It lets python classes define `validation methods` which validates the value of attributes

The famous python library called `ipywidgets` that provides widgets (sliders, checkboxes, etc) uses `traitlets` to call callback functions when its widget values changes. Many Python libraries like `ipywebrtc` etc which are based on `ipywidgets` are also based on `traitlets` indirectly.

As a part of this tutorial, we'll be explaining the usage of `traitlets` and creating eventful classes with them.

So without further delay, let's get started with coding.

We'll begin by importing all the necessary libraries.

```
import traitlets
```

Other Python Tutorials

- [asyncio - Synchronization Primitives in Concurrent Programming using Async/Await Syntax](#)
- [logging.config - Simple Guide to Configure Loggers from Dictionary and Config Files in Python](#)
- [mimetypes - Guide to Determine MIME Type of File](#)
- [textwrap - Simple Guide to Wrap and Fill Text in Python](#)
- [email - How to Represent an Email Message in Python?](#)

Other Categories

- [Artificial Intelligence \(2\)](#)
- [Data Science \(62\)](#)
- [Digital Marketing \(7\)](#)
- [Machine Learning \(33\)](#)

Newsletter Subscription

Email Address

Subscribe



Ads by Google

Send feedback

Why this ad?

point to any type of data by default unlike other programming languages so not present in python. As a programmer wants to enforce data types on attributes of a class to prevent the variable from getting misused, they want to use a library like `traitlets` which gets called when a change in attribute values occurs.

`traitlets` provides a list of below-mentioned functionalities:

`ipywidgets` makes extensive use of `traitlets` and other libraries like `bokeh`, `pyleaflet`, `pythreejs`, etc are also based on `traitlets` indirectly.

1. Classes with Typed Attributes

We'll start our tutorials by introducing how to create classes with typed attributes. We'll be creating classes where each attribute of the class has a predefined data type. We'll try setting various values for these attributes to check whether data type validation works.

Example 1

Our first example involves the `Student` class which has 2 attributes.

- `StudentId`
- `StudentName`

We'll first create it as a normal python class and explain how it lets us set any value of attributes without enforcing any datatype. We'll then convert it to traitlets classes that enforce data types.

```
class Student:
    def __init__(self, studentId, studentName):
        self.studentId = studentId
        self.studentName = studentName

student1 = Student(1, "Donald")
```

```
print("Student ID : ", student1.studentId)
print("Student Name : ", student1.studentName)
```

```
Student ID : 1
Student Name : Donald
```

```
student2 = Student(1.5, True)
```

```
print("Student ID : ", student2.studentId)
print("Student Name : ", student2.studentName)
```

```
Student ID : 1.5
Student Name : True
```

We want student id to be integer and student name to be a string but from the above examples, we can easily notice that normal python class does not enforce any data type. It also does not let us set any data type.

We'll now try to convert the above class to traitlets class. We'll also enforce integer data type for student id and string for student names. In order to convert normal python class to traitlets class, it needs to extend the `HasTraits` traitlets class. We have also defined data types for student id and student names by declaring them using `traitlets.Int()` and `traitlets.Unicode()` classes which will enforce respective data types.

NOTE

Please make a note that all classes which wants to utilize functionalities provided by **traitlets** must extend **HasTraits** traitlets class to convert normal python class to eventful class.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()

    def __init__(self, studentId, studentName):
        self.studentId = studentId
        self.studentName = studentName

student3 = Student(2, "Putin")
```

```
print("Student ID : ", student3.studentId)
print("Student Name : ", student3.studentName)
```

```
Student ID : 2
Student Name : Putin
```

We can clearly see from the above examples that when we try to create instances with proper data type then there is no failures.

We'll now try to modify attributes with different data types to check whether it lets us set data with different data types than declared ones. We'll be catching errors using the try-except block and print it to understand the type of error.

```
try:
    student3.studentId = 1.5
except Exception as e:
    print(e)
```

```
The 'studentId' trait of a Student instance must be an int, but a value of 1.5 <class 'float'> was specified.
```

```
try:
    student3.studentName = True
except Exception as e:
    print(e)
```

```
The 'studentName' trait of a Student instance must be a unicode string, but a value of True <class 'bool'> was specified.
```

We can clearly see from the above errors that assignments in both cases failed when we tried to assign float value to integer attribute and boolean value to string variable.

Below we are trying another example where we are trying to create an instance of a class with attribute values with different data type which fails as well.

```
try:
    student4 = Student("Fake", False)
except Exception as e:
    print(e)
```

The 'studentId' trait of a Student instance must be an int, but a value of 'Fake' <class 'str'> was specified.

Example 2

Below we are explaining another example where we are creating class instance without setting any attribute values. We'll then try to print out default values for each attribute to see which default values are set.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()

    def __init__(self):
        pass

student5 = Student()

print("Student ID : ", student5.studentId)
print("Student Name : ", student5.studentName)
```

Student ID : 0
Student Name :

We can see that in case of integer attribute default value of 0 has been set and the empty string has been set for string attribute.

There are situations when we want default values to be different than the one set for that data types by default. The `traitlets` provides annotation named `@default` which can be given to a method passing it attribute name of the class and it'll set the value returned by a method as a default value for that attribute of a class.

Below we are explaining the usage of `@default` annotation where we are setting a different default value for string attribute than default empty string.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()

    def __init__(self, studentId):
        self.studentId = studentId

    @traitlets.default("studentName")
    def _get_default_name(self):
        return "Pappu"

student6 = Student(2)

print("Student ID : ", student6.studentId)
print("Student Name : ", student6.studentName)
```

Student ID : 2
Student Name : Pappu

Below we are again trying to set a list as the value of student name and it fails like previous examples with a data type error.

```
try:
    student6.studentName = [1,2,3]
except Exception as e:
    print(e)
```

The 'studentName' trait of a Student instance must be a unicode string, but a value of [1, 2, 3] <class 'list'> was specified.

List of Common Data Types in `traitlets`

Below is a list of common data type classes provided by `traitlets` . We need to initialize attributes of classes with these class instances and respective data type will be forced.

- Integer/Int
- Long
- Float
- Complex
- Unicode
- Bytes
- List
- Set
- Tuple
- Dict
- Bool
- Enum

The `traitlets` provides many other data types than these. These are commonly used data types.

2. Callbacks using `observe()`

We'll now go ahead with an explanation of capturing events using traitlets. All traitlets class instances provide a method named `observe()` which accepts a function to call when particular events happen like a change in a particular attribute value. We'll be explaining below the implementation of event handling using traitlets in python below with examples.

Example 1: Observing State Change of Single Attribute using `observe()`

Below we are explaining the first example of events using traitlets. We want to monitor change in student address attribute of a class.

We have first declared a method named `monitor_address_change` which accepts one argument which will hold information about the change in an attribute. The change details passed to this method will be a dictionary which will hold old and new both values for student address attribute each time change to it happens.

Then we need to call `observe()` method on student instance and pass it method to call when change happens and `names` parameter holding attribute (student address) to monitor.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()
    studentAddress = traitlets.Dict()

    def __init__(self, studentId, studentName):
        self.studentId = studentId
        self.studentName = studentName

def monitor_address_change(addr_change_details):
    print(addr_change_details)
    print("Old Address : ", addr_change_details["old"])
    print("New Address : ", addr_change_details["new"])

student7 = Student(7, "Angela")

student7.observe(monitor_address_change, names = ["studentAddress"])

print("Student ID : ", student7.studentId)
print("Student Name : ", student7.studentName)
print("Student Address : ", student7.studentAddress)

Student ID : 7
Student Name : Angela
Student Address : {}

student7.studentAddress = {"Country" : "India", "State": "Gujarat", "City": "Surat"}

{'name': 'studentAddress', 'old': {}, 'new': {'Country': 'India', 'State': 'Gujarat', 'City': 'Surat'}, 'owner':
<__main__.Student object at 0x7f21701ae8d0>, 'type': 'change'}
Old Address : {}
New Address : {'Country': 'India', 'State': 'Gujarat', 'City': 'Surat'}

print("Student ID : ", student7.studentId)
print("Student Name : ", student7.studentName)
print("Student Address : ", student7.studentAddress)

Student ID : 7
Student Name : Angela
Student Address : {'Country': 'India', 'State': 'Gujarat', 'City': 'Surat'}
```

From the above example, we can clearly see that each time change to student address attribute happens, the `monitor_address_change` method gets called with state change details.

Example 2: Observing State Change of Single Attribute using `observe` Annotation

Below we are explaining the same example as above one but with a different approach than the above one. The `traitlets` also provides annotation named `@observe()` which can be assigned to a method of the class passing it name of class attribute changes to which will result in a call of that method.

The benefit of using annotation is that it'll be applied to all instances of the class by default and we won't need to call observe on all instances of the class.

We are declaring a method named `monitor_address_change` as the class method this time rather than outside of class. We have annotated it with `@observe()` annotation passing it student address attribute name. The method performs exactly the same operation as the last time of printing state change details.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()
    studentAddress = traitlets.Dict()

    def __init__(self, studentId, studentName):
        self.studentId = studentId
        self.studentName = studentName

    @traitlets.observe("studentAddress")
    def monitor_address_change(self, addr_change_details):
        print("Old Address : ", addr_change_details["old"])
        print("New Address : ", addr_change_details["new"])

student8 = Student(7, "Boris")

print("Student ID : ", student8.studentId)
print("Student Name : ", student8.studentName)
print("Student Address : ", student8.studentAddress)

student8.studentAddress = {"Country" : "India", "State": "Gujarat", "City": "Surat"}

Old Address : {}
New Address : {'Country': 'India', 'State': 'Gujarat', 'City': 'Surat'}

student8.studentName = "Johnson"

print("Student ID : ", student8.studentId)
print("Student Name : ", student8.studentName)
print("Student Address : ", student8.studentAddress)

Student ID : 7
Student Name : Johnson
Student Address : {'Country': 'India', 'State': 'Gujarat', 'City': 'Surat'}
```

We can see from the above value setting examples that every time we change student address, the method `monitor_address_change` gets called and it prints old and new addresses for our information.

Example 3: Observing State Change of Multiple Attributes

Below we are explaining an example where we are monitoring more than one attribute of the class using `@observe()` annotation. We can pass more than one attribute to annotation and it'll monitor changes to all of them.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()
    studentAddress = traitlets.Dict()

    def __init__(self, studentId):
        self.studentId = studentId

    @traitlets.observe("studentAddress", "studentName")
    def monitor_address_change(self, change_details):
        print("Old Data : ", change_details["old"])
        print("New Data : ", change_details["new"])

student9 = Student(7)

print("Student ID : ", student9.studentId)
print("Student Name : ", student9.studentName)
print("Student Address : ", student9.studentAddress)

Student ID : 7
Student Name : 
Student Address : {}

student9.studentName = "Imran"

Old Data : 
New Data : Imran
```

```
student9.studentAddress = {"Country":"Afganistan", "State":"Unknown", "City":"Kabul"}

Old Data : {}
New Data : {'Country': 'Afganistan', 'State': 'Unknown', 'City': 'Kabul'}

student9.studentName = "Imran Khan"
student9.studentAddress = {"Country":"Pakistan", "State":"Unknown", "City":"Karachi"}

Old Data : Imran
New Data : Imran Khan
Old Data : {'Country': 'Afganistan', 'State': 'Unknown', 'City': 'Kabul'}
New Data : {'Country': 'Pakistan', 'State': 'Unknown', 'City': 'Karachi'}

print("Student ID : ", student9.studentId)
print("Student Name : ", student9.studentName)
print("Student Address : ", student9.studentAddress)

Student ID : 7
Student Name : Imran Khan
Student Address : {'Country': 'Pakistan', 'State': 'Unknown', 'City': 'Karachi'}
```

From the above value settings examples, we can see that each time there is a change in student address or name, method `monitor_address_change` gets called to print old and new values for attributes.

3. Validation using `validate()` Annotation

The traitlets library also provides validation functionalities as a part of its framework. It lets us validate attribute values when they change and change value only if it meets certain validation. It provides `validate()` annotation for performing validation of attributes. Below we are explaining the usage of `validate()` by validating single and multiple attributes of a class. The validation method is used when you want further validation than basic data type validation.

Example 1: Validation of Single Class Attribute

To explain the usage of `@validate()` annotation, we'll be restricting our student id between 1 and 50 considering class has only 50 students.

We have designed a method named `_validate_id` which takes on an argument which is generally a dictionary holding the value of an attribute for which it is doing validation. We have an annotated method with `@validate()` passing it student id attribute name which will enforce method to execute when student id is set. We have put logic in the method that it'll raise an error if student id assigned to student instance is not between 1-50.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()
    studentAddress = traitlets.Dict()

    def __init__(self, studentId, studentName):
        self.studentId = studentId
        self.studentName = studentName

    @traitlets.validate("studentId")
    def _validate_id(self, student_id_details):
        if student_id_details["value"] <1 or student_id_details["value"] >50:
            raise traitlets.TraitError("Student ID should be between 1-50. Invalid Value : %d"%
(student_id_details["value"]))
        return student_id_details["value"]

student10 = Student(1, "Elon")

student10.studentId = 10

try:
    student10.studentId = -1
except Exception as e:
    print(e)

Student ID should be between 1-50. Invalid Value : -1
```

We can clearly see from the above value setting example that when we try to set a negative value for student id then `@validate()` annotated method raised error.

Example 2: Validation of Multiple Class Attributes

Below we have designed another example explaining the usage of `@validate()` annotation which is getting used this time to validate more than one attribute. We are validating two attributes this time. We want to enforce student id between 1-50 and student address should compulsory has country information.


```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()
    studentAddress = traitlets.Dict()

    def __init__(self, studentId, studentName):
        self.studentId = studentId
        self.studentName = studentName

    @traitlets.validate("studentId")
    def _validate_id(self, student_id_details):
        if student_id_details["value"] <1 or student_id_details["value"] >50:
            raise traitlets.TraitError("Student ID should be between 1-50. Invalid Value : %d"%(student_id_details["value"]))
        return student_id_details["value"]

    @traitlets.validate("studentAddress")
    def _validate_address(self, student_addr_details):
        if "Country" not in student_addr_details["value"]:
            raise traitlets.TraitError("Country is compulsory in Address. Invalid Value : %s"%(str(student_addr_details["value"])))
        return student_addr_details["value"]

student11 = Student(10, "Bill")
```

```
student11.studentAddress = {"Country":"US", "State":"NY", "City":"NY"}
```

```
try:
    student11.studentAddress = {"State":"California", "City":"San Jose"}
except Exception as e:
    print(e)
```

Country is compulsory in Address. Invalid Value : {'State': 'California', 'City': 'San Jose'}

```
student11.studentAddress
```

{'Country': 'US', 'State': 'NY', 'City': 'NY'}

We can clearly notice from the above examples that when we try to set value for an address without country information, assignment fails.

Below we are trying to set both student id and student address at the same time. We'll first try to set student id and then student address.

```
try:
    student11.studentId = -1
    student11.studentAddress = {"State":"Toronto", "City":"Toronto"}
except Exception as e:
    print(e)
```

Student ID should be between 1-50. Invalid Value : -1

```
student11.studentId, student11.studentAddress
```

(10, {'Country': 'US', 'State': 'NY', 'City': 'NY'})

We can notice that the above assignments fail as well.

Partial Asynchronous Update

The partial asynchronous updates happen when we try to change multiple attributes at the same time where some of them have valid values and some of them do not.

From the below assignment example, we can see that the student address has valid data getting assigned but student id is wrong. In this case, as address assignment is happening first, it'll be successful and student id will fail.

The partial asynchronous updates might not be ideal in all cases as we might want that update happens to all attributes getting changed or none gets updated.

```
try:
    student11.studentAddress = {"Country":"US", "State":"California", "City":"San Jose"}
    student11.studentId = -1
except Exception as e:
    print(e)
```

Student ID should be between 1-50. Invalid Value : -1

```
student11.studentId, student11.studentAddress
```

(10, {'Country': 'US', 'State': 'California', 'City': 'San Jose'})

We can see above that student address details got changed even though the error was raised when changing student id.

Synchronous Update of All Attributes & Validation

The synchronous update is related to the previous example where we want to get to update all attributes getting updated or none.

The `traitlets` provides method named `hold_trait_notifications()` for synchronous update of class attributes. We need to update attributes of class in the context of the `hold_trait_notifications()` method and it'll update all attributes if updates to all of them succeed without error else it'll rollback all attributes. It's almost like database transactions.

```
try:
    with student11.hold_trait_notifications():
        student11.studentAddress = {"Country": "US", "State": "NY", "City": "NY"}
        student11.studentId = -1
except Exception as e:
    print(e)
```

Student ID should be between 1-50. Invalid Value : -1

student11.studentId, student11.studentAddress

(10, {'Country': 'US', 'State': 'California', 'City': 'San Jose'})

We can notice from the above example that even though the student address was set first with a valid value and student address later with an invalid value, both student address and id did not get updated. It was the reverse case with the `try-except` block in the previous section.

4. Linking Attributes of Different Classes using `link()` & `dlink()` to Reflect Changes Across Classes

The fourth functionality that we are going to discuss traitlets is quite unique functionality which is heavily getting used by `ipywidgets` as well for linking widgets. The traitlets provide methods that let us link the attribute of one class with attribute of another class to keep values of an attribute of both class `synchronous`. It provides two kinds of links:

- **Bidirectional Link:** If the value of one attribute changes in the link then the value of another attribute also changes to the value of that first attribute and vice-versa is true as well.
- **Unidirectional Link:** It's a directional link meaning that if the value of left attribute changes then the value of the right attribute changes but the reverse is not true.

We'll be explaining the usage of both link types to link various attributes of different classes below with examples to further clear understanding of the concept.

Example 1 : Bidirectional Link using `link()`

Below we are explaining the usage of a bidirectional link example where we are designing two classes (Student & Sportsman). We want that id and name details between students and sportsmen should be in sync for the same student. We are also monitoring changes in id and name details for both using `@observe()` annotation.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()
    studentAddress = traitlets.Dict()

    def __init__(self, studentId):
        self.studentId = studentId

    @traitlets.observe("studentId", "studentName")
    def monitor_address_change(self, change_details):
        print("Old Data : ", change_details["old"])
        print("New Data : ", change_details["new"])

class Sportsman(traitlets.HasTraits):
    sportsmanId = traitlets.Int()
    sportsmanName = traitlets.Unicode()

    def __init__(self, sportsmanId):
        self.sportsmanId = sportsmanId

    @traitlets.observe("sportsmanId", "sportsmanName")
    def monitor_address_change(self, change_details):
        print("Old Data : ", change_details["old"])
        print("New Data : ", change_details["new"])

student12 = Student(10)
sportsman12 = Sportsman(10)
```

Old Data : 0
New Data : 10
Old Data : 0
New Data : 10

We can now create a bidirectional link between student id and sportsman id. The `traitlets` provides a method named `link()` for this purpose which accepts a list of tuples where the first element is class instance and 2nd element of the tuple is a class attribute to link to other class's attributes.

We are also creating a bidirectional link between student and sportsman names.

These bidirectional links will synchronize changes in id and name details between instances of both classes.

```
bidirectional_link1 = traitlets.link((student12, "studentId"),(sportsman12, "sportsmanId"))
bidirectional_link2 = traitlets.link((student12, "studentName"),(sportsman12, "sportsmanName"))
```

student12.studentName = "Lee"

Old Data :
New Data : Lee
Old Data :
New Data : Lee


```
sportsman12.sportsmanName
```

```
'Lee'
```

We can notice from the below example that when we change the value of sportsman id, it also changes the id of a student due to bidirectional link.

```
sportsman12.sportsmanId = 15
```

```
Old Data : 10
New Data : 15
Old Data : 10
New Data : 15
```

```
student12.studentId
```

```
15
```

We can also unlink attributes of class if we don't want to sync values anymore. We can call `unlink()` method on link instance created when creating links and it'll unlink instance attributes. The changes to one attribute then won't reflect in another.

```
bidirectional_link1.unlink()
bidirectional_link2.unlink()
```

```
student12.studentName = "Bruce"
```

```
Old Data : Lee
New Data : Bruce
```

```
sportsman12.sportsmanName
```

```
'Lee'
```

```
sportsman12.sportsmanId = 12
```

```
Old Data : 15
New Data : 12
```

```
student12.studentId
```

```
15
```

We can notice that after attributes are unlinked, changes are not getting reflected across.

Example 2 Unidirectional Link using `dlink()`

The below example is almost the same as the last example but instead of a bidirectional link between class attributes unidirectional link is created with direction from left to right. The `traitlets` provides `dlink()` method for this purpose.

```
class Student(traitlets.HasTraits):
    studentId = traitlets.Int()
    studentName = traitlets.Unicode()
    studentAddress = traitlets.Dict()

    def __init__(self, studentId):
        self.studentId = studentId

    @traitlets.observe("studentId", "studentName")
    def monitor_address_change(self, change_details):
        print("Old Data : ", change_details["old"])
        print("New Data : ", change_details["new"])

class Sportsman(traitlets.HasTraits):
    sportsmanId = traitlets.Int()
    sportsmanName = traitlets.Unicode()

    def __init__(self, sportsmanId):
        self.sportsmanId = sportsmanId

    @traitlets.observe("sportsmanId", "sportsmanName")
    def monitor_address_change(self, change_details):
        print("Old Data : ", change_details["old"])
        print("New Data : ", change_details["new"])

student13 = Student(10)
sportsman13 = Sportsman(10)
```

```
Old Data : 0
New Data : 10
Old Data : 0
New Data : 10
```

```
unidirectional_link1 = traitlets.dlink((student13, "studentId"),(sportsman13, "sportsmanId"))
unidirectional_link2 = traitlets.dlink((student13, "studentName"),(sportsman13, "sportsmanName"));
```

```
student13.studentName = "Lee"
```

```
Old Data :
New Data : Lee
Old Data :
New Data : Lee
```

```
sportsman13.sportsmanName

'Lee'

We can notice from below changes that when we change sportsman id, it does not change student id.

sportsman13.sportsmanId = 15

Old Data : 10
New Data : 15

student13.studentId

10

unidirectional_link1.unlink()
unidirectional_link2.unlink()

student13.studentName = "Bruce"

Old Data : Lee
New Data : Bruce

sportsman13.sportsmanName

'Lee'
```

This ends our small tutorial on `traitlets` library for creating classes with an attribute data type, attribute default values, callbacks, validation, and attribute linking between attributes of different class instances. Please feel free to let us know your views in the comments section.

References

- [An In-depth Guide to Widgets in Jupyter Notebook using ipywidgets](#)
- [bqplot - Interactive Plotting in Jupyter Notebook](#)

GoJS is a JavaScript and TypeScript library for developing interactive

Ad JavaScript graphing library for creating diagrams.

GoJS by Northwoods

Open

 Sunny Solanki

Published On : May-19,2020

Time Investment : ~30 mins

ALSO ON CODERZCOLUMN

Candlestick Chart in Python (mplfinance, ...

a year ago · 4 comments

Candlestick Chart in Python (mplfinance, plotly, bokeh)

How to Plot Radar Charts in Python ...

10 months ago · 2 comments

How to Plot Radar Charts in Python [plotly]?

Hottest A Applicati

a year ago ·

Hottest And Of 2020

Sponsored

[2020 war das Jahr von Bitcoin. Was können wir für 2021 erwarten?](#)
eToro

[Klaas Heufer-Umlauf: Darum flog er aus dem Doppelpass](#)
Sport1

[Mit 52 Jahren ist Andreas Türk ein Schatten seines früheren Selbst](#)
Cash Roadster

[Das Foto, das die Karriere einer Stabhochspringerin zerstört](#)
Affluent Times


[Mit 56 Jahren ist Linda de Mol ein Schatten ihres früheren Selbst](#)
Dad's News

[Mit 61 Jahren enthüllte Joachim Löw seinen wahren Partner](#)
Mighty Scoops

[Mit 51 Jahren enthüllt Marietta Slomka ihren wahren Partner](#)
Panda-Texte

What do you think?


1 Response

 Upvote


 Very Useful

CoderzColumn Comment Policy


Be respectful and always strive to be supportive. @mention a moderator if you have any questions.

0 Comments CoderzColumn  Disqus' Privacy Policy

 Recommend  Tweet  Share



LOG IN WITH

OR SIGN UP WITH DISQUS 

Be the first to comment.