free_properties Documentation

Release 2019

Yoel Cortes-Pena

Sep 25, 2022

Contents

CHAPTER 1

PropertyFactory

`free_properties.PropertyFactory`(*fget=None*, *fset=None*, *clsname=None*, *doc=None*, *units=None*, *slots=None*)

Create an FreeProperty subclass with getter and setter functions.

fget [function, optional] Should return value of instances. If not given, a decorator expecting fget will be returned.

fset [function, optional] Should set the value of instances.

clsname [str, optional] Name of the class. Defaults to the function name of fget.

doc [str, optional] Docstring of class. Defaults to the docstring of fget.

units [str, optional] Units of measure.

slots [tuple[str], optional] Slots for class.

The PropertyFactory is a FreeProperty class creator that functions similar to Python 'property' objects. Use the PropertyFactory to create a Weight class which calculates weight based on density and volume:

```
>>> from free_properties import PropertyFactory
>>> def getter(self):
...     '''Weight (kg) based on volume (m^3).'''
...     data = self.data
...     rho = data['rho'] # Density (kg/m^3)
...     vol = data['vol'] # Volume (m^3)
...     return rho * vol
>>>
>>> def setter(self, weight):
...     data = self.data
...     rho = data['rho'] # Density (kg/m^3)
...     data['vol'] = weight / rho
>>>
>>> # Initialize with a value getter, setter, and the class name.
>>> Weight = PropertyFactory(fget=getter, fset=setter, clsname='Weight', units='kg
↪')
```

It is more convinient to use the PropertyFactory as a decorator:

```
>>> @PropertyFactory(units='kg')
>>> def Weight(self):
...     '''Weight (kg) based on volume (m^3).'''
...     data = self.data
...     rho = data['rho'] # Density (kg/m^3)
...     vol = data['vol'] # Volume (m^3)
...     return rho * vol
>>>
>>> @Weight.setter
>>> def Weight(self, weight):
...     data = self.data
...     rho = data['rho'] # Density (kg/m^3)
...     data['vol'] = weight / rho
```

Create dictionaries of data and initialize new Weight objects:

```
>>> water_data = {'rho': 1000, 'vol': 3}
>>> ethanol_data = {'rho': 789, 'vol': 3}
>>> weight_water = Weight('Water', water_data)
>>> weight_ethanol = Weight('Ethanol', ethanol_data)
>>> weight_water
<Water: 3000 kg>
>>> weight_ethanol
<Ethanol: 2367 kg>
```

Note: The units are taken from the the function docstring. The first word in parentesis denotes the units.

These properties behave just like their dynamic value:

```
>>> weight_water + 30
3030
>>> weight_water + weight_ethanol
5367
```

Get and set the value through the 'value' attribute:

```
>>> weight_water.value
3000
>>> weight_water.value = 4000
>>> weight_water.value
4000.0
>>> water_data # Note that the volume changed too
{'rho': 1000, 'vol': 4.0}
```

In place magic methods will also change the property value:

```
>>> weight_water -= 1000
>>> weight_water
<Water: 3000 kg>
>>> water_data  # The change also affects the original data
{'rho': 1000, 'vol': 3.0}
```

CHAPTER 2

FreeProperty

class free_properties.**FreeProperty**(**args*, ***kwargs*)

Abstract Property class. Child classes must include a 'value' property.

CHAPTER 3

property_array

class free_properties.**property_array**

Create an array that allows for array-like manipulation of FreeProperty objects. All entries in a property_array must be instances of FreeProperty. Setting items of a property_array sets values of objects instead.

properties : array_like[FreeProperty]

Use the PropertyFactory to create a Weight property class which calculates weight based on density and volume:

```
>>> from free_property import PropertyFactory, property_array
>>>
>>> @PropertyFactory
>>> def Weight(self):
...     '''Weight (kg) based on volume (m^3).'''
...     data = self.data
...     rho = data['rho'] # Density (kg/m^3)
...     vol = data['vol'] # Volume (m^3)
...     return rho * vol
>>>
>>> @Weight.setter
>>> def Weight(self, weight):
...     data = self.data
...     rho = data['rho'] # Density (kg/m^3)
...     data['vol'] = weight / rho
```

Create dictionaries of data and initialize new properties:

```
>>> water_data = {'rho': 1000, 'vol': 3}
>>> ethanol_data = {'rho': 789, 'vol': 3}
>>> weight_water = Weight('Water', water_data)
>>> weight_ethanol = Weight('Ethanol', ethanol_data)
>>> weight_water
<Weight(Water): 3000 kg>
>>> weight_ethanol
<Weight(Ethanol): 2367 kg>
```

Create a property_array from data:

```
>>> prop_arr = property_array([weight_water, weight_ethanol])
>>> prop_arr
property_array([3000.0, 2367.0])
```

Changing the values of a property_array changes the value of its properties:

```
>>> # Addition in place
>>> prop_arr += 3000
>>> prop_arr
property_array([6000.0, 5367.0])
>>> # Note how the data also changes
>>> water_data
{'rho': 1000, 'vol': 6.0}
>>> ethanol_data
{'rho': 789, 'vol': 6.802281368821292}
>>> # Setting an item changes the property value
>>> prop_arr[1] = 2367
>>> ethanol_data
{'rho': 789, 'vol': 3}
```

New arrays have no connection to the property_array:

```
>>> prop_arr - 1000 #  Returns a new array
array([5000.0, 1367.0])
>>> water_data #  Data remains unchanged
{'rho': 1000, 'vol': 6.0}
```

all (*axis=None*, *out=None*, *keepdims=False*, ***, *where=True*)

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any (*axis=None*, *out=None*, *keepdims=False*, ***, *where=True*)

Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax (*axis=None*, *out=None*)

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin (*axis=None*, *out=None*)

Return indices of the minimum values along the given axis.

Refer to *numpy.argmin* for detailed documentation.

numpy.argmin : equivalent function

argpartition (*kth*, *axis=-1*, *kind='introselect'*, *order=None*)

Returns the indices that would partition this array.

Refer to *numpy.argpartition* for full documentation.

New in version 1.8.0.

numpy.argpartition : equivalent function

argsort (*axis=-1*, *kind=None*, *order=None*)
Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

numpy.argsort : equivalent function

choose (*choices*, *out=None*, *mode='raise'*)
Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

numpy.choose : equivalent function

clip (*min=None*, *max=None*, *out=None*, ***kwargs*)
Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to *numpy.clip* for full documentation.

numpy.clip : equivalent function

conj ()
Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate ()
Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy (*order='C'*)
Return a copy of the array.

order [{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their order= arguments, and this function always passes sub-classes through.)

numpy.copy : Similar function with different default behavior numpy.copyto

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

cumprod (*axis=None*, *dtype=None*, *out=None*)
Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

numpy.cumprod : equivalent function

cumsum (*axis=None*, *dtype=None*, *out=None*)
Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

numpy.cumsum : equivalent function

dot (*b*, *out=None*)
Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

max (*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)
Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean (*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *, *where=True*)
Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min (*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

numpy.amin : equivalent function

nonzero ()
Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

numpy.nonzero : equivalent function

prod(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*)
Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp(*axis=None*, *out=None*, *keepdims=False*)
Peak to peak (maximum - minimum) value along a given axis.

Refer to *numpy.ptp* for full documentation.

numpy.ptp : equivalent function

put(*indices*, *values*, *mode='raise'*)
Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

numpy.put : equivalent function

round(*decimals=0*, *out=None*)
Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

numpy.around : equivalent function

std(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, *, *where=True*)
Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

sum(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=0*, *where=True*)
Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

numpy.sum : equivalent function

trace(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)
Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

numpy.trace : equivalent function

var(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, *, *where=True*)
Returns the variance of the array elements, along given axis.

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index