Python Descriptors

Summary: in this tutorial, you'll learn about Python descriptors, how descriptors work, and how to apply them more effectively.

Introduction to the Python descriptors

Suppose you have a class (https://www.pythontutorial.net/python-oop/python-class/) `Person` with two instance attributes (https://www.pythontutorial.net/python-oop/python-instance-variables/) `first_name` and `last_name` :

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
```

And you want the `first_name` and `last_name` attributes to be non-empty strings. These plain attributes cannot guarantee this.

To enforce the data validity, you can use property (https://www.pythontutorial.net/python-oop/python-properties/) with a getter and setter methods, like this:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise ValueError('The first name must a string')

        if len(value) == 0:
            raise ValueError('The first name cannot be empty')

        self._first_name = value

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if not isinstance(value, str):
            raise ValueError('The last name must a string')

        if len(value) == 0:
            raise ValueError('The last name cannot be empty')

        self._last_name = value
```

In this `Person` class, the getter returns the attribute value while the setter validates it before assigning it to the attribute.

This code works perfectly fine. However, it is redundant because the validation logic validates the first and last names is the same.

Also, if the class has more attributes that require a non-empty string, you need to duplicate this logic in other properties. In other words, this validation logic is not reusable.

To avoid duplicating the logic, you may have a method that validates data and reuse this method in other properties. This approach will enable reusability. However, Python has a better way to solve this by using descriptors.

First, define a descriptor class that implements three methods `__set_name__` , `__get__` , and `__set__` :

```
class RequiredString:
    def __set_name__(self, owner, name):
        self.property_name = name

    def __get__(self, instance, owner):
        if instance is None:
            return self

        return instance.__dict__[self.property_name] or None

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise ValueError(f'The {self.property_name} must be a string')

        if len(value) == 0:
            raise ValueError(f'The {self.property_name} cannot be empty')

        instance.__dict__[self.property_name] = value
```

Second, use the `RequiredString` class in the `Person` class:

```
class Person:
    first_name = RequiredString()
    last_name = RequiredString()
```

If you assign an empty string or a non-string value to the `first_name` or `last_name` attribute of the `Person` class, you'll get an error.

For example, the following attempts to assign an empty string to the `first_name` attribute:

```
try:
    person = Person()
    person.first_name = ''
except ValueError as e:
    print(e)
```

Error:

```
The first_name must be a string
```

Also, you can use the `RequiredString` class in any class with attributes that require a non-empty string value.

Besides the `RequiredString` , you can define other descriptors to enforce other data types like age, email, and phone. And this is just a simple application of the descriptors.

Let's understand how descriptors work.

Descriptor protocol

In Python, the descriptor protocol consists of three methods:

- `__get__` gets an attribute value
- `__set__` sets an attribute value

- `__delete__` deletes an attribute

Optionally, a descriptor can have the `__set_name__` method that sets an attribute on an instance of a class to a new value.

What is a descriptor

A descriptor is an object of a class that implements one of the methods specified in the descriptor protocol.

Descriptors have two types: data descriptor and non-data descriptor.

1. A data descriptor is an object of a class that implements the `__set__` and/or `__delete__` method.
2. A non-data descriptor is an object that implements the `__get__` method only.

The descriptor type specifies the property lookup resolution that we'll cover in the next tutorial (https://www.pythontutorial.net/python-oop/python-data-descriptors/) .

How descriptors work

The following modifies the `RequiredString` class to include the `print` statements that print out the arguments.

```
class RequiredString:
    def __set_name__(self, owner, name):
        print(f'__set_name__ was called with owner={owner} and name={name}')
        self.property_name = name

    def __get__(self, instance, owner):
        print(f'__get__ was called with instance={instance} and owner={owner}')
        if instance is None:
            return self

        return instance.__dict__[self.property_name] or None
    def __set__(self, instance, value):
        print(f'__set__ was called with instance={instance} and value={value}')

        if not isinstance(value, str):
            raise ValueError(f'The {self.property_name} must a string')

        if len(value) == 0:
            raise ValueError(f'The {self.property_name} cannot be empty')

        instance.__dict__[self.property_name] = value


class Person:
    first_name = RequiredString()
    last_name = RequiredString()
```

The __set_name__ method

When you compile the code, you'll see that Python creates the descriptor objects for `first_name` and `last_name` and automatically call the `__set_name__` method of these objects. Here's the output:

```
__set_name__ was called with owner=<class '__main__.Person'> and name=first_name
__set_name__ was called with owner=<class '__main__.Person'> and name=last_name
```

In this example, the owner argument of `__set_name__` is set to the `Person` class in the `__main__` module, and the `name` argument is set to the `first_name` and `last_name` attribute accordingly.

It means that Python automatically calls the `__set_name__` when the owning class `Person` is created. The following statements are equivalent:

```
first_name = RequiredString()
```

and

```
first_name.__set_name__(Person, 'first_name')
```

Inside, the `__set_name__` method, we assign the `name` argument to the `property_name` instance attribute of the `descriptor` object so that we can access it later in the `__get__` and `__set__` method:

```
self.property_name = name
```

The `first_name` and `last_name` are the class variables (https://www.pythontutorial.net/python-oop/python-class-variables/) of the `Person` class. If you look at the `Person.__dict__` class attribute, you'll see two descriptor objects `first_name` and `last_name` :

```
from pprint import pprint

pprint(Person.__dict__)
```

Output:

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Person' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              'first_name': <__main__.RequiredString object at 0x0000019D6AB947F0>,
              'last_name': <__main__.RequiredString object at 0x0000019D6ACFBE80>})
```

The __set__ method

Here's the `__set__` method of the `RequiredString` class:

```
def __set__(self, instance, value):
    print(f'__set__ was called with instance={instance} and value={value}')

    if not isinstance(value, str):
        raise ValueError(f'The {self.property_name} must be a string')

    if len(value) == 0:
        raise ValueError(f'The {self.property_name} cannot be empty')

    instance.__dict__[self.property_name] = value
```

When you assign the new value to a descriptor, Python calls `__set__` method to set the attribute on an instance of the owner class to the new value. For example:

```
person = Person()
person.first_name = 'John'
```

Output:

```
__set__ was called with instance=<__main__.Person object at 0x000001F85F7167F0> a
```

In this example, the `instance` argument is `person` object and the value is the string `'John'`. Inside the `__set__` method, we raise a `ValueError` if the new value is not a string or if it is an empty string.

Otherwise, we assign the value to the instance attribute `first_name` of the `person` object:

```
instance.__dict__[self.property_name] = value
```

Note that Python uses `instance.__dict__` dictionary to store instance attributes of the `instance` object.

Once you set the `first_name` and `last_name` of an instance of the `Person` object, you’ll see the instance attributes with the same names in the instance’s `__dict__` . For example:

```
person = Person()
print(person.__dict__)  # {}

person.first_name = 'John'
person.last_name = 'Doe'

print(person.__dict__) # {'first_name': 'John', 'last_name': 'Doe'}
```

Output:

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

The __get__ method

The following shows the `__get__` method of the `RequiredString` class:

```
def __get__(self, instance, owner):
    print(f'__get__ was called with instance={instance} and owner={owner}')
    if instance is None:
        return self

    return instance.__dict__[self.property_name] or None
```

Python calls the `__get__` method of the `Person` ’s object when you access the `first_name` attribute. For example:

```
person = Person()
```

```
person.first_name = 'John'
print(person.first_name)
```

Output:

```
__set__ was called with instance=<__main__.Person object at 0x000001F85F7167F0> a
__get__ was called with instance=<__main__.Person object at 0x000001F85F7167F0> a
```

The `__get__` method returns the descriptor if the `instance` is `None`. For example, if you access the `first_name` or `last_name` from the `Person` class, you'll see the descriptor object:

```
print(Person.first_name)
```

Output:

```
<__main__.RequiredString object at 0x000001AF1DA147F0>
```

If the `instance` is not `None`, the `__get__()` method returns the value of the attribute with the name `property_name` of the `instance` object.

Summary

- Descriptors are objects of class that implements one of the method in the descriptor protocol including `__set__`, `__get__`, `__del__`