Python Metaclass Example

Summary: in this tutorial, you'll learn about a Python metaclass example that creates classes with many features.

Introduction to the Python metaclass example

The following defines a `Person` class with two attributes `name` and `age` :

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value
    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        self._age = value

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age

    def __hash__(self):
        return hash(f'{self.name, self.age}')

    def __str__(self):
        return f'Person(name={self.name},age={self.age})'

    def __repr__(self):
        return f'Person(name={self.name},age={self.age})'
```

Typically, when defining a new class, you need to:

- Define a list of object’s properties.
- Define an `__init__` (https://www.pythontutorial.net/python-oop/python-__init__/) method to initialize object’s attributes.
- Implement the `__str__` (https://www.pythontutorial.net/python-oop/python-__str__/) and `__repr__` (https://www.pythontutorial.net/python-oop/python-__repr__/) methods to represent the objects in human-readable and machine-readable formats.
- Implement the `__eq__` (https://www.pythontutorial.net/python-oop/python-__eq__/) method to compare objects by values of all properties.
- Implement the `__hash__` (https://www.pythontutorial.net/python-oop/python-__hash__/) method to use the objects of the class as keys of a dictionary (https://www.pythontutorial.net/python-basics/python-

dictionary/) or elements of a set (https://www.pythontutorial.net/python-basics/python-set/) .

As you can see, it requires a lot of code.

Imagine you want to define a Person class like this and automagically has all the functions above:

```
class Person:
    props = ['first_name', 'last_name', 'age']
```

To do that, you can use a metaclass.

Define a metaclass

First, define the `Data` metaclass that inherits from the `type` (https://www.pythontutorial.net/python-oop/python-type-class/) class:

```
class Data(type):
    pass
```

Second, override the `__new__` (https://www.pythontutorial.net/python-oop/python-__new__/) method to return a new class object:

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)
        return class_obj
```

Note that the `__new__` method is a static method (https://www.pythontutorial.net/python-oop/python-static-methods/) of the `Data` metaclass. And you don't need to use the `@staticmethod` decorator because Python treats it special.

Also, the `__new__` method creates a new class like the `Person` class, not the instance of the `Person` class.

Create property objects

First, define a `Prop` class that accepts an attribute name and contains three methods for creating a property object(`set` , `get` , and `delete`). The `Data` metaclass will use this `Prop` class for adding property objects to the class.

```
class Prop:
    def __init__(self, attr):
        self._attr = attr

    def get(self, obj):
        return getattr(obj, self._attr)

    def set(self, obj, value):
        return setattr(obj, self._attr, value)

    def delete(self, obj):
        return delattr(obj, self._attr)
```

Second, create a new static method `define_property()` that creates a property object for each attribute from the `props` list:

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)
        Data.define_property(class_obj)

        return class_obj

    @staticmethod
    def define_property(class_obj):
        for prop in class_obj.props:
            attr = f'_{prop}'
            prop_obj = property(
                fget=Prop(attr).get,
                fset=Prop(attr).set,
                fdel=Prop(attr).delete
            )
            setattr(class_obj, prop, prop_obj)

        return class_obj
```

The following defines the `Person` class that uses the `Data` metaclass:

```
class Person(metaclass=Data):
    props = ['name', 'age']
```

The `Person` class has two properties `name` and `age` :

```
pprint(Person.__dict__)
```

Output:

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Person' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              'age': <property object at 0x000002213CA92090>,
              'name': <property object at 0x000002213C772A90>,
              'props': ['name', 'age']})
```

Define __init__ method

The following defines an `init` static method and assign it to the `__init__` attribute of the class object:

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)

        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))

        return class_obj

    @staticmethod
    def init(class_obj):
        def _init(self, *obj_args, **obj_kwargs):
            if obj_kwargs:
                for prop in class_obj.props:
                    if prop in obj_kwargs.keys():
                        setattr(self, prop, obj_kwargs[prop])

            if obj_args:
                for kv in zip(class_obj.props, obj_args):
                    setattr(self, kv[0], kv[1])

        return _init

    # more methods
```

The following creates a new instance of the `Person` class and initialize its attributes:

```
p = Person('John Doe', age=25)
print(p.__dict__)
```

Output:

```
{'_age': 25, '_name': 'John Doe'}
```

The `p.__dict__` contains two attributes `_name` and `_age` based on the predefined names in the `props` list.

Define __repr__ method

The following defines the `repr` static method that returns a function and uses it for the `__repr__` attribute of the class object:

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)

        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))

        # define __repr__
        setattr(class_obj, '__repr__', Data.repr(class_obj))

        return class_obj

    @staticmethod
    def repr(class_obj):
        def _repr(self):
            prop_values = (getattr(self, prop) for prop in class_obj.props)
            prop_key_values = (f'{key}={value}' for key, value in zip(class_obj.p
            prop_key_values_str = ', '.join(prop_key_values)
            return f'{class_obj.__name__}({prop_key_values_str})'

        return _repr
```

The following creates a new instance of the `Person` class and displays it:

```
p = Person('John Doe', age=25)
print(p)
```

Output:

```
Person(name=John Doe, age=25)
```

Define __eq__ and __hash__ methods

The following defines the `eq` and `hash` methods and assigns them to the `__eq__` and `__hash__` of the class object of the metaclass:

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)

        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))

        # define __repr__
        setattr(class_obj, '__repr__', Data.repr(class_obj))

        # define __eq__ & __hash__
        setattr(class_obj, '__eq__', Data.eq(class_obj))
        setattr(class_obj, '__hash__', Data.hash(class_obj))

        return class_obj

    @staticmethod
    def eq(class_obj):
        def _eq(self, other):
            if not isinstance(other, class_obj):
                return False

            self_values = [getattr(self, prop) for prop in class_obj.props]
            other_values = [getattr(other, prop) for prop in other.props]

            return self_values == other_values

        return _eq

    @staticmethod
    def hash(class_obj):
        def _hash(self):
            values = (getattr(self, prop) for prop in class_obj.props)
            return hash(tuple(values))

        return _hash
```

The following creates two instances of the Person and compares them. If the values of all properties are the same, they will be equal. Otherwise, they will not be equal:

```
p1 = Person('John Doe', age=25)
p2 = Person('Jane Doe', age=25)

print(p1 == p2)  # False
```

```
p2.name = 'John Doe'
print(p1 == p2)  # True
```

Put it all together

```
from pprint import pprint


class Prop:
    def __init__(self, attr):
        self._attr = attr

    def get(self, obj):
        return getattr(obj, self._attr)

    def set(self, obj, value):
        return setattr(obj, self._attr, value)

    def delete(self, obj):
        return delattr(obj, self._attr)


class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)

        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))
```

```python
        # define __repr__
        setattr(class_obj, '__repr__', Data.repr(class_obj))

        # define __eq__ & __hash__
        setattr(class_obj, '__eq__', Data.eq(class_obj))
        setattr(class_obj, '__hash__', Data.hash(class_obj))

        return class_obj

    @staticmethod
    def eq(class_obj):
        def _eq(self, other):
            if not isinstance(other, class_obj):
                return False

            self_values = [getattr(self, prop) for prop in class_obj.props]
            other_values = [getattr(other, prop) for prop in other.props]

            return self_values == other_values

        return _eq

    @staticmethod
    def hash(class_obj):
        def _hash(self):
            values = (getattr(self, prop) for prop in class_obj.props)
            return hash(tuple(values))

        return _hash

    @staticmethod
    def repr(class_obj):
        def _repr(self):
            prop_values = (getattr(self, prop) for prop in class_obj.props)
            prop_key_values = (f'{key}={value}' for key, value in zip(class_obj.p
            prop_key_values_str = ', '.join(prop_key_values)
            return f'{class_obj.__name__}({prop_key_values_str})'

        return _repr

    @staticmethod
    def init(class_obj):
        def _init(self, *obj_args, **obj_kwargs):
            if obj_kwargs:
                for prop in class_obj.props:
                    if prop in obj_kwargs.keys():
                        setattr(self, prop, obj_kwargs[prop])

            if obj_args:
                for kv in zip(class_obj.props, obj_args):
                    setattr(self, kv[0], kv[1])

        return _init

    @staticmethod
    def define_property(class_obj):
        for prop in class_obj.props:
            attr = f'_{prop}'
            prop_obj = property(
                fget=Prop(attr).get,
                fset=Prop(attr).set,
                fdel=Prop(attr).delete
            )
            setattr(class_obj, prop, prop_obj)

        return class_obj
```

```
class Person(metaclass=Data):
    props = ['name', 'age']


if __name__ == '__main__':
    pprint(Person.__dict__)

    p1 = Person('John Doe', age=25)
    p2 = Person('Jane Doe', age=25)

    print(p1 == p2)  # False

    p2.name = 'John Doe'
    print(p1 == p2)  # True
```

Decorator

The following defines a class called `Employee` that uses the `Data` metaclass:

```
class Employee(metaclass=Data):
    props = ['name', 'job_title']


if __name__ == '__main__':
    e = Employee(name='John Doe', job_title='Python Developer')
    print(e)
```

Output:

```
Employee(name=John Doe, job_title=Python Developer)
```

It works as expected. However, specifying the metaclass is quite verbose. To improve this, you can use a function decorator (https://www.pythontutorial.net/advanced-python/python-decorators/) .

First, define a function decorator that returns a new class which is an instance of the `Data` metaclass:

```
def data(cls):
    return Data(cls.__name__, cls.__bases__, dict(cls.__dict__))
```

Second, use the `@data` decorator for any class that uses the `Data` as the metaclass:

```
@data
class Employee:
    props = ['name', 'job_title']
```

The following shows the complete code:

```
class Prop:
    def __init__(self, attr):
        self._attr = attr

    def get(self, obj):
        return getattr(obj, self._attr)

    def set(self, obj, value):
        return setattr(obj, self._attr, value)

    def delete(self, obj):
        return delattr(obj, self._attr)


class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)
        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))

        # define __repr__
        setattr(class_obj, '__repr__', Data.repr(class_obj))

        # define __eq__ & __hash__
        setattr(class_obj, '__eq__', Data.eq(class_obj))
        setattr(class_obj, '__hash__', Data.hash(class_obj))

        return class_obj

    @staticmethod
    def eq(class_obj):
        def _eq(self, other):
            if not isinstance(other, class_obj):
                return False

            self_values = [getattr(self, prop) for prop in class_obj.props]
            other_values = [getattr(other, prop) for prop in other.props]

            return self_values == other_values

        return _eq

    @staticmethod
    def hash(class_obj):
        def _hash(self):
            values = (getattr(self, prop) for prop in class_obj.props)
            return hash(tuple(values))
        return _hash

    @staticmethod
    def repr(class_obj):
        def _repr(self):
            prop_values = (getattr(self, prop) for prop in class_obj.props)
            prop_key_values = (f'{key}={value}' for key, value in zip(class_obj.p
            prop_key_values_str = ', '.join(prop_key_values)
            return f'{class_obj.__name__}({prop_key_values_str})'

        return _repr

    @staticmethod
    def init(class_obj):
        def _init(self, *obj_args, **obj_kwargs):
            if obj_kwargs:
                for prop in class_obj.props:
                    if prop in obj_kwargs.keys():
                        setattr(self, prop, obj_kwargs[prop])

            if obj_args:
                for kv in zip(class_obj.props, obj_args):
                    setattr(self, kv[0], kv[1])

        return _init

    @staticmethod
    def define_property(class_obj):
        for prop in class_obj.props:
            attr = f'_{prop}'
            prop_obj = property(
                fget=Prop(attr).get,
                fset=Prop(attr).set,
                fdel=Prop(attr).delete
            )
            setattr(class_obj, prop, prop_obj)

        return class_obj


class Person(metaclass=Data):
    props = ['name', 'age']


def data(cls):
    return Data(cls.__name__, cls.__bases__, dict(cls.__dict__))


@data
class Employee:
    props = ['name', 'job_title']
```

Python 3.7 provided a `@dataclass` decorator specified in the PEP 557 (https://www.python.org/dev/peps/pep-0557/) that has some features like the `Data` metaclass. Also, the dataclass (https://www.pythontutorial.net/python-oop/python-dataclass/) offers more features that help you save time when working with classes.