Theory: Magic methods

26 minutes 7 / 7 problems solved

Start practicing

5597 users solved this topic. Latest completion was about 9 hours ago.

Current topic:

✓ Magic methods 3★ ···

Topic depends on:

✓ Methods and attributes 3★ ···

Topic is required for:

Inheritance ···

Math magic methods ···

Decorators in OOP ···

Table of contents:

There are different ways to enrich the functionality of your classes in Python. One of them is creating custom methods which you've already learned about. Another way, the one that we'll cover in this topic, is using "**magic**" **methods.**

§1. What are "magic" methods?

Magic methods are special methods that make using your objects much easier. They are recognizable in the code of the class definitions because they are enclosed in **double underscores:** `__init__` , for example, is one of those "magic" methods in Python. Since they are characterized by double underscores they are often called **dunders.**

Dunders are not meant to be invoked directly by you or the user of your class, it happens internally on a certain action. For example, we do not explicitly call the `__init__` method when we create a new object of the class, but instead, this method is invoked internally. All we need to do is to define the method inside the class in a way that makes sense for our project.

There are many different dunders that you can use, but in this topic, we will focus on the most helpful ones.

§2. __new__ vs __init__

So far we've been calling `__init__` the constructor of the class, but in reality, it is its initializer. New objects of the class are in fact created by the `__new__` method that in its turn calls the `__init__` method.

The first argument of the `__new__` method is `cls`. It represents the class itself, similar to how `self` represents an instance of the class. This also makes `__new__` a different kind of method since it doesn't require an instance of the class. This makes sense since it is supposed to ***create*** those instances. The method returns a new instance of the class which is then passed to the `__init__` method.

Usually, there is no need to define a special `__new__` method, but it can be useful if we want to return instances of other classes or restrict the number of objects in our class.

Imagine, for example, that we want to create a class `Sun` and make sure that we create only one object of this class. We would need to define a class variable that would track the number of instances in the class and forbid the creation of new ones if the limit has been reached.

```
class Sun:
    n = 0  # number of instances of this class

    def __new__(cls):
        if cls.n == 0:
            cls.n += 1
            return object.__new__(cls)  # create new object of the class
```

The code above may be a bit unexpected so let's analyze it. We first check that the class variable `n` has a value of zero. If it does, it means that no instances of the class have been created and we can do that. We then update the class variable and call `__new__` method of `object` class which allows us to create a new instance.

If we now try to create 2 objects of this class we will not succeed:

```
sun1 = Sun()
sun2 = Sun()

print(sun1)  # <__main__.Sun object at 0x1106884a8>
print(sun2)  # None
```

§3. __str__ vs __repr__

Printing out information and data is very important when programming. You can print the results of calculations for yourself or the user of your program, find the mistakes in the code or print out messages.

For example, let's consider the class `Transaction`:

```
class Transaction:
    def __init__(self, number, funds, status="active"):
        self.number = number
        self.funds = funds
        self.status = status
```

If we create a transaction and try to print it out we will not get what we want:

```
payment = Transaction("000001", 9999.999)
print(payment)
# example of the output: <__main__.Transaction object at 0x11068f5f8>
```

Instead of the values that we would like to see, we get information about the object itself. This can be altered if we deal with `__str__` or `__repr__` methods.

As the names suggest, `__str__` defines the behavior of the `str()` function and `__repr__` defines the `repr()` function. A general rule with the `__str__` and `__repr__` methods is that the output of the `__str__` should be ***highly readable*** and the output of the `__repr__` should be ***unambiguous***. In other words, `__str__` creates a representation for users and `__repr__` creates a representation for developers and debuggers. If possible, `__repr__` should return Python code that could be used to create this object or, at least, a comprehensive description.

Both `__repr__` and `__str__` should return a string object!

A good rule is to always define the `__repr__` method first since it is the method used by developers in debugging. It is also a fallback method for `__str__` which means that if the `__str__` method isn't defined, in the situations where it's needed, the `__repr__` will be called instead. This is, for example, the case with `print()`.

In our example here, let's create the `__repr__` method that would create an unambiguous representation of the transaction and all its attributes.

```
class Transaction:
    def __init__(self, number, funds, status="active"):
        self.number = number
        self.funds = funds
        self.status = status

    def __repr__(self):
        return "Transaction({}, {})".format(self.number, self.funds)
```

Now if we try to print any transaction we will get a standard readable string:

```
payment = Transaction("000001", 9999.999)
print(payment)
# Transaction(000001, 9999.999)
```

You can see that we've called `print` and got the representation from `__repr__`. Now let's add `__str__` and see if things change.

```python
class Transaction:
    def __init__(self, number, funds, status="active"):
        self.number = number
        self.funds = funds
        self.status = status

    def __repr__(self):
        return "Transaction({}, {})".format(self.number, self.funds)


    def __str__(self):

        return "Transaction {} for {} ({})".format(self.number, self.funds, self.status)




payment = Transaction("000001", 9999.999)

print(payment)

# Transaction 000001 for 9999.999 (active)

print(repr(payment))

# Transaction(000001, 9999.999)
```

Now that we have `__str__`, when we call `print`, we get the representation defined there. To see the "official" representation we need to directly call the `repr` function.

§4. Summary

Magic methods are said to add "magic" to your classes and that is somewhat true. Dunders really make working with classes much easier and far more efficient.

In this topic, we've covered only a couple of these magic methods. We highly encourage you to look them up (for example, in "A Guide to Python's Magic Methods" by Rafe Kettler) and try them out in your projects. As for the magic methods for arithmetics and comparisons, we'll look into them in another topic!

Report a typo

Thanks for your feedback!

Write here how we could improve this theory

Start practicing

Comments (16) Hints (1) Useful links (1) Show discussion