Python Classes With Dunder Methods

Introduction

In Python, special methods are a set of predefined methods you can use to enrich your classes. They are easy to recognize because they start and end with double underscores, for example __init__ or __str__.

As it quickly became tiresome to say under-under-method-under-under Pythonistas adopted the term “dunder methods”, a short form of “double under.”

Object Initialization: __init__

Right upon starting my class I already need a special method. To construct account objects from the Account class I need a constructor which in Python is the __init__ dunder:

```
class Account:
    """A simple account class"""

    def __init__(self, owner, amount=0):
        """
        This is the constructor that lets us create
        objects from this class
        """
        self.owner = owner
        self.amount = amount
        self._transactions = []
```

The constructor takes care of setting up the object. In this case it receives the owner name, an optional start amount and defines an internal transactions list to keep track of deposits and withdrawals.

This allows us to create new accounts like this:

```
>>> acc = Account('bob')  # default amount = 0
>>> acc = Account('bob', 10)
```

Object Representation: __str__, __repr__

It's common practice in Python to provide a string representation of your object for the consumer of your class (a bit like API documentation.) There are two ways to do this using dunder methods:

1. **__repr__**: The "official" string representation of an object. This is how you would make an object of the class. The goal of __repr__ is to be unambiguous.
2. **__str__**: The "informal" or nicely printable string representation of an object. This is for the end user.

Let's implement these two methods on the Account class:

```
class Account:
    # ... (see above)

    def __repr__(self):
        return 'Account({!r}, {!r})'.format(self.owner, self.amount)

    def __str__(self):
        return 'Account of {} with starting amount: {}'.format(
            self.owner, self.amount)
```

If you don't want to hardcode "Account" as the name for the class you can also use self.__class__.__name__ to access it programmatically.

If you wanted to implement just one of these *to-string* methods on a Python class, make sure it's __repr__ , because __str__ call falls back to __repr__ .

Now I can query the object in various ways and always get a nice string representation:

```
>>> str(acc)
'Account of bob with starting amount: 10'

>>> print(acc)
"Account of bob with starting amount: 10"

>>> repr(acc)
"Account('bob', 10)"
```

Note:

The __str__ method is what happens when you print anything, and the __repr__ method is what happens when you use the repr() function (or when you look at it with the interactive prompt).

If no __str__ method is given, Python will print the result of __repr__ instead. If you define __str__ but not __repr__, Python will use what you see above as the __repr__, but still use __str__ for printing.

Iteration: __len__, __getitem__, __reversed__

So first, I'll define a simple method to add transactions. I'll keep it simple because this is just setup code to explain dunder methods, and not a production-ready accounting system:

```
def add_transaction(self, amount):
    if not isinstance(amount, int):
        raise ValueError('please use int for amount')
    self._transactions.append(amount)
```

I also defined a property to calculate the balance on the account so I can conveniently access it with account.balance. This method takes the start amount and adds a sum of all the transactions:

```
@property
def balance(self):
    return self.amount + sum(self._transactions)
```

Let's do some deposits and withdrawals on the account:

```
>>> acc = Account('bob', 10)

>>> acc.add_transaction(20)
>>> acc.add_transaction(-10)
>>> acc.add_transaction(50)
>>> acc.add_transaction(-20)
>>> acc.add_transaction(30)

>>> acc.balance
80
```

Now I have some data and I want to know:

1. How many transactions were there?
2. Index the account object to get transaction number …
3. Loop over the transactions

With the class definition I have this is currently not possible. All of the following statements raise TypeError exceptions:

```
>>> len(acc)
TypeError

>>> for t in acc:
...     print(t)
TypeError

>>> acc[1]
TypeError
```

Dunder methods to the rescue! It only takes a little bit of code to make the class iterable:

```python
class Account:
    # ... (see above)

    def __len__(self):
        return len(self._transactions)

    def __getitem__(self, position):
        return self._transactions[position]
```

Now the previous statements work:

```python
>>> len(acc)
5

>>> for t in acc:
...     print(t)
20
-10
50
-20
30

>>> acc[1]
-10
```

To iterate over transactions in reversed order you can implement the __reversed__ special method:

```
def __reversed__(self):
    return self[::-1]

>>> list(reversed(acc))
[30, -20, 50, -10, 20]
```

To reverse the list of transactions I used Python’s reverse list slice syntax.

Operator Overloading for Merging Accounts: __add__

In Python, everything is an object. We are completely fine adding two integers or two strings with the + (plus) operator, it behaves in expected ways:

```
>>> 1 + 2
3

>>> 'hello' + ' world'
'hello world'
```

Again, we see polymorphism at play: Did you notice how + behaves different depending the type of the object? For integers it sums, for strings it concatenates. Again doing a quick dir() on the object reveals the corresponding "dunder" interface into the data model:

```
>>> dir(1)
[...
'__add__',
...
'__radd__',
...]
```

Our Account object does not support addition yet, so when you try to add two instances of it there's a TypeError:

```
>>> acc + acc2
TypeError: "unsupported operand type(s) for +: 'Account' and 'Account'"
```

Let's implement __add__ to be able to merge two accounts. The expected behavior would be to merge all attributes together: the owner name, as well as starting amounts and transactions. To do this we can benefit from the iteration support we implemented earlier:

```
def __add__(self, other):
    owner = '{}&{}'.format(self.owner, other.owner)
    start_amount = self.amount + other.amount
    acc = Account(owner, start_amount)
    for t in list(self) + list(other):
        acc.add_transaction(t)
    return acc
```

Yes, it is a bit more involved than the other dunder implementations so far. It should show you though that you are in the driver's seat. You can implement addition however you please. If we wanted to ignore historic transactions—fine, you can also implement it like this:

```
def __add__(self, other):
    owner = self.owner + other.owner
    start_amount = self.balance + other.balance
    return Account(owner, start_amount)
```

I think the former implementation would be more realistic though, in terms of what a consumer of this class would expect to happen.

Now we have a new merged account with starting amount $110 (10 + 100) and balance of $240 (80 + 160):

```
>>> acc3 = acc2 + acc
>>> acc3
Account('tim&bob', 110)

>>> acc3.amount
110
>>> acc3.balance
240
>>> acc3._transactions
[20, 40, 20, -10, 50, -20, 30]
```

Note this works in both directions because we're adding objects of the same type. In general, if you would add your object to a builtin (int, str, ...) the __add__ method of the builtin wouldn't know anything about your object. In that case you need to implement the reverse add method (__radd__) as well.

Summary of magic methods

Magic Method	When it gets invoked
`__new__(cls [,...])`	`instance = MyClass(arg1, arg2)`
`__init__(self [,...])`	`instance = MyClass(arg1, arg2)`
`__cmp__(self, other)`	`self == other`, `self > other`, etc.
`__pos__(self)`	`+self`
`__neg__(self)`	`-self`
`__invert__(self)`	`~self`
`__index__(self)`	`x[self]`
`__nonzero__(self)`	`bool(self)`
`__getattr__(self, name)`	`self.name # name doesn't exist`
`__setattr__(self, name, val)`	`self.name = val`
`__delattr__(self, name)`	`del self.name`

Conclusion

I hope you feel a little less afraid of dunder methods after reading this article. A strategic use of them makes your classes more Pythonic, because they emulate builtin types with Python-like behaviors.