**Enriching Your Python Classes With Dunder (Magic, Special) Methods**

What Python’s “magic methods” are and how you would use them to make a simple account class more Pythonic.

## What Are Dunder Methods?

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.”

These “dunders” or “special methods” in Python are also sometimes called “magic methods.” But using this terminology can make them seem more complicated than they really are—at the end of the day there’s nothing “magical” about them. [You should treat these methods like a normal language feature.](http://www.pixelmonkey.org/2013/04/11/python-double-under-double-wonder)

Dunder methods let you emulate the behavior of built-in types. For example, to get the length of a string you can call len('string'). But an empty class definition doesn’t support this behavior out of the box:

class NoLenSupport:

pass

>>> obj = NoLenSupport()

>>> len(obj)

TypeError: "object of type 'NoLenSupport' has no len()"

To fix this, you can add a \_\_len\_\_ dunder method to your class:

class LenSupport:

def \_\_len\_\_(self):

return 42

>>> obj = LenSupport()

>>> len(obj)

42

Another example is slicing. You can implement a \_\_getitem\_\_ method which allows you to use Python’s list slicing syntax: obj[start:stop].

## Special Methods and the Python Data Model

This elegant design is known as the [Python data model](https://docs.python.org/3/reference/datamodel.html) and lets developers tap into rich language features like sequences, iteration, operator overloading, attribute access, etc.

You can see Python’s data model as a powerful API you can interface with by implementing one or more dunder methods. If you want to write more Pythonic code, knowing how and when to use dunder methods is an important step.

For a beginner this might be slightly overwhelming at first though. No worries, in this article I will guide you through the use of dunder methods using a simple Account class as an example.

## Enriching a Simple Account Class

Throughout this article I will enrich a simple Python class with various dunder methods to unlock the following language features:

* Initialization of new objects
* Object representation
* Enable iteration
* Operator overloading (comparison)
* Operator overloading (addition)
* Method invocation
* Context manager support (with statement)

You can find the final code example [here](https://github.com/pybites/dunders/blob/master/account.py). I’ve also put together a [Jupyter notebook](https://github.com/pybites/dunders/blob/master/richer-classes-with-dunders.ipynb) so you can more easily play with the examples.

## 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](https://dbader.org/blog/python-repr-vs-str) (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 enduser.

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\_\_](https://dbader.org/blog/python-repr-vs-str).

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)"

## Iteration: \_\_len\_\_, \_\_getitem\_\_, \_\_reversed\_\_

In order to iterate over our account object I need to add some transactions. 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](https://pybit.es/property-decorator.html) 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:

class Account:

# ... (see above)

def \_\_len\_\_(self):

return len(self.\_transactions)

def \_\_getitem\_\_(self, position):

return self.\_transactions[position]

Now the previous statements work:

>>> 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](https://www.youtube.com/watch?v=sGhY8dQdu4A) syntax. I also had to wrapp the result of reversed(acc) in a list() call because reversed() returns a a [reverse iterator](https://dbader.org/blog/python-reverse-list), not a list object we can print nicely in the REPL. Check out [this tutorial on iterators in Python](https://dbader.org/blog/python-iterators) if you’d like to learn more about how this approach works in detail.

All in all, this account class is starting to look quite Pythonic to me now.

## Operator Overloading for Comparing Accounts: \_\_eq\_\_, \_\_lt\_\_

We all write dozens of statements daily to compare Python objects:

>>> 2 > 1

True

>>> 'a' > 'b'

False

This feels completely natural, but it’s actually quite amazing what happens behind the scenes here. Why does > work equally well on integers, strings and other objects (as long as they are the same type)? This polymorphic behavior is possible because these objects implement one or more comparison dunder methods.

An easy way to verify this is to use the dir() builtin:

>>> dir('a')

['\_\_add\_\_',

...

'\_\_eq\_\_', <---------------

'\_\_format\_\_',

'\_\_ge\_\_', <---------------

'\_\_getattribute\_\_',

'\_\_getitem\_\_',

'\_\_getnewargs\_\_',

'\_\_gt\_\_', <---------------

...]

Let’s build a second account object and compare it to the first one (I am adding a couple of transactions for later use):

>>> acc2 = Account('tim', 100)

>>> acc2.add\_transaction(20)

>>> acc2.add\_transaction(40)

>>> acc2.balance

160

>>> acc2 > acc

TypeError:

"'>' not supported between instances of 'Account' and 'Account'"

What happened here? We got a TypeError because I have not implemented any comparison dunders nor inherited them from a parent class.

Let’s add them. To not have to implement all of the comparison dunder methods, I use the [functools.total\_ordering](https://docs.python.org/2/library/functools.html#functools.total_ordering) decorator which allows me to take a shortcut, only implementing \_\_eq\_\_ and \_\_lt\_\_:

from functools import total\_ordering

@total\_ordering

class Account:

# ... (see above)

def \_\_eq\_\_(self, other):

return self.balance == other.balance

def \_\_lt\_\_(self, other):

return self.balance < other.balance

And now I can compare Account instances no problem:

>>> acc2 > acc

True

>>> acc2 < acc

False

>>> acc == acc2

False

## Operator Overloading for Merging Accounts: \_\_add\_\_

In Python, [everything is an object](https://dbader.org/blog/python-first-class-functions). 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. You can see an example for that [here](http://www.marinamele.com/2014/04/modifying-add-method-of-python-class.html).

## Callable Python Objects: \_\_call\_\_

You can make an object callable like a regular function by adding the \_\_call\_\_ dunder method. For our account class we could print a nice report of all the transactions that make up its balance:

class Account:

# ... (see above)

def \_\_call\_\_(self):

print('Start amount: {}'.format(self.amount))

print('Transactions: ')

for transaction in self:

print(transaction)

print('\nBalance: {}'.format(self.balance))

Now when I call the object with the double-parentheses acc() syntax, I get a nice account statement with an overview of all transactions and the current balance:

>>> 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()

Start amount: 10

Transactions:

20

-10

50

-20

30

Balance: 80

Please keep in mind that this is just a toy example. A “real” account class probably wouldn’t print to the console when you use the function call syntax on one of its instances. In general, the downside of having a \_\_call\_\_ method on your objects is that it can be hard to see what the purpose of calling the object is.

Most of the time it’s therefore better to add an explicit method to the class. In this case it probably would’ve been more transparent to have a separate Account.print\_statement() method.

## Context Manager Support and the With Statement: \_\_enter\_\_, \_\_exit\_\_

My final example in this tutorial is about a slightly more advanced concept in Python: Context managers and adding support for the with statement.

Now, [what is a “context manager” in Python](https://dbader.org/blog/python-context-managers-and-with-statement)? Here’s a quick overview:

A context manager is a simple “protocol” (or interface) that your object needs to follow so it can be used with the with statement. Basically all you need to do is add \_\_enter\_\_ and \_\_exit\_\_ methods to an object if you want it to function as a context manager.

Let’s use context manager support to add a rollback mechanism to our Account class. If the balance goes negative upon adding another transaction we rollback to the previous state.

We can leverage the Pythonic with statement by adding two more dunder methods. I’m also adding some print calls to make the example clearer when we demo it:

class Account:

# ... (see above)

def \_\_enter\_\_(self):

print('ENTER WITH: Making backup of transactions for rollback')

self.\_copy\_transactions = list(self.\_transactions)

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

print('EXIT WITH:', end=' ')

if exc\_type:

self.\_transactions = self.\_copy\_transactions

print('Rolling back to previous transactions')

print('Transaction resulted in {} ({})'.format(

exc\_type.\_\_name\_\_, exc\_val))

else:

print('Transaction OK')

As an exception has to be raised to trigger a rollback, I define a quick helper method to validate the transactions in an account:

def validate\_transaction(acc, amount\_to\_add):

with acc as a:

print('Adding {} to account'.format(amount\_to\_add))

a.add\_transaction(amount\_to\_add)

print('New balance would be: {}'.format(a.balance))

if a.balance < 0:

raise ValueError('sorry cannot go in debt!')

Now I can use an Account object with the with statement. When I make a transaction to add a positive amount, all is good:

acc4 = Account('sue', 10)

print('\nBalance start: {}'.format(acc4.balance))

validate\_transaction(acc4, 20)

print('\nBalance end: {}'.format(acc4.balance))

Executing the above Python snippet produces the following printout:

Balance start: 10

ENTER WITH: Making backup of transactions for rollback

Adding 20 to account

New balance would be: 30

EXIT WITH: Transaction OK

Balance end: 30

However when I try to withdraw too much money, the code in \_\_exit\_\_ kicks in and rolls back the transaction:

acc4 = Account('sue', 10)

print('\nBalance start: {}'.format(acc4.balance))

try:

validate\_transaction(acc4, -50)

except ValueError as exc:

print(exc)

print('\nBalance end: {}'.format(acc4.balance))

In this case we get a different result:

Balance start: 10

ENTER WITH: Making backup of transactions for rollback

Adding -50 to account

New balance would be: -40

EXIT WITH: Rolling back to previous transactions

ValueError: sorry cannot go in debt!

Balance end: 10