Published in The Startup

Andrew Scott Follow
Dec 30, 2019 · 4 min read ★ · Listen

Save

Welcome to Python, Meet the Dunders

A quick introduction to several of the __magic__ methods in python

What is a Dunder?

"Dunder" method name is the common pronunciation for python's built-in method names that start and end with double underscores. Since "Dunder" is easier to say than "double underscore", the name stuck. These methods are also sometimes referred to as

While we sometimes treat these methods as a special language feature, there's really nothing special about them. An introductory understanding of how to use dunder methods, and which ones are available can help you create much more intuitive and easy to use classes.

In this post we'll look at several of the dunder methods used for object creation, representation, and comparison. In later posts we will explore other dunder methods such as emulating containers, emulating numeric types, and more.

The Basics

If you've written any python you've almost definitely used at least one dunder method, whether you realized it or not.

__init__

`__init__` is a method you've almost certainly used before. It is the dunder method used to initialize a new object. You may not have been aware of this, but when creating a new instance of an object, the dunder method `__new__` is actually called first to create a new instance of your class, then the `__init__` method is called to initialize that newly created instance.

A few notes about `__init__` ; if you're working with a derived class, you're derived class' `__init__` method must explicitly call the base class' `__init__` method using `super.__init__()` . You must also never return a non- `None` value from `__init__` , as doing so will raise a `TypeError` .

Representation Methods

There are two common, and one less common dunder methods used for representing objects. You've undoubtedly called these methods many times, even if you weren't aware you were using them.

__str__

`__str__` is the most common of the representational methods. The `__str__` method will return an "informal" printable representation of an object and return an str type.




`__str__` method on whatever object you're trying to print. Interestingly enough, if `__str__` is not defined for an object, it will default to calling `__repr__`.

__repr__

You may be less familiar with `__repr__`, but it is the second of the common representational special methods and it is used to return the "official" string representation of an object. If an accurate representation of the object cannot be achieved due to object complexity, returning a useful description of the instance is also acceptable. The `__repr__` method is called by `repr()`.

```
class WidgitWithoutStr:
    """
    A class with no __str__ or __repr__ methods defined.
    """
    def __init__(self, name):
        self.name = name

class WidgitWithStrOnly(WidgitWithoutStr):
    """
    A class with __str__ defined.
    """
    def __str__(self):
        return self.name

class WidgitWithReprOnly(WidgitWithoutStr):
    """
    A class with __repr__ defined.
    """
    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.name)

print(WidgitWithoutStr("Nobody")) # <__main__.WidgitWithoutStr object at 0x10b8c1790>
print(WidgitWithStrOnly("Bob")) # Bob
print(WidgitWithReprOnly("Mary")) # WidgitWithReprOnly(Mary)
```


__bytes__

`__bytes__` computes a byte-string representation of an object and returns an object of

```
        self.name = name

    def __bytes__(self):
        return bytes(self.name, "utf-8")

print(bytes(Widgit("Dave"))) # b'Dave'
print(bytes("Dave", "utf-8")) # b'Dave'
```

bytes_ex.py hosted with ❤ by GitHub view raw

Rich Comparison Methods

The so-called “rich comparison” methods will return a bool type and are used implement the backing methods for comparison operators such as `==` , `!=` , `<` , `>` and more. I won’t go over all of these methods here, since you probably see where this is going, but these are the dunder methods called when you compare objects.

Search this file…

dunder	operator
__eq__	==
__ne__	!=
__lt__	<
__le__	<=
__gt__	>
__ge__	>=

__rich_comparisons__.csv hosted with ❤ by GitHub view raw

__eq__

`__eq__` is the dunder method used for checking equality between objects. This is pretty self-explanatory for strings and numeric objects, but what about more complex objects? The `__eq__` method goes hand in hand with the `__hash__` method, which as its name might suggest, takes a hashed collection (set, dict, etc.) and returns an integer. If a class does not define an `__eq__` method it should also not define a `__hash__` method.

```
class Widgit:
    def __init__(self, name):
```




```
            return self.name == other.name
        return False

print(WidgitWithEq("Bob") == WidgitWithEq("Dave")) # False
print(WidgitWithEq("Bob") == Widgit("Bob")) # False
print(WidgitWithEq("Bob") == "Bob") # False
print(WidgitWithEq("Bob") == WidgitWithEq("Bob")) # True
```

__eq__1.py hosted with ❤ by GitHub view raw

Beware of incomplete implementations of __eq__ and other comparison methods, otherwise you may get unexpected results.

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

class WidgetWithAltEq(Widgit):
    def __eq__(self, other):
        return self.name == other.name

print(WidgetWithAltEq("Bob") == Dog("Bob")) # True
```

__eq__2.py hosted with ❤ by GitHub view raw

__ne__

It's also interesting that you don't actually need to define the `__ne__` in cases where you have already defined `__eq__` as python will just assign `__ne__` to the inverse of `__eq__`. The opposite is actually also true, if `__eq__` is not defined but `__ne__` is, python will assign __eq__ to the inverse of `__ne__` — however strongly it's advised to define both or `__eq__` only to reduce confusion.

```
class WidgitWithEq(Widgit):
    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.name == other.name
        return False

print(WidgitWithEq("Bob") != WidgitWithEq("Dave")) # True
```

   

```
            return self.name != other.name
        return True

print(WidgitWithNe("Bob") != WidgitWithNe("Dave")) # True
print(WidgitWithNe("Bob") != WidgitWithNe("Bob")) # False
```

ne 1.py hosted with ❤ by GitHub view raw

That's it — for now

Hopefully you enjoyed this quick look at some of the common dunder methods. In Part 2 we'll look at attribute access, emulating container types, and emulating numeric types.

Welcome to Python, Meet the Dunders Pt. 2

In Part 2 we'll look at attribute access, emulating container types, and emulating numeric types

medium.com

Welcome to Python, meet the Dunders

In Part 3 we will explore callables, context managers, and other special attributes in python.

medium.com

Hey, I'm Andrew Scott, a software developer and the creator of Ochrona. Ochrona focuses on improving python security by providing insights into your project's dependencies and doing so with a major focus on Developer Experience (DX). Sign up for our Mailing List :)