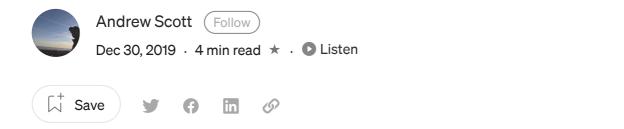


Open in app



Published in The Startup

You have 1 free member-only story left this month. Upgrade for unlimited access.



# 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











Open in app

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 <u>object creation</u>, <u>representation</u>, and <u>comparison</u>. 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.











Open in app

 $\_\_str\_\_$  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:
 2
3
         A class with no __str__ or __repr__ methods defined.
         def __init__(self, name):
             self.name = name
7
     class WidgitWithStrOnly(WidgitWithoutStr):
8
9
         A class with __str__ defined.
10
11
12
         def __str__(self):
             return self.name
13
14
     class WidgitWithReprOnly(WidgitWithoutStr):
15
16
17
         A class with __repr__ defined.
18
         def __repr__(self):
19
             return "{}({})".format(self.__class__.__name__, self.name)
20
21
     print(WidgitWithoutStr("Nobody")) # <__main__.WidgitWithoutStr object at 0x10b8c1790>
22
     print(WidgitWithStrOnly("Bob")) # Bob
     print(WidgitWithReprOnly("Mary")) # WidgitWithReprOnly(Mary)
aku amal uamumii baakad ...ikb 🦰 bi. Cikl liib
```

#### \_\_bytes\_\_

\_bytes\_ computes a byte-string representation of an object and returns an object of

347 | Q 1 | •••









```
Get unlimited access

Open in app

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 \( \bigcip \) by GitHub
```

### **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.



#### \_\_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.

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



Open in app

```
8          return self.name == other.name
9          return False
10
11     print(WidgitWithEq("Bob") == WidgitWithEq("Dave")) # False
12     print(WidgitWithEq("Bob") == Widgit("Bob")) # False
13     print(WidgitWithEq("Bob") == "Bob") # False
14     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.

```
1
    class Dog:
2
        def __init__(self, name):
3
            self.name = name
4
5
    class WidgetWithAltEq(Widgit):
6
        def __eq__(self, other):
7
            return self.name == other.name
8
    print(WidgetWithAltEq("Bob") == Dog("Bob")) # True
9
__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.

```
1  class WidgitWithEq(Widgit):
2   def __eq__(self, other):
3     if isinstance(other, self.__class__):
4         return self.name == other.name
5         return False
6
7   print(WidgitWithEq("Bob") != WidgitWithEq("Dave")) # True
```



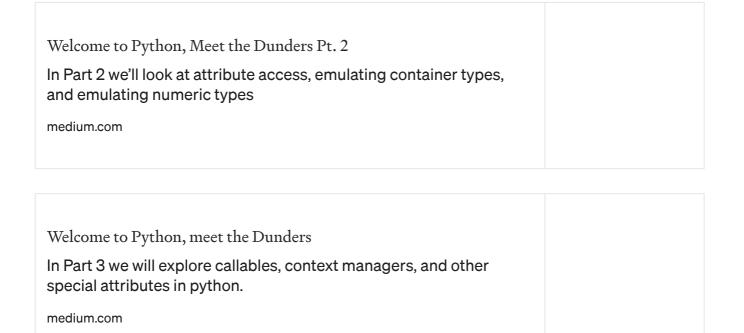






#### That's it — for now

Hopefully you enjoyed this quick look at some of the common dunder methods. In <u>Part 2</u> we'll look at a<u>ttribute access</u>, <u>emulating container types</u>, and <u>emulating numeric types</u>.



Hey, I'm Andrew Scott, a software developer and the creator of <u>Ochrona</u>. <u>Ochrona</u> 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 <u>Mailing List</u>:)











Open in app

# Sign up for Top 5 Stories

By The Startup

Get smarter at building your thing. Join 176,621+ others who receive The Startup's top 5 stories, tools, ideas, books delivered straight into your inbox, once a week. Take a look.

Emails will be sent to bnousilal@gmail.com. Not you?



 $\stackrel{\leftarrow}{\sqsubseteq}^{+}$  Get this newsletter







