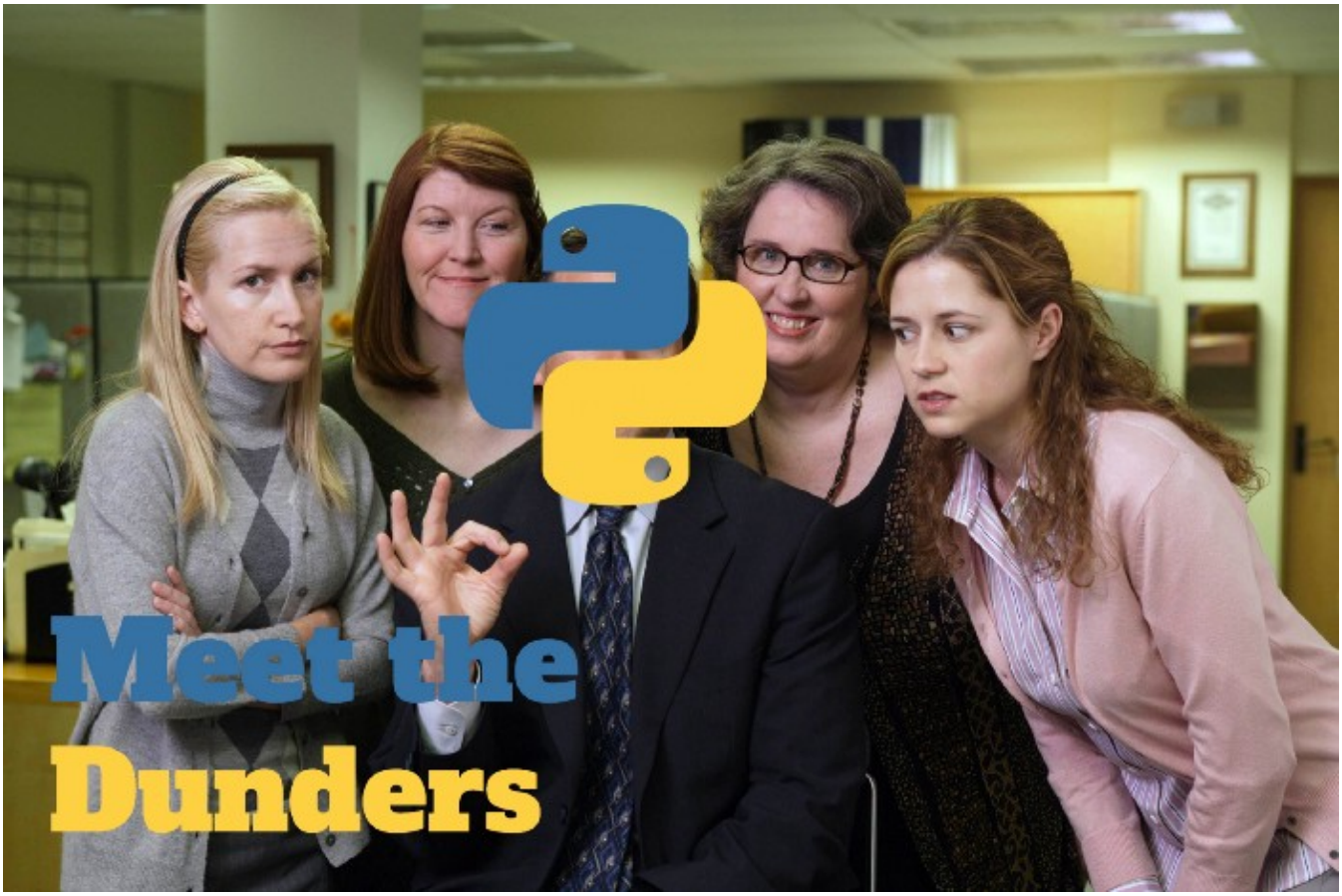
Published in The Startup

Andrew Scott Follow

Dec 31, 2019 · 4 min read ★ · Listen

Welcome to Python, Meet the Dunders Pt. 2

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



If you haven't read it already, check out Part 1 of this series here.

medium.com

Attribute Access Methods

There are a number of different dunder methods that allow customization of behavior for accessing, assigning, and removing class attributes on instances.

Search this file…

dunder	method or description
__getattr__	called when default accessor fails
__getattribute__	called unconditionally to implement attribute accesses for instances of the class
__setattr__	called when an attribute assignment is attempted
__delattr__	called when an attribute deletion is attempted
__dir__	dir()
__get__	called to get the attribute of the owner class or of an instance of that class
__set__	called to set the attribute on an instance instance of the owner class to a new value
__delete__	called to delete the attribute on an instance of the owner class
__set_name__	called at the time the owning class owner is created

accessors.csv hosted with ❤ by GitHub view raw

__getattribute__ & __getattr__

`__getattribute__` and `__getattr__` often cause confusion due to their similar names and behaviors. `__getattribute__` is invokes before actually looking for a property on an object, meaning it can be used to add custom behavior or catch exceptions if they arise. `__getattr__` is special in that it is only ever invoked if called directly by `__getattribute__` or if `__getattribute__` or `__get__` raise an AttributeError. If you want to have any special handling for the case where an attribute isn't found, it should be handled in `__getattr__`.

```
class Dog:
    """
    A classic dog, no frills.
    """
    def __init__(self, name, breed, age):
```

```
        print("__getattribute__({})".format(name))
        return super().__getattribute__(name)

    def __getattr__(self, name):
        print("__getattr__({})".format(name))
        return None

leo = Dog("Leo", "maltese", 1)

print(leo.breed)
# __getattribute__(breed)
# maltese

print(leo.gender)
# __getattribute__(gender)
# __getattr__(gender)
# None
```

__dir__

`__dir__` is called when `dir()` is called on an object. `__dir__` should return a sequence of valid attributes of the object. dir() will convert this sequence into a list and sort it before returning. I won't cover this here, just know if you ever need to override the default `dir()` behavior for a class you can do so with the `__dir__` dunder method.

Container Methods

Python offers a variety of dunder methods for implementing container functionality in custom classes. Container types in python include sequences (lists, tuples, etc.) and mappings (dictionaries, etc.). It should be noted that since there are many of these that many of the methods can be covered by build on top of the Abstract Base Classes for Containers offered in the standard library module collections.abc. Since there's a number of these we'll just look at a few of them.

Search this file…

dunder	method or description
__len__	len()
__length_hint__	length_hint()
__getitem	self[key]

10	__contains__	X in object

__containers__.csv hosted with ❤ by GitHub view raw

__len__

`__len__` is another dunder method you've probably used countless times without realizing it. It is invoked whenever you use the built-in `len()` function on an object.

```
class CustomSeq:
    """
    A custom implementation of a sequence.
    """
    def __init__(self, items):
        self.items = items

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

len(CustomSeq([1,2,3])) # 3
len(CustomSeq("abc4")) # 4
```

__len__.py hosted with ❤ by GitHub view raw

Typically this method is called on a sequence of some sort such as a list, or even a str, but you can also defined this for any object, even if it doesn't necessarily make sense.

```
class Galaxy:
    """
    Guide to the Galaxy.
    """
    def __init__(self, question):
        self.question = question

    def __len__(self):
        return 42

len(Galaxy("What is the answer to life?")) # 42
```

__len__2.py hosted with ❤ by GitHub view raw

care needs to be taken for negative key arguments). If the collection is a mapping it should accept a key value. In the case that the item can not be found an IndexError should be returned for sequences and a KeyError returned from mappings.

```
class FruitColors:
    """
    Stores colors for fruit.
    """
    _colors = {}
    def __init__(self, fruits):
        if isinstance(fruits, dict):
            self._colors = fruits

    def __getitem__(self, key):
        try:
            return self._colors[key]
        except KeyError:
            return "unknown"

my_fruits = FruitColors({"orange": "orange", "apple": "red", "banana": "yellow"})

print(my_fruits["apple"]) # red
print(my_fruits["kiwi"]) # unknown
```

getitem .py hosted with ❤ by GitHub view raw

__contains__

The membership operators (`in` and `not in`) are typically implemented by iterating through a container (complexity **O(n)**), however, if the container class has the `__contains__` method implemented it will use that instead. The assumption is that you would provide this method if there was a more efficient search that could be used to test for membership.

```
class Veggies:
    """
    Just a list of veggies
    """
    _veggies = {}
    def __init__(self, veggies):
```

```
        return key in set(self._veggies)

my_veggies = Veggies(["onion", "beet", "carrot"])

"apple" in my_veggies # False
"beet" in my_veggies # True
```

contains .py hosted with ❤ by GitHub view raw

__iter__

This method is what allows containers to be iterated on. In the event that the container is a sequence it should return an iterator object for all items in the sequence and if the container is a mapping type it should return an iterator of all of the keys.

```
class Dogs:
    """
    An ordered list of dog breeds.
    """
    _breeds = []
    def __init__(self, *breeds):
        self._breeds = [*breeds]
        self._breeds.sort()

    def __iter__(self):
        for breed in self._breeds:
            yield breed

my_favorite_dogs = Dogs("maltese", "french bulldog")

print(type(iter(my_favorite_dogs))) # <class 'generator'>
print([dog for dog in my_favorite_dogs]) # ['french bulldog', 'maltese']
```

iter .py hosted with ❤ by GitHub view raw

Numeric Methods

Out of all of the dunder method classifications, none are as expansive as the numeric methods. These range from everything from supporting addition and subtraction to performing an exclusive OR operation and everything in between.

Search this file…

6	__truediv__	x / y
7	__floordiv__	x // y
8	__mod__	x % y
9	__divmod__	divmod()
10	__pow__	pow() or x**y
11	__lshift__	x << y
12	__rshift__	x >> y
13	__and__	x & y
14	__xor__	x ^ y
15	__or__	x | y
16	__radd__	y + x
17	__rsub__	y - x
18	__rmul__	y * x
19	__rmatmul__	Added for matrix multiplication in python 3.5 using the @ operator
20	__rtruediv__	y / x
21	__rfloordiv__	y // x
22	__rmod__	y % x
23	__rdivmod__	divmod()
24	__rpow__	pow() or y**x
25	__rlshift__	y << x
26	__rrshift__	y >> x
27	__rand__	y & x
28	__rxor__	y ^ x
29	__ror__	y | x
30	__iadd__	x += y
31	__isub__	x -= y
32	__imul__	x *= y
33	__imatmul__	Added for matrix multiplication in python 3.5 using the @ operator
34	__itruediv__	x /= y
35	__ifloordiv__	x //= y
36	__imod__	x %= y
37	__idivmod__	divmod()
38	__ipow__	pow() or x**=y
39	__ilshift__	x <<= y
40	__irshift__	x >>= y
41	__iand__	x &= y
42	__ixor__	x ^= y
43	__ior__	x |= y
44	__neg__	-x
45	__pos__	+x
46	__abs	abs()




	__index__	index()
52	__round__	round()
53	__trunc__	math.trunc()
54	__floor__	math.floor()
55	__ceil__	math.ceil()

__add__

As you may have expected, `__add__` is the dunder method that implements the binary addition operator (i.e. x + y). When it's called it actually structured as `x.__add__(y)` . `__radd__` would handle adding in the opposite order (y + x), though `__radd__` would only be called if the class for object y did not have __add__ implemented. Finally we have `__iadd__` which should assign the result to the first operand and return that object. `__iadd__` can be used to override the `+=` operator.




__pow__

`__pow__` implements the `pow()` method, which can also be expressed as x**y. It works exactly the same way as `__add__`, except rather than performing addition it is raising one operand to the power of the second operand.

__index__

`__index__` was originally added so that any object that implemented it could be used in slicing. However what this really boils down to is that `__index__` must return an integer as they are the only type supported natively in slicing.

That's it — for now

Hopefully you enjoyed this quick look at some of the common dunder methods. In Part 3 we'll look at callables, context managers, and other special attributes.

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 :)