A Guided Tour of Dunder Methods

Tom Clark

Before we start, you will want to clone the repository at
`https://github.com/tclark/dunder-guided-tour`.

Python!

There is plenty to love about Python, but in particular I love its clean and expressive syntax.

Let's take a moment to appreciate how cool Python's strings are.

Strings are great

```
>>> s1 = "Spam"
>>> s2 = "Cheese"
```

We can compare them

```
>>> s1 == s2
False
>>> s1 < s2
False
```

and even do some arithmetic.

```
>>> s1 + s2
'SpamCheese'
>>> 3 * s1
'SpamSpamSpam'
```

Strings are great

```
>>> s = "Spanish Inquisition"
```

Loops!

```
>>> for c in s:
...   print(c)
...
S
p
a
...
```

Strings are great

```
>>> s = "And now for something completely different"
```

We can access individual elements by index.

```
>>> s[4]
'n'
>>> s[-1]
't'
```

We can slice them

```
>>> s[3:10:2]
' o o'
```

Dunder Methods

Strings and other built-in types get much of their power from the way they support operations like these, but it doesn't stop there.

We can give our own classes the same expressive power by implementing the right *dunder methods*.

You've seen these

```
class StorageJar:

    def __init__(self, jar_type):
        self.jar_type = jar_type


jar = StorageJar("2 oz")
```

Warnings

- You can create your own methods or attributes with double-underscored names. Don't do it.
- You can use an "official" dunder method name but write the function so that its signature doesn't conform. Don't do it.

Don't be reason we can't have nice things.

The plan

We're going to look at some common Python features and see how our classes can work with them by implementing the necessary dunder methods

- String output
- Operator overloading
- Iterators
- Sequences
- Context managers

After we look at each one, we'll take a few minutes to do some hands-on coding examples before proceeding to the next.

String output

Representing an object as a string

```
>>> class Parrot:
...   def __init__(self, breed, alive):
...     self.breed = breed
...     self.alive = alive
...
>>> p = Parrot("Norwegian Blue", True)
>>> print(p)
<__main__.Parrot object at 0x1012a8490>
```

`p` is of type `Parrot`, but the `print()` function needs a string. So what's happening?

Obtaining a string from an object

When we invoke `print(p)`

1. The interpreter checks whether the `Parrot` class implements `__str__()`. It doesn't.
2. The interpreter falls back to using the `__repr__()` method. `Parrot` doesn't override it, so its parent implementation is used.

With dunders

```
>>> class Parrot:
...   def __init__(self, type, alive):
...     self.breed = breed
...     self.alive = alive
...   def __str__(self):
...     if self.alive:
...       return f'{self.breed} parrot'
...     return 'This is an ex-parrot!'
...   def __repr__(self):
...     return f"Parrot('{self.breed}', {self.alive})"
...
```

With dunders

```
>>> p = Parrot('Norwegian Blue', True)
>>> print(p)
Norwegian Blue parrot
>>> str(p)
'Norwegian Blue parrot'
>>> repr(p)
"Parrot('Norwegian Blue', True)"
```

Guidelines

1. `__str__()` should return a user-friendly string.
2. `__repr__()` should return a developer-friendly string. When practical, a string representing the call to the constructor that produces the object in preferred.

Practice

See the `strings` directory in the tutorial repo.

Operator overloading

Recall what we know about strings

```
>>> s1 = "Spam"
>>> s2 = "Cheese"
```

We can compare them

```
>>> s1 == s2
False
>>> s1 < s2
False
```

and even do some arithmetic.

```
>>> s1 + s2
'SpamCheese'
>>> 3 * s1
'SpamSpamSpam'
```

Like many languages, Python supports a range operators that we typically associate with numbers. Strings are not numbers, but for *some* operators it's easy enough to imagine analogous operations so Python overloads the operators.

If you can imagine analogous operations for your own classes, you can implement operator overloading by defining the right dunder methods.

Feeling a bit peckish

```
>>> class Cheese:
...   def __init__(self, name, runniness):
...     self.name = name
...     self.runniness = runniness
...     self.available = False
```

Cheese is great, and we should be able to sort out what == means for cheeses.

Cheesy equality

```
...   def __eq__(self, other):
...     if isinstance(other, Cheese):
...       return (self.name == other.name
...         and self.runniness == other.runniness)
...     return NotImplemented
...
```

Two things to notice:

- We check the type of `other` to be sure we can handle the comparison.
- Rather than raising an exception, we return the special object `NotImplemented` if we can't make a comparison with other.

Comparing

```
>>> class Fruit:
...   def __init__(self, name):
...     self.name = name
...
>>> edam = Cheese('Edam', 1)
>>> cheddar = Cheese('Cheddar', 1)
>>> apple = Fruit('Apple')
>>> edam == cheddar
False
>>> edam == apple
False
>>> edam == edam
True
```

Notice that no exception is raised, even when comparing fruit. This may or may not be a good thing.

Comparison operators and their dunders

==	`__eq__()`
!=	`__ne__()`
>	`__gt__()`
>=	`__ge__()`
<	`__lt__()`
<=	`__le__()`

Some notes:

- You don't have to override any or all of these if they don't make sense for your classes.
- If you override ==, you get != "for free", but best practise is to explicitly override !=.
- If you want to override all of these, see `functools.total_ordering`.

Arithmetic

Suppose (bear with me) we wanted to add various Cheeses together to make some sort of Frankencheese? Again, we implement the right dunder method.

```
...   def __add__(self, other):
...     if isinstance(other, Cheese):
...       newname = self.name + '/' + other.name
...       newrunny = (self.runniness + other.runniness) // 2
...       return Cheese(newname, newrunny)
...     return NotImplemented
```

More arithmetic

If we can add two `Cheeses`, then it also makes sense that we could multiply a `Cheese` and an integer. For this slightly more complicated case we would need to

- Implement `__mul__()`, for expressions like `colby * 3`.
- Implement `__rmul__()`, for expressions like `3 * colby`.
- We might also want `__imul__()`, to support expressions like `colby *= 3`.

The actual implementations are left as an exercise (really).

See `https://docs.python.org/3.6/reference/datamodel.html?highlight=datamodel#emulating-numeric-types`

Practice

See the `operators` directory in the tutorial repo.

Iterators

Again, recall strings

```
>>> s = "Spanish Inquisition"
>>> for c in s:
...   print(c)
...
S
p
a
...
```

We say that Python strings are *Iterables*, which means that they can supply *Iterators* that let us access their elements one at a time.

Iterables/Iterators

- Iterables implement `__iter__()`, which returns an Iterator
- Iterators implement `__next__()`, which returns successive elements from the iterable collection. When the elements have been exhausted, `__next__()` raises a `StopIteration` exception.

To be clear, these are two distinct but related objects[1].

[1] Usually

Example: I'll have your spam

```
>>> class Breakfast:
...   def __init__(self, items):
...     self.items = items
...
>>> my_items = ['spam', 'spam', 'spam', 'spam', 'spam',
... 'spam', 'spam', 'baked beans', 'spam', 'spam', 'spam']
>>> brekky = Breakfast(my_items)
```

It would be nice to be able to iterate over all that spam.

First approach

```
...   def __iter__(self):
...     return iter(self.items)
...
```

`self.items` is a list, which is iterable. We just ask for its iterator.

Second approach

```
...   def __iter__(self):
...     for item in self.items:
...       yield item
...
```

This generator function returns a type of iterator

Third approach

```
...   def __iter__(self):
...     return BreakfastIterator(self)
...
```

For this we need to supply a `BreakfastIterator` class that implements `__next__()`

End result

Whichever approach we use, we can now do this with our `Breakfast`.

```
>>> for food in brekky:
...   if food == 'spam':
...     eat(food)
```

Practice

See the `iterators` directory in the tutorial repo.

Sequences

Hey, let's consider strings one more time

```
>>> s = "And now for something completely different"
>>> s[4]
'n'
>>> s[-1]
't'
>>> s[3:10:2]
' o o'
```

We can access a string's elements with an integer index or a slice because strings are *sequences*.

More spam

```
>>> class Breakfast:
...   def __init__(self, items):
...     self.items = items
...
>>> my_items = ['spam', 'spam', 'spam', 'spam', 'spam',
... 'spam', 'spam', 'baked beans', 'spam', 'spam', 'spam']
>>> brekky = Breakfast(my_items)
```

It makes some sense to be able to treat our `Breakfast` objects as sequences.

First approach

```
...   def __getitem__(self, key):
...     return self.items[key]
...
...   def __len__(self):
...     return len(self.items)
```

This works pretty well since `self.items` is a list, which is also a sequence, so we just delegate the work to the list. I do, however, have an issue with this.

Second approach

```
...   def __getitem__(self, key):
...     if isinstance(key, slice):
...       return Breakfast(self.items[key])
...     return self.items[key]
...
```

When we take a *slice* of a `Breakfast`, the result should be also be a `Breakfast`.

Mutable vs. immutable sequences

If our sequence is meant to be immutable, then we need only implement `__getitem__()` and `__len__()`.

For mutable sequences we need to implement some other dunders. `Breakfast` is pretty mutable.

Methods for mutability

```
...   def __setitem__(self, key, val):
...     self.items[key] = val
...
...   def __delitem__(self, key):
...     del self.items[key]
...
...   def insert(self, key, val):
...     self.items.insert(key, val)
```

See `https://docs.python.org/3/library/collections.abc.html`

The result

```
>>> my_items = ['spam', 'spam', 'spam', 'spam', 'spam',
... 'spam', 'spam', 'baked beans', 'spam', 'spam', 'spam']
>>> brekky = Breakfast(my_items)
>>> brekky[2]
'spam'
>>> del brekky[7]
```

Practice

See the `sequences` directory in the tutorial repo.

Context managers

Oh, an example that's not a string.

```
>>> with open("life_of_brian", "r") as script:
...   for line in script:
...     print(line)
...
```

This is an elegant way to read a file. The thing that makes it work is a *Context Manager*, which is supplied by the call to `open()`.

Required dunders

For a object to serve as a context manager, it needs to implement two dunders

`__enter__()`, which prepares resources to be used in the code block defined by the `with` statement, and which is executed as the block is entered.

`__exit__()`, which cleans up resources, and is executed when the code block is exited.

Example context manager

```
>>> class Announcer:
...   def __enter__(self):
...     print("It's...")
...
...   def __exit__(self, ex_type, ex_val, ex_trace):
...     print("And now for something completely different.")
...
```

Example context manager

```
>>> with Announcer():
...   print("Monty Python's Flying Circus")
...
It's...
Monty Python's Flying Circus
And now for something completely different.
```

Practice

See the `contexts` directory in the tutorial repo.

Thank you!