String representations of class instances: __str__ and __repr__ methods

Python 3.x 3.0–3.6 , Python 2.x 2.7

Examples

Motivation

So you've just created your first class in Python, a neat little class that encapsulates a playing card:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

Elsewhere in your code, you create a few instances of this class:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)
```

You've even created a list of cards, in order to represent a "hand":

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Now, during debugging, you want to see what your hand looks like, so you do what comes naturally and write:

```
print(my_hand)
```

But what you get back is a bunch of gibberish:

```
[<__main__.Card instance at 0x0000000002533788>,
 <__main__.Card instance at 0x00000000025B95C8>,
 <__main__.Card instance at 0x00000000025FF508>]
```

Confused, you try just printing a single card:

```
print(ace_of_spades)
```

And again, you get this weird output:

```
<__main__.Card instance at 0x0000000002533788>
```

Have no fear. We're about to fix this.

First, however, it's important to understand what's going on here. When you wrote `print(ace_of_spades)` you told Python you wanted it to print information about the `Card` instance your code is calling `ace_of_spades` . And to be fair, it did.

That output is comprised of two important bits: the `type` of the object and the object's `id` . The second part alone (the hexidecimal number) is enough to uniquely identify the object at the time of the `print` call.[1]

What really went on was that you asked Python to "put into words" the essence of that object and then display it to you. A more explicit version of the same machinery might be:

```
string_of_card = str(ace_of_spades)
print(string_of_card)
```

In the first line, you try to turn your `Card` instance into a string, and in the second you display it.

The Problem

The issue you're encountering arises due to the fact that, while you told Python everything it needed to know about the `Card` class for you to *create* cards, you *didn't* tell it how you wanted `Card` instances to be converted to strings.

And since it didn't know, when you (implicitly) wrote `str(ace_of_spades)` , it gave you what you saw, a generic representation of the `Card` instance.

The Solution (Part 1)

But *we can* tell Python how we want instances of our custom classes to be converted to strings. And the way we do this is with the `__str__` "dunder" (for double-underscore) or "magic" method.

Whenever you tell Python to create a string from a class instance, it will look for a `__str__` method on the class, and call it.

Consider the following, updated version of our `Card` class:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

        card_name = special_names.get(self.pips, str(self.pips))

        return "%s of %s" % (card_name, self.suit)
```

Here, we've now defined the `__str__` method on our `Card` class which, after a simple dictionary lookup for face cards, **returns** a string formatted however we decide.

(Note that "returns" is in bold here, to stress the importance of returning a string, and not simply printing it. Printing it may seem to work, but then you'd have the card printed when you did something like `str(ace_of_spades)` , without even having a print function call in your main program. So to be clear, make sure that `__str__` returns a string.).

The `__str__` method is a method, so the first argument will be `self` and it should neither accept, nor be passed additonal arguments.

Returning to our problem of displaying the card in a more user-friendly manner, if we again run:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

We'll see that our output is much better:

```
Ace of Spades
```

So great, we're done, right?

Well just to cover our bases, let's double check that we've solved the first issue we encountered, printing the list of `Card` instances, the `hand` .

So we re-check the following code:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

And, to our surprise, we get those funny hex codes again:

```
[<__main__.Card instance at 0x00000000026F95C8>,
 <__main__.Card instance at 0x000000000273F4C8>,
 <__main__.Card instance at 0x0000000002732E08>]
```

What's going on? We told Python how we wanted our `Card` instances to be displayed, why did it apparently seem to forget?

The Solution (Part 2)

Well, the behind-the-scenes machinery is a bit different when Python wants to get the string representation of items in a list. It turns out, Python doesn't care about `__str__` for this purpose.

Instead, it looks for a different method, `__repr__` , and if *that's* not found, it falls back on the "hexidecimal thing".[2]

So you're saying I have to make two methods to do the same thing? One for when I want to `print` my card by itself and another when it's in some sort of container?

No, but first let's look at what our class *would* be like if we were to implement both `__str__` and `__repr__` methods:

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (S)" % (card_name, self.suit)

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (R)" % (card_name, self.suit)
```

Here, the implementation of the two methods `__str__` and `__repr__` are exactly the same, except that, to differentiate between the two methods, (S) is added to strings returned by `__str__` and (R) is added to

differentiate between the two methods, (S) is added to strings returned by __str__ and (R) is added to strings returned by __repr__ .

Note that just like our __str__ method, __repr__ accepts no arguments and **returns** a string.

We can see now what method is responsible for each case:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs',  4)
six_of_hearts = Card('Hearts', 6)

my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]

print(my_hand)           # [Ace of Spades (R), 4 of Clubs (R), 6 of Hearts (R)]

print(ace_of_spades)     # Ace of Spades (S)
```

As was covered, the __str__ method was called when we passed our Card instance to print and the __repr__ method was called when we passed *a list of our instances* to print .

At this point it's worth pointing out that just as we can explicitly create a string from a custom class instance using str() as we did earlier, we can also explicitly create a **string representation** of our class with a built-in function called repr() .

For example:

```
str_card = str(four_of_clubs)
print(str_card)                     # 4 of Clubs (S)

repr_card = repr(four_of_clubs)
print(repr_card)                    # 4 of Clubs (R)
```

And additionally, if defined, we *could* call the methods directly (although it seems a bit unclear and unnecessary):

```
print(four_of_clubs.__str__())     # 4 of Clubs (S)

print(four_of_clubs.__repr__())    # 4 of Clubs (R)
```

About those duplicated functions...

Python developers realized, in the case you wanted identical strings to be returned from str() and repr() you might have to functionally-duplicate methods -- something nobody likes.

So instead, there is a mechanism in place to eliminate the need for that. One I snuck you past up to this point. It turns out that if a class implements the __repr__ method *but not* the __str__ method, and you pass an instance of that class to str() (whether implicitly or explicitly), Python will fallback on your __repr__ implementation and use that.

So, to be clear, consider the following version of the Card class:

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)
```

Note this version *only* implements the __repr__ method. Nonetheless, calls to str() result in the user-friendly version:

```
print(six_of_hearts)            # 6 of Hearts  (implicit conversion)
print(str(six_of_hearts))       # 6 of Hearts  (explicit conversion)
```

as do calls to repr() :

```
print([six_of_hearts])          #[6 of Hearts] (implicit conversion)
print(repr(six_of_hearts))      # 6 of Hearts  (explicit conversion)
```

Summary

In order for you to empower your class instances to "show themselves" in user-friendly ways, you'll want to consider implementing at least your class's __repr__ method. If memory serves, during a talk Raymond Hettinger said that ensuring classes implement __repr__ is one of the first things he looks for while doing Python code reviews, and by now it should be clear why. The amount of information you *could* have added to debugging statements, crash reports, or log files with a simple method is overwhelming when compared to the paltry, and often less-than-helpful (type, id) information that is given by default.

If you want *different* representations for when, for example, inside a container, you'll want to implement both __repr__ and __str__ methods. (More on how you might use these two methods differently below).

Both methods implemented, eval-round-trip style __repr__()

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    # Called when instance is converted to a string via str()
    # Examples:
    #   print(card1)
    #   print(str(card1)
    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)

    # Called when instance is converted to a string via repr()
    # Examples:
    #   print([card1, card2, card3])
    #   print(repr(card1))
    def __repr__(self):
        return "Card(%s, %d)" % (self.suit, self.pips)
```

Syntax

Parameters

Remarks

A note about implemeting both methods

When both methods are implemented, it's somewhat common to have a `__str__` method that returns a human-friendly representation (e.g. "Ace of Spaces") and `__repr__` return an `eval` -friendly representation.

In fact, the Python docs for `repr()` note exactly this:

> For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(), otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object.

What that means is that `__str__` might be implemented to return something like "Ace of Spaces" as shown previously, `__repr__` might be implemented to instead return `Card('Spades', 1)`

This string could be passed directly back into `eval` in somewhat of a "round-trip":

```
object -> string -> object
```

An example of an implementation of such a method might be:

```
def __repr__(self):
    return "Card(%s, %d)" % (self.suit, self.pips)
```

Notes

[1] This output is implementation specific. The string displayed is from cpython.

[2] You may have already seen the result of this `str()` / `repr()` divide and not known it. When strings containing special characters such as backslashes are converted to strings via `str()` the backslashes appear as-is (they appear once). When they're converted to strings via `repr()` (for example, as elements of a list being displayed), the backslashes are escaped and thus appear twice.