

# What is the difference between `__str__` and `__repr__` in Python

## Purpose of `__str__` and `__repr__` in Python

Before we dive into the discussion, let's check out the official documentation of Python about these two functions:

`object.__repr__(self)`: called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object.

`object.__str__(self)`: called by the `str()` built-in function and by the print statement to compute the "informal" string representation of an object.

Quote from [Python's Data Model](#)

From the official documentation, we know that both `__repr__` and `__str__` are used to "represent" an object. `__repr__` should be the "official" representation while `__str__` is the "informal" representation.

So, what does Python's default `__repr__` and `__str__` implementation of any object look like?

For example, let's say we have a `int` `x` and a `str` `y` and we want to know the return value of `__repr__` and `__str__` of these two objects:

```
1 >>> x = 1
2 >>> repr(x)
3 '1'
4 >>> str(x)
5 '1'
6 >>> y = 'a string'
7 >>> repr(y)
8 "'a string'"
9 >>> str(y)
10 'a string'
```

While the return of `repr()` and `str()` are identical for `int x`, you should notice the difference between the return values for `str y`. It is important to realize the default implementation of `__repr__` for a `str` object can be called as an argument to `eval` and the return value would be a valid `str` object:

```
1 >>> repr(y)
2 "'a string'"
3 >>> y2 = eval(repr(y))
4 >>> y == y2
5 True
```

While the return value of `__str__` is not even a valid statement that can be executed by `eval`:

```
1 >>> str(y)
2 'a string'
3 >>> eval(str(y))
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   File "<string>", line 1
7     a string
8         ^
9 SyntaxError: unexpected EOF while parsing
```

Therefore, a "formal" representation of an object should be callable by **`eval()`** and return the same object, if possible. If not possible, such as in the case where the object's members are referring itself that leads to infinite circular reference, then `__repr__` should be unambiguous and contain as much information as possible.

```
1 >>> class ClassA(object):
2 ...     def __init__(self, b=None):
3 ...         self.b = b
4 ...
5 ...     def __repr__(self):
6 ...         return '%s(%r)' % (self.__class__, self.b)
7 ...
8 >>>
9 >>> class ClassB(object):
10 ...     def __init__(self, a=None):
11 ...         self.a = a
12 ...
13 ...     def __repr__(self):
14 ...         return '%s(%r)' % (self.__class__, self.a)
15 ...
16 >>> a = ClassA()
17 >>> b = ClassB(a=a)
18 >>> a.b = b
19 >>> repr(b)
20 RuntimeError: maximum recursion depth exceeded while calling a Python
    object
```

Instead of literally following the requirement of `__repr__` for `ClassB` which causes an infinite recursion problem where `a.__repr__` calls `b.__repr__` which calls `a.__repr__` which calls `b.__repr__`, on and on forever, you could define `ClassB.__repr__` in a different way. A way that shows as much information about an object as possible would be just as good as a valid eval-confined `__repr__`.

```
1 >>> class ClassB(object):
2 ...     def __init__(self, a=None):
3 ...         self.a = a
4 ...
5 ...     def __repr__(self):
6 ...         return '%s(a=a)' % (self.__class__)
7 ...
8
9 >>> a = ClassA()
10 >>> b = ClassB(a=a)
11 >>> a.b = b
12 >>> repr(a)
13 "<class '__main__.ClassA'>(<class '__main__.ClassB'>(a=a))"
14 >>> repr(b)
15 "<class '__main__.ClassB'>(a=a)"
```

Since `__repr__` is the official representation for an object, you always want to call `"repr(an_object)"` to get the most comprehensive information about an object. However, sometimes `__str__` is useful as well. Because `__repr__` could be too complicated to inspect if the object in question is complex (imagine an object with a dozen attributes), `__str__` is helpful to serve as a quick overview of complicated objects. For example, suppose you want to inspect a `datetime` object in the middle of a lengthy log file to find out why the `datetime` of a user's photo is not correct:

```
1 >>> from datetime import datetime
2 >>> now = datetime.now()
3 >>> repr(now)
4 'datetime.datetime(2013, 2, 5, 4, 43, 11, 673075)'
5 >>> str(now)
6 '2013-02-05 04:43:11.673075'
```

The `__str__` representation of `now` looks cleaner and easier to read than the formal representation generated from `__repr__`. Sometimes, being able to quickly grasp what's stored in an object is valuable to grab the "big" picture of a complex program.

## Gotchas between `__str__` and `__repr__` in Python

One important catch to keep in mind is that container's `__str__` uses contained objects' `__repr__`.

```
1 >>> from datetime import datetime
2 >>> from decimal import Decimal
3 >>> print((Decimal('42'), datetime.now()))
4 (Decimal('42'), datetime.datetime(2013, 2, 5, 4, 53, 32, 646185))
5 >>> str((Decimal('42'), datetime.now()))
6 "(Decimal('42'), datetime.datetime(2013, 2, 5, 4, 57, 2, 459596))"
```

Since Python favours unambiguity over readability, the `__str__` call of a tuple calls the contained objects' `__repr__`, the "formal" representation of an object. Although the formal representation is harder to read than an informal one, it is unambiguous and more robust against bugs.

## Tips and Suggestions between `__str__` and `__repr__` in Python

- Implement `__repr__` for every class you implement. There should be no excuse.
- Implement `__str__` for classes which you think readability is more important of non-ambiguity.