


More on Python Double Underline Functions in Python OOP

`__str__()`, `__repr__()`,
`__eq__()`, `__hash__()`

__str__()

When we print an instance of a class, the following result is obviously not desired.

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4 a = A(5)
5 print(a) # prints <__main__.A object at 0x102
```




The screenshot shows a Python REPL window titled '*REPL* [python]'. The output of the command `print(a)` is displayed as `<__main__.A object at 0x000000000219C2B0>`. The prompt `>>>` is visible on the next line.

- When Python meets `print(a)` in the code, it calls `str(a)`, which calls `a.__str__()`
- The `__str__()` method that Python originally has returns that dirty result.
- Therefore we should redefine the `__str__()` method in the particular class (as how we want the instance to be shown, or represented)

__str__()

Define the `__str__()` method (should return a string that you want to see when you print an instance of the class)

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4     def __str__(self):
5         return "A(x=%d)" % self.x
6 a = A(5)
7 print(a) # prints A(x=5) (better)
8
9
```



The screenshot shows a Python REPL window with a tab labeled '*REPL* [python]'. The output of the code is 'A(x=5)', which is displayed on the first line of the REPL. The prompt '>>>' is visible on the second line.

-Much better!

-How about `print([a])`? We want Python to show `[A(x=5)]`, right?

The problem:

```
class A(object):
    def __init__(self, x):
        self.x = x
a1 = A(5)
a2 = A(5)
print(a1 == a2)  # False!
```

The partial solution: `__eq__`

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return (self.x == other.x)
a1 = A(5)
a2 = A(5)
print(a1 == a2)  # True
print(a1 == 99)  # crash (darn!)
```

A better solution:

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return (isinstance(other, A) and (self.x == other.x))
a1 = A(5)
a2 = A(5)
print(a1 == a2)  # True
print(a1 == 99)  # False (huzzah!)
```

__repr__()

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4     def __str__(self):
5         return "A(x=%d)" % self.x
6 a = A(5)
7 print(a) # prints A(x=5) (better)
8 print([a]) # prints [<__main__.A object at 0x102136278>] (yuck!)
```

REPL [python] x

```
A(x=5)
[<__main__.A object at 0x000000000218C358>]
>>> |
```

print(a) 는 object를 단순히 string으로 print하는 경우이므로 __str__()이 call되지만

print([a])는 먼저 [a]를 python이 evaluation을 해야 하고,

python이 evaluation하기 위해서는 __repr__()를 call해야 함

`__repr__()`

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4     def __repr__(self):
5         return "A(x=%d)" % self.x
6 a = A(5)
7 print(a) # prints A(x=5) (better)
8 print([a]) # [A(x=5)]
```

REPL [python] ×

```
A(x=5)
[A(x=5)]
>>>
```

-As shown here, the code works as desired!

Actually, `__repr__()` should return a string in computer-readable form so that `(eval(repr(obj)) == obj)`

- `__repr__()`이 구현되어 있으면 `__str__()`는 필요하지 않다

The problem:

```
class A(object):
    def __init__(self, x):
        self.x = x

a = A(5)
print(a) # prints <__main__.A object at 0x102916128> (yuck!)
```

The partial solution: __str__

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return "A(x=%d)" % self.x

a = A(5)
print(a) # prints A(x=5) (better)
print([a]) # prints [<__main__.A object at 0x102136278>] (yuck!)
```

The better solution: __repr__

*# Note: repr should be a computer-readable form so that
(eval(repr(obj)) == obj), but we are not using it that way.
So this is a simplified use of repr.*

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __repr__(self):
        return "A(x=%d)" % self.x

a = A(5)
print(a) # prints A(x=5) (better)
print([a]) # [A(x=5)]
```

`__eq__()`

- Let's create two instances of a class, where the two instances have the same instance variables.
- Then check the equality of those two. (They should be equal, right?) But....

```
1 ▼ class A(object):  
2     def __init__(self, x):  
3         self.x = x  
4     a1 = A(5)  
5     a2 = A(5)  
6     print(a1 == a2)
```

REPL [python] ×


False
>>>

-WHAT??!!!!

`__eq__()`

-In fact....

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4
5 print(A(5) == A(5))
```



The screenshot shows a Python REPL window titled '*REPL* [python]'. The output of the code is 'False', followed by a prompt '>>>> |'.

-WHAT??!!!

`__eq__()`

- When Python meets `==` in the code, it calls the `__eq__()` method
- We should redefine and tell Python how equality testing should work.

```
1▼ class A(object):
2    def __init__(self, x):
3        self.x = x
4    def __eq__(self, other):
5        return (self.x == other.x)
6
7    a1 = A(5)
8    a2 = A(5)
9    print(a1 == a2)  # True
10
```

REPL [python] ×

True
>>>

But there is one more problem....

`__eq__()`

- When we say `print(a1 == 99)`, we want Python to show `False`, right?
- BUT the code crashes!

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4     def __eq__(self, other):
5         return (self.x == other.x)
6 a1 = A(5)
7 a2 = A(5)
8 print(a1 == 99)
```

```
*REPL* [python] x
Traceback (most recent call last):
  File "C:\Users\Administrator\Desktop\sketchbook.py", line 8, in <module>
    print(a1 == 99)
  File "C:\Users\Administrator\Desktop\sketchbook.py", line 5, in __eq__
    return (self.x == other.x)
AttributeError: 'int' object has no attribute 'x'
>>>
```

Why?

Take a look at our `__eq__()` method we defined.

In this case, `99` is the `other`. But what is `99.x`? No such thing!!

-We should first check if the `other` variable is really a class. More specifically, whether it's an instance of that class in the first place.

`__eq__()`

- We can check whether it's an instance of that class by using the `isinstance` method, which is Python's built-in method.
- In general, `isinstance(a, A)` returns `True` if `a` is an instance of the class `A` (or any subclass of `A`), and `False` otherwise.

```
1▼ class A(object):  
2    def __init__(self, x):  
3        self.x = x  
4    def __eq__(self, other):  
5        return isinstance(other, A) and (self.x == other.x)  
6    a1 = A(5)  
7    a2 = A(5)  
8    print(a1 == a2)    # True  
9    print(a1 == 99)    # False (yay!)
```

REPL [python] x

```
True  
False  
>>>
```

- There we go!

__hash__()

-Let's add an instance of the class into a set and then check if that instance is in the set.

We just added the instance into the set, so we expect that Python shows **True**, right? But...

```
1 class A(object):
2     def __init__(self, x):
3         self.x = x
4
5 s = set()
6 s.add(A(5))
7 print(A(5) in s)
```

Python의 built-in data type
으로의 set에 있는 in 연산자

REPL [python] ×

False
>>>

- Python shows **False**!

`__hash__()`

-Remember that when adding an element or searching for an element in a set, Python first hashes the element to decide which position in the set to locate that element. In that process, Python calls the `__hash__()` method, which we should redefine.

-When defining `__hash__()`, we should always define the `__eq__()` method as well since the `in` method actually compares our target element with all the elements in the set, one by one, to check for equality.

-When defining `__hash__()`, we simply use the Python's built-in hash function. Moreover, if an instance of the class has multiple (unique) instance variables, it's better to hash all of them (to get a more identical distribution). So we hash the tuple that contains all the hashable variables. (We must use a tuple, not a list, since tuple is immutable and list is mutable).

`__hash__()`

```
1 class A(object):
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5     def __hash__(self):
6         # hash a tuple that contains all the hashable things
7         return hash((self.x, self.y))
8     def __eq__(self, other):
9         return (isinstance(other, A) and (self.x == other.x))
10
11 s = set()
12 s.add(A(5, 7))
13 print(A(5, 7) in s)
```

REPL [python] x

True

>>>

There we go!

Using in Set

The problem:

```
class A(object):
    def __init__(self, x):
        self.x = x

s = set()
s.add(A(5))
print(A(5) in s) # False
```

The solution: __hash__ and __eq__

```
class A(object):
    def __init__(self, x):
        self.x = x
    def __hash__(self):
        return hash(self.x)
    def __eq__(self, other):
        return (isinstance(other, A) and (self.x == other.x))

s = set()
s.add(A(5))
print(A(5) in s) # True (whew!)
```

A better (more generalizable) solution

*# Your getHashables method should return the values upon which
your hash method depends, that is, the values that your __eq__
method requires to test for equality.*

```
class A(object):
    def __init__(self, x):
        self.x = x
    def getHashables(self):
        return (self.x, ) # return a tuple of hashables
    def __hash__(self):
        return hash(self.getHashables())
    def __eq__(self, other):
        return (isinstance(other, A) and (self.x == other.x))

s = set()
s.add(A(5))
print(A(5) in s) # True (still works!)
```


Using in Dictionary

The problem (same as sets):

```
class A(object):
    def __init__(self, x):
        self.x = x

d = dict()
d[A(5)] = 42
print(d[A(5)]) # crashes
```

The solution (same as sets):

```
class A(object):
    def __init__(self, x):
        self.x = x
    def getHashables(self):
        return (self.x, ) # return a tuple of hashables
    def __hash__(self):
        return hash(self.getHashables())
    def __eq__(self, other):
        return (isinstance(other, A) and (self.x == other.x))

d = dict()
d[A(5)] = 42
print(d[A(5)]) # works!
```