Operator overloading: comparison

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh
Content Quality Analyst @ DataCamp

Object equality

```
class Customer:
  def __init__(self, name, balance):
    self.name, self.balance = name, balance



customer1 = Customer("Maryam Azar", 3000)
customer2 = Customer("Maryam Azar", 3000)
customer1 == customer2
```

```
False
```

Object equality

```
class Customer:
  def __init__(self, name, balance, id):
    self.name, self.balance = name, balance
    self.id = id

customer1 = Customer("Maryam Azar", 3000, 123)
customer2 = Customer("Maryam Azar", 3000, 123)
customer1 == customer2
```

```
False
```

Variables are references

```
customer1 = Customer("Maryam Azar", 3000, 123)
customer2 = Customer("Maryam Azar", 3000, 123)
```

```
print(customer1)
```

```
<__main__.Customer at 0x1f8598e2e48>
```

```
print(customer2)
```

```
<__main__.Customer at 0x1f8598e2240>
```

Custom comparison

```
import numpy as np


# Two different arrays containing the same data
array1 = np.array([1,2,3])
array2 = np.array([1,2,3])


array1 == array2
```

```
True
```

Overloading __eq__()

```
class Customer:
  def __init__(self, id, name):
    self.id, self.name = id, name
  # Will be called when == is used
  def __eq__(self, other):
    # Diagnostic printout
    print("__eq__() is called")

    # Returns True if all attributes match
    return (self.id == other.id) and \
           (self.name == other.name)
```

- `__eq__()` is called when 2 objects of a class are compared using `==`
- accepts 2 arguments, `self` and `other` - objects to compare
- returns a Boolean

Comparison of objects

```
# Two equal objects

customer1 = Customer(123, "Maryam Azar")
customer2 = Customer(123, "Maryam Azar")

customer1 == customer2
```

```
__eq__() is called
True
```

```
# Two unequal objects - different ids

customer1 = Customer(123, "Maryam Azar")
customer2 = Customer(456, "Maryam Azar")

customer1 == customer2
```

```
__eq__() is called
False
```

Other comparison operators

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

- `__hash__()` to use objects as dictionary keys and in sets

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Operator overloading: string representation

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh
Content Quality Analyst @ DataCamp

Printing an object

```
class Customer:
  def __init__(self, name, balance):
    self.name, self.balance = name, balance

cust = Customer("Maryam Azar", 3000)
print(cust)
```

```
<__main__.Customer at 0x1f8598e2240>
```

```
import numpy as np

arr = np.array([1,2,3])
print(arr)
```

```
[1 2 3]
```

`__str__()`

- `print(obj)`, `str(obj)`

```
print(np.array([1,2,3]))
```

```
[1 2 3]
```

```
str(np.array([1,2,3]))
```

```
[1 2 3]
```

- informal, for end user
- ***str***ing representation

`__repr__()`

- `repr(obj)`, printing in console

```
repr(np.array([1,2,3]))
```

```
array([1,2,3])
```

```
np.array([1,2,3])
```

```
array([1,2,3])
```

- formal, for developer
- ***repr***oducible ***repr***esentation
- fallback for `print()`

Implementation: str

```
class Customer:
  def __init__(self, name, balance):
    self.name, self.balance = name, balance

  def __str__(self):
    cust_str = """
    Customer:
      name: {name}
      balance: {balance}
    """.format(name = self.name, \
               balance = self.balance)
    return cust_str
```

```
cust = Customer("Maryam Azar", 3000)

# Will implicitly call __str__()
print(cust)
```

```
Customer:
  name: Maryam Azar
  balance: 3000
```

Implementation: repr

```
class Customer:
  def __init__(self, name, balance):
    self.name, self.balance = name, balance

  def __repr__(self):
    # Notice the '...' around name
    return "Customer('{name}', {balance})".format(name = self.name, balance = self.balance)
cust = Customer("Maryam Azar", 3000)
cust # <--- # Will implicitly call __repr__()
```

```
Customer('Maryam Azar', 3000) # <--- not Customer(Maryam Azar, 3000)
```

- Surround string arguments with quotation marks in the `__repr__()` output

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Exceptions

OBJECT-ORIENTED PROGRAMMING IN PYTHON

Alex Yarosh
Content Quality Analyst @ DataCamp

```
a = 1
a / 0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

```
a = [1,2,3]
a[5]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    a[5]
IndexError: list index out of range
```

```
a = 1
a + "Hello"
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    a + "Hello"
TypeError: unsupported operand type(s) for +: /
'int' and 'str'
```

```
a = 1
a + b
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    a + b
NameError: name 'b' is not defined
```

Exception handling

- Prevent the program from terminating when an exception is raised
- `try` - `except` - `finally`:

```
try:
  # Try running some code
except ExceptionNameHere:
  # Run this code if ExceptionNameHere happens
except AnotherExceptionHere:      #<-- multiple except blocks
  # Run this code if AnotherExceptionHere happens
...
finally:                          #<-- optional
  # Run this code no matter what
```

Raising exceptions

- `raise ExceptionNameHere('Error message here')`

```
def make_list_of_ones(length):
    if length <= 0:
       raise ValueError("Invalid length!")  # <--- Will stop the program and raise an error
    return [1]*length
```

```
make_list_of_ones(-1)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    make_list_of_ones(-1)
  File "<stdin>", line 3, in make_list_of_ones
    raise ValueError("Invalid length!")
ValueError: Invalid length!
```

Exceptions are classes

- standard exceptions are inherited from `BaseException` or `Exception`

```
BaseException
+-- Exception
    +-- ArithmeticError                    # <---
    |    +-- FloatingPointError
    |    +-- OverflowError
    |    +-- ZeroDivisionError             # <---
    +-- TypeError
    +-- ValueError
    |    +-- UnicodeError
    |         +-- UnicodeDecodeError
    |         +-- UnicodeEncodeError
    |         +-- UnicodeTranslateError
    +-- RuntimeError
   ...
+-- SystemExit
...
```

[1] https://docs.python.org/3/library/exceptions.html

Custom exceptions

- Inherit from `Exception` or one of its subclasses
- Usually an empty class

```
class BalanceError(Exception): pass
```

```
class Customer:
    def __init__(self, name, balance):
     if balance < 0 :
        raise BalanceError("Balance has to be non-negative!")
     else:
        self.name, self.balance = name, balance
```

```
cust = Customer("Larry Torres", -100)
```

```
Traceback (most recent call last):
  File "script.py", line 11, in <module>
    cust = Customer("Larry Torres", -100)
  File "script.py", line 6, in __init__
    raise BalanceError("Balance has to be non-negative!")
BalanceError: Balance has to be non-negative!
```

- Exception interrupted the constructor → object not created

```
cust
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    cust
NameError: name 'cust' is not defined
```

Catching custom exceptions

```
try:
  cust = Customer("Larry Torres", -100)
except BalanceError:
  cust = Customer("Larry Torres", 0)
```

Let's practice!

OBJECT-ORIENTED PROGRAMMING IN PYTHON