Operator overloading: comparing objects

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

George Boorman
Curriculum Manager, DataCamp

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)

# Check for equality
customer1 == customer2
```

```
False
```

Object equality

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

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

```
False
```

Variables are references

```
customer_one = Customer("Maryam Azar", 3000, 123)
customer_two = Customer("Maryam Azar", 3000, 123)
print(customer_one)
```

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

```
print(customer_two)
```

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

- `print()` output refers to the memory chunk that the variable is assigned to
- `==` compares references, not data

Custom comparison

```
# Two different lists containing the same data
list_one = [1,2,3]
list_two = [1,2,3]


list_one == list_two
```

```
True
```

The __eq__() method

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

The __eq__() method

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

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

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
```

Checking types

- What if two objects of different classes have the same attributes and values?
  - Python will evaluate them as equal

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

    def __eq__(self, other):
        # Returns True if the objects have the same attributes
        # and are of the same type
        return (self.acc_id == other.acc_id) and (self.name == other.name)\
            and (type(self) == type(other))
```

Other comparison operators

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

- Customize by defining within a class

Let's practice!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Inheritance comparison and string representation

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

George Boorman
Curriculum Manager, DataCamp

datacamp

Comparing objects from different classes

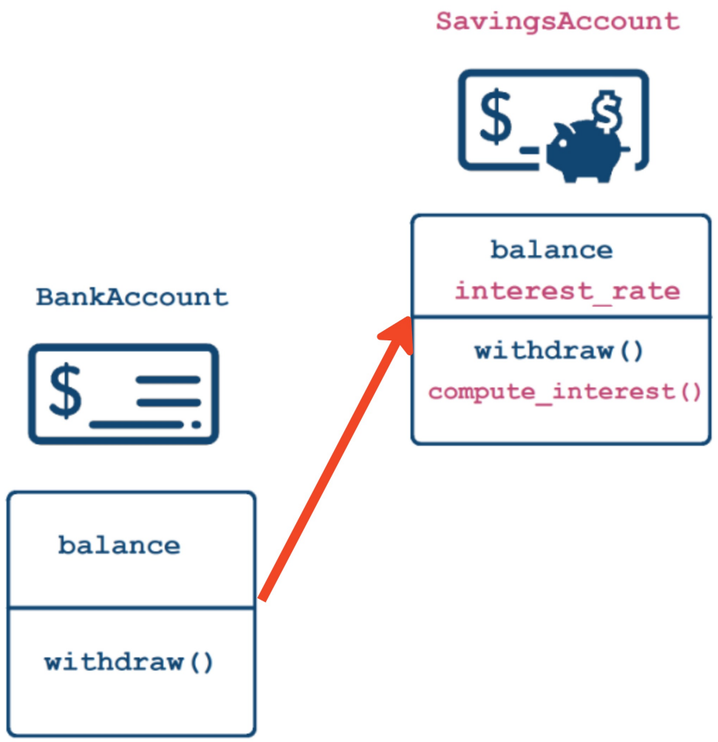
```
class Customer:
    def __init__(self, acc_id, name):
        self.acc_id, self.name = acc_id, name

    def __eq__(self, other):
        # Returns True if the objects have the same attributes
        # and are of the same type
        return (self.acc_id == other.acc_id) and (self.name == other.name)\
            and (type(self) == type(other))
```

Comparing objects with inheritance

What if one object *inherits* from the class of the other object?

Bank and savings accounts



__eq__ in parent/child classes

```
class BankAccount:
    def __init__(self, number, balance=0):
        self.balance = balance
        self.number = number

    def withdraw(self, amount):
        self.balance -= amount

    # Define __eq__ that returns True
    # if the number attributes are equal
    def __eq__(self, other):
        print("BankAccount __eq__() called")
        return self.number == other.number
```

```
class SavingsAccount(BankAccount):
    def __init__(self, number, balance, interes
        BankAccount.__init__(self, number,
                             balance)
          self.interest_rate = interest_rate

    # Define __eq__ that returns True
    # if the number attributes are equal
    def __eq__(self, other):
        print("SavingsAccount __eq__() called")
        return self.number == other.number
```

Comparing parent and child objects

```
ba = BankAccount(123, 10000)
sa = SavingsAccount(456, 2000, 0.05)
# Compare the two objects
ba == sa
```

```
SavingsAccount __eq__() called
False
```

```
sa == ba
```

```
SavingsAccount __eq__() called
False
```

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>
```

```
a_list = [1,2,3]
print(a_list)
```

```
[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: repr

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

    def __repr__(self):
        # Notice the '...' around name
        return f"Customer('{self.name}', {self.balance})"

cust = Customer("Maryam Azar", 3000)
# Will implicitly call __repr__()
cust
```

```
Customer('Maryam Azar', 3000)
```

Implementation: str

```
class Customer:
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance
    def __str__(self):
        cust_str = f"""
        Customer:
            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
```

Let's practice!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Exceptions

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

George Boorman
Curriculum Manager, DataCamp

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:
    print(5 + "a")
except TypeError:
    print("You can't add an integer to a string, but you can multiply them!")
# Can have multiple except blocks
except AnotherExceptionHere:
    # Run this code if AnotherExceptionHere happens
# Optional finally block
finally:
    print(5 * "a")
```

Exception handling output

```
You can't add an integer to a string, but you can multiply them!
aaaaa
```

Raising exceptions

```
def make_list_of_ones(length):
    if length <= 0:
        # Custom message if ValueError occurs
        # Will stop the program and raise the error
       raise ValueError("Invalid length!")
    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

- 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 = name
            self.balance = balance
```

Exception in constructor

```
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!
```

Exceptions terminate the program

- 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!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Congratulations

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

George Boorman
Curriculum Manager, DataCamp

datacamp

Classes and objects

Term	Definition
Class	A **blueprint**/template used to build objects
Object	A combination of *data* and *functionality*; An **instance** of a class

Attributes and methods

Term	Definition
Class	A **blueprint**/template used to build objects
Object	A combination of *data* and *functionality*; An **instance** of a class
State	*Data* associated with an object, assigned through **attributes**
Behavior	An object's *functionality*, defined through **methods**

Core concepts

Encapsulation:

- Bundling of data and methods

Inheritance:

- Extending the functionality of existing code

Polymorphism:

- Creating a unified interface

Comparisons

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

String representation

`__str__()`

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

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

```
[1 2 3]
```

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

```
'[1, 2, 3]'
```

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

`__repr__()`

- `repr(obj)`, printing in console

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

```
[1,2,3]
```

```
[1,2,3]
```

```
[1,2,3]
```

- formal, for developer
- ***repr***oducible ***repr***esentation

Error-handling

```
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
# Use try-except to catch errors
try:
    cust = Customer("Larry Torres", -100)
except BalanceError:
    cust = Customer("Larry Torres", 0)
```

Where to next?

- Multiple inheritance
- Descriptors
- Custom attributes
- Custom iterators
- Type hints
- Abstract base classes

Let's practice!

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON