Object Oriented Programming
Using
Python

Teemu Matilainen
teemu.matilainen@savonia.fi

Lecture 5#

- You realize operator overloading
- You realize what are objects as attributes
- You understand the use of Class methods

Python Special Functions

Class functions that begin with double underscore __ are called special functions in Python.

Function	Description
__init__()	initialize the attributes of the object
__str__()	returns a string representation of the object
__len__()	returns the length of the object
__add__()	adds two objects
__call__()	call objects of the class like a normal function

Python Special Functions

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Reference: https://www.programiz.com/python-programming/operator-overloading

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Python Special Functions…

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Reference: https://www.programiz.com/python-programming/operator-overloading

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)
Equal to	p1 == p2	p1.__eq__(p2)
Not equal top	1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

How to implement the operator

```python
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type
        self.size = 0

    def start(self):
        print(f"The engine starts running with {self.fuel_type} fuel.")

    def stop(self):
        print("The engine stops.")

    def engine_size(self, size: int):
        self.size = size
```

```python
from engine import Engine

class Car:
    def __init__(self, make, model, engine):
        self.make = make
        self.model = model
        self.engine = engine

    def is_the_same_type(self, c1: "Car"):
        return self is c1

    def __eq__(self, value):
        if isinstance(value, Car):
            if  self.make == value.make and \
                self.model == value.model and \
                self.engine == value.engine:  # What about this?????
                return True
            return False
```

How to implement the operator…

```python
from engine import Engine
from car_operator import Car


if __name__ == "__main__":

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("gasoline")

    my_first_car = Car("Mitsubishi", "Outlander", my_first_engine)
    my_second_car = Car("Mitsubishi", "Outlander", my_second_engine)

    print(my_first_car == my_second_car)

    print(my_first_car is my_first_car)
    print(my_first_car is my_second_car)
```

```
False
True
False
```

How to implement the operator…

```python
from engine import Engine
from car_operator import Car


if __name__ == "__main__":

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("gasoline")

    my_first_car = Car("Mitsubishi", "Outlander", my_first_engine)
    my_second_car = Car("Mitsubishi", "Outlander", my_second_engine)

    print(my_first_car == my_second_car)

    my_first_engine.engine_size(size = 2)
    my_second_engine.engine_size(size = 3)

    print(my_first_car == my_second_car)

    print(my_first_engine <  my_second_engine)
    print(my_second_engine <  my_first_engine)

    print(my_first_car is my_first_car)
    print(my_first_car is my_second_car)
```

```
False
False
True
False
```

How to implement the operator…

Let's add __eq__ to Engine class.

```python
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type
        self.size = 0

    def start(self):
        print(f"The engine starts running with {self.fuel_type} fuel.")

    def stop(self):
        print("The engine stops.")

    def engine_size(self, size: int):
        self.size = size

    def __eq__(self, value):
        if isinstance(value, Engine):
            return (self.fuel_type == value.fuel_type)
        return False
```

```
True
True
True
False
```

```python
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type
        self.size = 0

    def start(self):
        print(f"The engine starts running with {self.fuel_type} fuel.")

    def stop(self):
        print("The engine stops.")

    def engine_size(self, size: int):
        self.size = size

    def __eq__(self, value):
        if isinstance(value, Engine):
            return (self.fuel_type == value.fuel_type and self.size == value.size)
        return False
```

```
True
False
True
False
```

One more operator example…

```
    def __lt__(self, object):
        if self.size < object.size:
            return True
        else:
            return False
```

```
from engine import Engine
from car_operator import Car


if __name__ == "__main__":

    my_first_engine = Engine("gasoline")
    my_second_engine = Engine("gasoline")

    my_first_car = Car("Mitsubishi", "Outlander", my_first_engine)
    my_second_car = Car("Mitsubishi", "Outlander", my_second_engine)

    print(my_first_car == my_second_car)

    my_first_engine.engine_size(size = 2)
    my_second_engine.engine_size(size = 3)

    print(my_first_car == my_second_car)

    print(my_first_engine <  my_second_engine)
    print(my_second_engine <  my_first_engine)

    print(my_first_car is my_first_car)
    print(my_first_car is my_second_car)
```

```
True
False
True
False
True
False
```

What is a __call__ operator?

```python
class Engine:
    def __init__(self, fuel_type):
        self.fuel_type = fuel_type
        self.size = 0

    def start(self):
        print(f"The engine starts running with {self.fuel_type} fuel.")

    def stop(self):
        print("The engine stops.")

    def engine_size(self, size: int):
        self.size = size

    def __eq__(self, value):
        if isinstance(value, Engine):
            return (self.fuel_type == value.fuel_type and self.size == value.size)
        return False

    def __lt__(self, object):
        if self.size < object.size:
            return True
        else:
            return False

    def __call__(self, *args: Any, **kwds: Any):
        print(f"Instance of Engine called with args: {args} and kwargs: {kwds}")
```

What is a __call__ operator?

```
print(my_first_engine <  my_second_engine)
print(my_second_engine <  my_first_engine)

print(my_first_car is my_first_car)
print(my_first_car is my_second_car)

# Calling the instance as if it were a function
my_first_engine(1, 2, keyword_arg='example')
my_first_engine(1, 2, 3, 4, arg='example')
```

```
False
True
False
True
False
Instance of Engine called with args: (1, 2) and kwargs: {'keyword_arg': 'example'}
Instance of Engine called with args: (1, 2, 3, 4) and kwargs: {'arg': 'example'}
```

This can be useful in situations where you want instances of your class to be callable, providing a more function-like behavior for objects.

Benefits of Operator Overloading:

Operator overloading offers several advantages, including:

Enhanced Code Readability:

It enhances code readability by enabling the utilization of familiar operators, contributing to a more intuitive understanding of the code.

Consistent Object Behavior:

Ensures that instances of a class behave consistently with both built-in types and other user-defined types, promoting a uniform and predictable experience.

Simplified Code Writing:

Streamlines code writing, particularly for intricate data types, making the codebase more straightforward and easier to comprehend.

Code Reusability:

Facilitates code reuse through the implementation of a single operator method, which can be employed for multiple operators, promoting efficiency in development.

Objects as attributes

We've previously explored instances of classes with lists as attributes. Since there are no restrictions preventing us from incorporating mutable objects as attributes in our classes, we can seamlessly employ instances of our own classes as attributes within other classes we've created. In the upcoming examples, we'll introduce the Book, Member, and BorrowedBook classes. The BorrowedBook class utilizes the first two classes. The class definitions are intentionally brief and straightforward to emphasize the practice of utilizing instances of our own classes as attributes.

Let’s define the class Book in a file named book.py:

```python
class Book:
    def __init__(self, name: str, code: str, price: int):
        self.name = name
        self.code = code
        self.price = price
```

Next, the class Member

```python
class Member:
    def __init__(self, name: str, member_number: str, credits: int):
        self.name = name
        self.member_number = member_number
        self.credits = credits
```

- **Finally, we create the class BorrowedBook. This class uses the other two classes, they have to be imported before they can be used:**

```python
from book import Book
from member import Member

class BorrowedBook:
    def __init__(self, member: Member, book: Book, price: int):
        self.member = member
        self.book = book
        self.price = price
```

- Here is an example of a main function which will add some borrowed books to a list:

```
from borrowedbook import BorrowedBook
from book import Book
from member import Member

# Create a list of members
members = []
members.append(Member("name 1", "1111", 23))
members.append(Member("name 2", "2222", 56))
members.append(Member("name 3", "3333", 12))
members.append(Member("name 4", "4444", 2))

itp = Book("Introduction to Programming", "qwerty", 50)

read = []
for member in members:
    read.append(BorrowedBook(member, itp, 30))

# Print out the name of the member for each completed course
for book in read:
    print(book.member.name)
```

```
name 1
name 2
name 3
name 4
```

What does these dots mean in a print(book.member.name)?

- book is an instance of the class BorrowedBook
- member refers to an attribute of the BorrowedBook object, which is an object of type Member
- the attribute name in the Member object contains the name of the member

```
itp = Book("Introduction to Programming", "qwerty", 50)

read = []
for member in members:
    read.append(BorrowedBook(member, itp, 30))

# Print out the name of the member for each completed course
for book in read:
    print(book.member.name)
```

When do we need import statement?

```
from borrowedbook import BorrowedBook
from book import Book
from member import Member
```

An import statement is only necessary when using code which is defined outside the current file (or Python interpreter session). This includes situations where we want to use something defined in the **Python standard library**.

A list of objects as an attribute of an object???

Now we add new methods to a Book class and then we will use the created Author class

Create an Author class

```python
class Author:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age
```

```python
from author import Author

class Book:
    def __init__(self, name: str, code: str, price: int):
        self.name = name
        self.code = code
        self.price = price
        self.authors = []

    def add_author(self, author: Author):
        self.authors.append(author)

    def summary(self):
        names = []
        for author in self.authors:
            names.append(author.name)

        print("Book:", self.name)
        print("Authors:", len(self.authors))
        print("Authors names are from each book:", names)
```

An example of our class in action:

```python
from book import Book
from author import Author


if __name__ == "__main__":

    b = Book("The Catcher in the Rye","bbat", 10)
    b.add_author(Author("J. D. Salinger", 45))
    b.add_author(Author("Imaginary writer II", 22))
    b.add_author(Author("Imaginary writer III", 10))
    b.summary()
```

```
Book: The Catcher in the Rye
Authors: 3
Authors names are from each book: ['J. D. Salinger', 'Imaginary writer II', 'Imaginary writer III']
```

```python
from book import Book

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book: Book):
        self.books.append(book)
```

Using nested loops and nested object lists…

```python
from book import Book
from author import Author
from library import Library


if __name__ == "__main__":

    b = Book("The Catcher in the Rye","bbat", 10)
    b.add_author(Author("J. D. Salinger", 45))
    b.add_author(Author("Imaginary writer II", 22))
    b.add_author(Author("Imaginary writer III", 10))
    b.summary()

    b2 = Book("Shining","ccas", 10)
    b2.add_author(Author("John Doe", 45))
    b2.add_author(Author("Writer 2", 22))
    b2.add_author(Author("Writer 3", 10))
    b2.summary()

    l = Library()
    l.add_book(b)
    l.add_book(b2)

    for book in l.books:
        print("Name of the book: " + book.name)
        for author in book.authors:
            print("Author: " + author.name)
```

```
Book: The Catcher in the Rye
Authors: 3
Authors names are from each book: ['J. D. Salinger', 'Imaginary writer II', 'Imaginary writer III']
Book: Shining
Authors: 3
Authors names are from each book: ['John Doe', 'Writer 2', 'Writer 3']
Name of the book: The Catcher in the Rye
Author: J. D. Salinger
Author: Imaginary writer II
Author: Imaginary writer III
Name of the book: Shining
Author: John Doe
Author: Writer 2
Author: Writer 3
```

What is Class Method?

We know already how to use:

- Instance method
- Static methods
- …and also, Dynamic method

A class method is declared using the @classmethod decorator, where the first parameter is consistently named **cls**. The usage of **cls** is analogous to the self parameter, with the distinction being that **cls** references the class itself, while **self** refers to an instance of the class. Neither parameter needs to be explicitly provided in the argument list when calling the function; Python automatically assigns the appr values.

```python
class Book:
    note = "This is a - book paperback"

    def __init__(self, name: str, code: str, price: int):
        self.name = name
        self.code = code
        self.price = price
        self.authors = []

    def add_author(self, author: Author):
        self.authors.append(author)

    def summary(self):
        names = []
        for author in self.authors:
            names.append(author.name)

        print("Book:", self.name)
        print("Authors:", len(self.authors))
        print("Authors names are from each book:", names)

    @classmethod
    def price(cls, name, code, price):
        return cls(name, code, price)

    @classmethod
    def hidden_information(cls):
        return cls.note
```

```python
new_book = Book.price("The Catcher in the Rye", "Plaa plaa plaa", 123)
print(new_book.name)
print(new_book.hidden_information())
```

Class Method continues?

Copying an object…

```python
    @classmethod
    def price(cls, name, code, price):
        return cls(name, code, price)

    @classmethod
    def hidden_information(cls):
        return cls.note

    @classmethod
    def copy_book(cls, old_book: "Book"):
        return cls(old_book.name, old_book.code, old_book.price)
```

```python
new_book = Book.price("The Catcher in the Rye", "Plaa plaa plaa", 123)
print(new_book.name)
print(new_book.hidden_information())

another_new_book = Book.copy_book(new_book)
print(new_book)
print(another_new_book)
```

```
The Catcher in the Rye
This is a - book paperback
<book.Book object at 0x0000026BA674F320>
<book.Book object at 0x0000026BA674F350>
```