Object Oriented Programming II

Raul P. Pelaez

March 17, 2026

Contents

1 Introduction

In this lesson we build upon the fundamentals of Object Oriented Programming introduced previously. We will explore operator overloading, type casting, and class dunder methods.

2 The Python Data model: integrating our classes with Python

Special methods (often called "dunder" methods) enable customization of how objects interact with Python's built-in functions and operators. For instance, when calling `len(something)` Python will look for a `__len__` method in the object `something`. If it exists, Python will call it to determine the length of the object. When using a dunder method, Python will try to fall back to a default behavior if the method is not implemented. For instance, we can print an object by calling `print(something)`. If `__str__` or `__repr__` are not implemented, Python will fall back to a default representation.

Info

Common Dunder Methods
Key special methods include:

- `__init__`: Initializes new objects.
- `__str__` and `__repr__`: Define string representations.
- `__len__`: Returns the length of an object.
- `__eq__`: Checks for equality between objects.
- `__add__`, `__sub__`, `__mul__`, etc.: Implement arithmetic operations.
- `__getitem__`: Makes the class `iterable` and enables the operator `[]`. Only makes sense for "containers".
- `__setitiem__`: Allows to change the value of an item in a container-like class.
- `__call__`: Makes an instance callable like a function (e.g., `instance()`).

Advice

Never call dunder methods directly
Calling `something.__str__()` relies on an actual method named `__str__` being defined, while `str(something)` will always work, even if `__str__` is not defined.

Example:

```python
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"'{self.title}' by {self.author}"

    def __len__(self):
        return self.pages

b = Book("1984", "George Orwell", 328)
print(b)   # User-friendly representation
print(len(b))   # Retrieves number of pages
```

Output

```
'1984' by George Orwell
328
```

2.1 Operator Overloading

Python allows classes to define their own behavior for built-in operators by overriding special methods.

Info

Defining Operator Overloads
Operator overloading gives your classes the ability to respond to operators such as +, -, *, and others. By implementing methods like `__add__`, you enable instances of your class to be added together in a natural way.

Example:

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        raise TypeError("Operands must be of type Vector")

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
print(v1 + v2)  # Output: Vector(6, 8)
print(v1 == v2)  # Output: False
print(v1 == Vector(2, 3))  # Output: True
```

Output

```
Vector(6, 8)
False
True
```

Advice

Always check the type of the other operand in your operator overloads to ensure safe and predictable behavior.

3 Casting

Casting in Python refers to converting an object from one data type to another. This can be done explicitly using built-in functions.

Info

Types of Casting
Casting may be explicit, using functions like `int()`, `float()`, or `str()`, or implicit, where Python automatically converts types in expressions.

Example:

```python
# Explicit casting example
num_str = "123"
num_int = int(num_str)
print(num_int, type(num_int))

# Implicit casting in arithmetic operations
result = 3 + 4.5  # The integer 3 is cast to float
print(result, type(result))
```

Output

```
123 <class 'int'>
7.5 <class 'float'>
```

We can make our classes castable by implementing the `__int__`, `__float__`, or `__str__` methods.

Example:

```python
class Temperature:
    def __init__(self, celsius):
        self.celsius = float(celsius)

    def __float__(self):
        return self.celsius

    def to_fahrenheit(self):
        return self.celsius * 9/5 + 32

temp = Temperature(25)
print(float(temp))  # Output: 25.0
print(temp.to_fahrenheit())  # Output: 77.0
```

Output

```
25.0
77.0
```

4 Exercises

4.1 The Deck of cards

Goal

Lets create a deck of cards using classes that we can use to code card games. We will focus on the French deck, which consists of 52 cards divided into four suits: hearts, diamonds, clubs, and spades. Each suit contains 13 ranks: Ace, 2-10, Jack, Queen, and King. We will try to make as much use of the dunder methods as possible. For instance, instead of writing a method to draw a random card, we can simply write the `__getitem__` and `__len__` methods to allow indexing the deck and use `random.choice` to draw a card. We will write two classes:

- A class `Card` that represents a single card and has attributes for suit and rank.
- A class `FrenchDeck` that represents a deck of cards and has a list of cards as an attribute.

With these classes, we could write code to simulate a game of poker or blackjack. For instance, we could shuffle the deck by calling `random.shuffle(deck)` and draw a card by calling `deck[0]`.

Milestone

Write the `Card` class with attributes for suit and rank. Implement the `__str__` method to provide a user-friendly representation of the card.

Milestone

Write the `FrenchDeck` class with a list of cards as an attribute. Write its initializer to populate the list with all 52 cards in a French deck. Make sure the list of cards is a private member, naming it with a leading underscore.

hint

- Store all possible ranks and suits as class attributes. You can denote the "figures" as `"JQKA"`.
- Use a list comprehension to generate all the cards in the deck.

Milestone

Write the rest of the dunder methods for the `FrenchDeck`, in particular, you should implement:

- `__len__` to return the number of cards in the deck.
- `__getitem__` to allow indexing the deck.
- `__setitem__` to allow changing the value of a card in the deck.

Write some tests to ensure the deck behaves as expected. And have error handling in mind when implementing them. For instance, is the user trying to set an invalid card in the deck? After this, you should be able to run the following example:

```python
import random
# Your class here
# ...
deck = FrenchDeck()
print(random.choice(deck))  # Draw a random card
print(deck[0])  # Draw the first card
for card in deck:
    print(card)
random.shuffle(deck)
print(deck[0])  # Draw the first card after shuffling
```

Advanced milestone

After shuffling the deck, use the `sorted` Python built-in function to sort the deck by rank and suit. You can use the `key` argument to `sorted` to define a custom sorting function.

Advanced milestone

Implement the BlackJack game using the deck of cards you created. The game should have the following rules:

- The game is played between a player and a dealer.
- The player and dealer are dealt two cards each.
- The player can choose to hit (draw a card) or stand (keep their current hand).
- The dealer must hit if their hand is below 17. Figures count as 10, and aces can be 1 or 11.
- The player wins if their hand is closer to 21 than the dealer's hand without going over.

hint

You can use the `input` function to get user input. For instance, you can ask the player if they want to hit or stand.