07. Object-Oriented Programming

October 13, 2025

0.1 1. Dunder (magic) methods and attributes. Operator overloading

Dunder methods (short for "double underscore methods") and attributes in Python, also known as magic methods or special methods, are a set of predefined methods and attributes that begin and end with double underscores (`__`). These methods and attributes are used to provide special functionality to classes and objects, allowing for the customization of behavior in specific situations. Here's an overview:

0.1.1 Dunder Methods

Dunder methods allow you to define how objects of your classes behave with standard Python operations. Some common dunder methods include:

1. **Initialization and Representation**
   - `__init__(self, ...)`: Initializes a new instance of a class. Called when an object is created.
   - `__repr__(self)`: Returns an official string representation of an object, typically one that can be used to recreate the object.
   - `__str__(self)`: Returns a string representation of an object, used by the `print()` function.
2. **Arithmetic Operations**
   - `__add__(self, other)`: Defines behavior for the `+` operator.
   - `__sub__(self, other)`: Defines behavior for the `-` operator.
   - `__mul__(self, other)`: Defines behavior for the `*` operator.
   - `__truediv__(self, other)`: Defines behavior for the `/` operator.
   - `__floordiv__(self, other)`: Defines behavior for the `//` operator.
   - `__mod__(self, other)`: Defines behavior for the `%` operator.
   - `__pow__(self, other)`: Defines behavior for the `**` operator.
3. **Comparison Operations**
   - `__eq__(self, other)`: Defines behavior for the equality operator `==`.
   - `__ne__(self, other)`: Defines behavior for the inequality operator `!=`.
   - `__lt__(self, other)`: Defines behavior for the less-than operator `<`.
   - `__le__(self, other)`: Defines behavior for the less-than-or-equal-to operator `<=`.
   - `__gt__(self, other)`: Defines behavior for the greater-than operator `>`.
   - `__ge__(self, other)`: Defines behavior for the greater-than-or-equal-to operator `>=`.
4. **Container Methods**
   - `__len__(self)`: Defines behavior for the `len()` function.
   - `__getitem__(self, key)`: Defines behavior for indexing `obj[key]`.

- `__setitem__(self, key, value)`: Defines behavior for assignment to an index `obj[key] = value`.
- `__delitem__(self, key)`: Defines behavior for deleting an indexed value `del obj[key]`.

For more examples on special methods see this tutorial or the official documentation.

0.1.2 Dunder Attributes

Dunder attributes provide specific information about the class or its instances. Some common dunder attributes include:

- `__dict__`: A dictionary or other mapping object used to store an object's (writable) attributes.
- `__class__`: A reference to the class of the instance.
- `__name__`: The name of the class or function.
- `__module__`: The name of the module in which the class was defined.
- `__bases__`: A tuple containing the base classes of a class.

0.1.3 Examples

Here are a few examples to illustrate how dunder methods work:

1. **Custom String Representation:**

[15]:
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"Person(name={self.name}, age={self.age})"

    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

[16]:
```python
p = Person("Anna", 20)
```

[17]:
```python
repr(p)
```

[17]:
```
'Person(name=Anna, age=20)'
```

[18]:
```python
str(p)
```

[18]:
```
'Anna is 20 years old'
```

2. **Custom Addition:**

[19]:
```python
class Vector:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

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

[20]:
```
v1 = Vector(2, 3)
v2 = Vector(5, 7)
v3 = v1 + v2
```

[21]:
```
v3
```

[21]:
```
Vector(7, 10)
```

0.1.4 Exercises 1

1. Create class `Dish` - instance attributes: `dish_id (int)`, `name (str)`, `price (int)`
2. Create class `Menu` - instance attributes: `dishes (list of Dish objects)`. Implement appropriate methods so that `Menu` objects support the following operations:
   - `d = Dish(0, 'Lasagna', 20)`
   - `m = Menu()`
   - `m.append(d)` - dish appended to m.dishes
   - `m[0]` - implement getitem on Menu
   - `d in m` - implement membership test operators
   - `len(m)` - return length of m.dishes
3. Have 2 dishes created with same values for attributes (`d1` and `d2`). Add `d1` to the menu instance `m`. Test membership of `d2` in `m`. Does it find `d2` in menu? Why?
4. Modify `Dish` to test for equality by looking at `dish_id`, `name`, `price` being the same, so that the dishes above would make this true `d1 == d2`. Test `d2 in m` again.
5. Modify the getitem dunder to get the dish with the `dish_id` equal with the argument given. Raise `KeyError` if not found.

0.2 2. Static and class methods

Besides instance methods, in Python we can have two more types of methods:

- class methods
- static methods

Class methods are similar to class variables. They are common to all instances, can be called from both the instance and the class, and have access to other class methods and to class variables.

Static methods, on the other hand, don't have acces to the class or the instance. They are simple functions which make sense only in the context of the class, but otherwise don't use any internal class data. They can also be called from the instance or from the class.

In order to mark a method as either class/static method, we use the respective built-in decorator:

`@classmethod`

@staticmethod

[1]:
```
import math

class Pizza:
    TOPPINGS = ('mozzarella', 'prosciutto', 'tomatoes')

    def __init__(self, radius, toppings):
        for topping in toppings:
            if not self.validate_topping(topping):
                raise ValueError(f'Accepted toppings: {self.TOPPINGS}')
        self.radius = radius
        self.toppings = toppings

    def area(self):
        return self.circle_area(self.radius)

    @classmethod
    def validate_topping(cls, topping):
        if topping in cls.TOPPINGS:
            return True
        return False

    @staticmethod
    def circle_area(r):
        return math.pi * r ** 2


margherita = Pizza(15, ['mozzarella', 'tomatoes'])
print(margherita.area())
```

```
706.8583470577034
```

[23]:
```
new_topping = 'ham'
if Pizza.validate_topping(new_topping):
    margherita.toppings.append(new_topping)
print(margherita.toppings)
```

```
['mozzarella', 'tomatoes']
```

[24]:
```
Pizza.circle_area(17)
```

[24]:
```
907.9202768874502
```

0.2.1 Exercises 2

1. Create a class `BankAccount` with an attribute `balance`. Implement a static method `validate_amount(amount)` to check if an amount is positive and raise an exception otherwise. Create two methods in this class, one to withdraw money and another one to deposit

money into the account. The withdraw method will not allow withdrawing more money than available: it will raise an exception and not change the balance. The two instance methods should use the static method to validate that the amount is positive.

2. Create a class `Temperature` with an instance attribute `celsius`. Add a static method `celsius_to_fahrenheit(celsius)` to convert Celsius to Fahrenheit and a class method `from_fahrenheit(cls, fahrenheit)` to create an instance from a Fahrenheit value. Use the two methods.

   `Temperature in degrees Fahrenheit (°F) = (Temperature in degrees Celsius (°C) * 9/5) + 32.`

0.3 3. Access control solutions

"Private" instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls.

0.4 4. Getter/Setter/Deleter methods. The `property` class

When we need to manage an attribute's getting/setting/deleting, we can do it through a `property`. `property` is a built-in class that can be used as a decorator that can expose a data member to the caller, but manage getting/setting/deleting that attribute through methods. It can be used for:

- data validation (check condition before setting)
- computed attributes
- any other operations we want to make at the same time with attribute getting/setting/deleting.

[25]:
```
class Pizza:
    TOPPINGS = ('mozzarella', 'prosciutto', 'tomatoes')

    def __init__(self, topping):
        self._topping = topping

    @property
    def topping(self):
        print('getter called!')
        return self._topping.capitalize()
```

```
    @topping.setter
    def topping(self, value):
        print('setter called!')
        if value not in self.TOPPINGS:
            raise ValueError
        self._topping = value

    @topping.deleter
    def topping(self):
        print('deleter called!')
        del self._topping


my_pizza = Pizza('mozzarella')
print(my_pizza.topping)
```

```
getter called!
Mozzarella
```

[26]:
```
try:
    my_pizza.topping = 'parmesan'
except ValueError:
    print('Invalid topping')
```

```
setter called!
Invalid topping
```

[27]:
```
del my_pizza.topping
```

```
deleter called!
```

0.4.1 Exercises 4

1. Improve the `BankAccount` from the previous exercise:
   - Make the argument balance of the `__init__` an optional argument with default value 0
   - Override the dunder method `__str__` so that we can print bank objects more easily. Make it tell us the bank name and balance. Make `__repr__` do the same thing.
   - Make `balance` an `_` attribute to suggest it is *protected*. Make `balance` a `property` in order to return it. Update *deposit* and *withdraw* to use *_balance*
2. Create a class `Employee` with three instance attributes:
   - `name`
   - `bank_account` (it should be a `BankAccount` object; pass an already created `BankAccount` instance at `Employee` initialisation)
   - `salary (default 0)`
3. Write a method `raise_salary` that receives a parameter `percent` that should be one of the following values: 5, 10, 20. Raise a `ValueError` if another value is received by this method. The `raise_salary` method should raise the salary with 5%, 10% or 20%.

4. Create a method `receive_salary` that will deposit in the employee's bank account an amount equal to its salary.
5. Use a `property` for salary management. Salary should be set only on initialisation; you shouldn't be able to set the salary from outside the class.
6. Make `bank_account` protected by one `_` and add a property `net_worth` that returns the balance from the employee bank_account. (should be called w/o parentheses)
7. Add a method `spend` that substracts a given amount from the employee's bank_account
8. Create an `Employee` instance and call `raise_salary`, `receive_salary` and `spend` on it. Print its `net_worth` afterwards.

0.5 5. `__new__`

0.6 6. Metaclasses

Metaclasses in Python are advanced constructs that control the behavior and creation of classes. A metaclass is essentially a class of a class; it defines how classes behave and can be used to customize the creation of classes and their instances.

0.6.1 Understanding Metaclasses

1. **Class Definition and Instantiation**:
   - In Python, when you define a class, you are creating an instance of the type `type`.
   - For example:

[38]:
```
class MyClass:
    pass
```

[39]:
```
type(MyClass)
```

[39]: `type`

2. **Role of Metaclasses**:
   - A metaclass defines how a class behaves. A class is an instance of a metaclass.
   - By default, the metaclass for all classes in Python is `type`.
3. **Custom Metaclasses**:
   - You can create custom metaclasses to control the creation and behavior of classes.
   - A metaclass is defined by inheriting from `type`.

0.6.2 Creating and Using a Metaclass

1. **Defining a Metaclass**:
   - To create a metaclass, you define a class that inherits from `type`.
   - Override special methods like `__new__` and `__init__` to customize class creation.

[40]:
```
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, dct)

    def __init__(cls, name, bases, dct):
```

```
        print(f"Initializing class {name}")
        super().__init__(name, bases, dct)

    def __call__(cls, *args, **kwargs):
        print("Metaclass called")
        return super().__call__(*args, **kwargs)
```

2. **Using a Metaclass**:

- Specify the metaclass for a class using the `metaclass` keyword argument.

[41]:
```
class MyClass(metaclass=MyMeta):
    pass
```

```
Creating class MyClass
Initializing class MyClass
```

- When the class is instantiated, the `__call__` method of its metaclass is called:

[37]:
```
MyClass()
```

```
Metaclass called
```

[37]:
```
<__main__.MyClass at 0x1134dba70>
```

0.7 7. Design Patterns

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that can be customized to solve a recurring design problem.

A design pattern is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

0.7.1 Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

Problem A Singleton is a class with two main characteristics:

- It can have at most one instance
- It should be globally accessible in the program

These properties are both important, although in practice you'll often hear people calling something a Singleton even if it has only one of these properties.

Having only one instance is usually a mechanism for controlling access to some shared resource. For example, two threads may work with the same file, so instead of both opening it separately, a Singleton can provide a unique access point to both of them.

Global accessibility is important because after your class has been instantiated once, you'd need to pass that single instance around in order to work with it. It can't be instantiated again. That's why it's easier to make sure that whenever you try to instantiate the class again, you just get the same instance you've already had.

Solution The Singleton class can be implemented in different ways in Python. Some possible methods include: base class, decorator, metaclass. We will use the metaclass to see a practical example of metaclasses:

[1]:
```
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]


class Singleton(metaclass=SingletonMeta):
    pass


class OtherSingleton(metaclass=SingletonMeta):
    pass


sgt = Singleton()
other_sgt = OtherSingleton()


if __name__ == "__main__":
    s1 = Singleton()
    s2 = Singleton()

    o1 = OtherSingleton()
    o2 = OtherSingleton()
    o3 = OtherSingleton()
    o4 = OtherSingleton()

    print("Instances:", SingletonMeta._instances)

    if s1 is sgt:
        print("Singleton works, both variables contain the same instance.")
    else:
        print("Singleton failed, variables contain different instances.")
```

```
Instances: {<class '__main__.Singleton'>: <__main__.Singleton object at
0x1045e0440>, <class '__main__.OtherSingleton'>: <__main__.OtherSingleton object
at 0x1045e0590>}
Singleton works, both variables contain the same instance.
```

0.7.2 Iterator

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Problem Collections are one of the most used data types in programming. Nonetheless, a collection is just a container for a group of objects.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs and other complex data structures.

But no matter how a collection is structured, it must provide some way of accessing its elements so that other code can use these elements. There should be a way to go through each element of the collection without accessing the same elements over and over.

This may sound like an easy job if you have a collection based on a list. You just loop over all of the elements. But how do you sequentially traverse elements of a complex data structure, such as a tree? For example, one day you might be just fine with depth-first traversal of a tree. Yet the next day you might require breadth-first traversal. And the next week, you might need something else, like random access to the tree elements.

Adding more and more traversal algorithms to the collection gradually blurs its primary responsibility, which is efficient data storage. Additionally, some algorithms might be tailored for a specific application, so including them into a generic collection class would be weird.

On the other hand, the client code that's supposed to work with various collections may not even care how they store their elements. However, since collections all provide different ways of accessing their elements, you have no option other than to couple your code to the specific collection classes.

Solution The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an iterator.

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end. Because of this, several iterators can go through the same collection at the same time, independently of each other.

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it doesn't return anything, which means that the iterator has traversed all of the elements.

All iterators must implement the same interface. This makes the client code compatible with any collection type or any traversal algorithm as long as there's a proper iterator. If you need a special way to traverse a collection, you just create a new iterator class, without having to change the collection or the client.

[46]:
```python
from collections.abc import Iterator


class Fibonacci(Iterator):
    """Iterator that yields numbers in the Fibonacci sequence"""

    def __init__(self, max_value):
        self.max = max_value

    def __iter__(self):
        self.a, self.b = 0, 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib


if __name__ == "__main__":
    fib_sequence = Fibonacci(100)
    for i in fib_sequence:
        print(i)
```

```
0
1
1
2
3
5
8
13
21
34
55
89
```

0.7.3 Prototype

Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

All prototype classes should have a common interface that makes it possible to copy objects even if their concrete classes are unknown. Prototype objects can produce full copies since objects of the same class can access each other's private fields.

Problem We need to clone an object, but may not know its exact type, parameters, they may not all be assigned through the constructor itself or may depend on system state at a particular point during the runtime.

If we try to do it directly we'll add a lot of dependencies branching in our code, and it may not even work at the end.

Solution The Prototype design pattern addresses the problem of copying objects by delegating it to the objects themselves. Python offers support for creating copies of objects through the `copy` and `deepcopy` functions from `copy` module. These functions can create copies for most objects without customization:

[8]:
```
import copy
from datetime import date


class Person:
    def __init__(self, name, date_of_birth, hobbies=None):
        self.name = name
        self.__date_of_birth = date_of_birth
        self.hobbies = hobbies

    @property
    def age(self):
        today = date.today()
        age = today.year - self.__date_of_birth.year
        if (today.month, today.day) < (self.__date_of_birth.month, self.
 ↪__date_of_birth.day):
            age -= 1
        return age

    def __str__(self):
        return f"{self.__class__.__name__} object at {id(self)} (name={self.
 ↪name} age={self.age} hobbies={self.hobbies})"


if __name__ == "__main__":
    p = Person("Jane", date(2001, 6, 10), ["dancing", "painting"])

    shallow_copy_p = copy.copy(p)
    deep_copy_p = copy.deepcopy(p)

    p.hobbies.append("yoga")

    print(p)
    print(shallow_copy_p)
    print(deep_copy_p)
```

```
Person object at 4566347472 (name=Jane age=23 hobbies=['dancing', 'painting',
'yoga'])
Person object at 4417543760 (name=Jane age=23 hobbies=['dancing', 'painting',
'yoga'])
Person object at 4566307264 (name=Jane age=23 hobbies=['dancing', 'painting'])
```

A class can also define its own copy implementation, by defining the magic methods `__copy__` and `__deepcopy__`.