OOP

Object-oriented programming
part. 2

Composition | Aggregation

Composition and aggregation are specialised form of Association. Whereas Association is a relationship between two classes without any rules.

Composition is a way to combine simple objects or data types into more complex ones

Aggregation is a type of composition. It Differs from ordinary composition in that it does not imply ownership.

Composition implies that the contained class cannot exist independently of the container. If the container is destroyed, the child is also destroyed.

Composition

Composition: Take for example a **Page** and a **Book**. The **Page** cannot exist without the **Book**, because the book is composed of **Pages**. If the **Book** is destroyed, the **Page** is also destroyed. In code, this usually refers to the child instance being created inside the container class:

```
class Book:
    def __init__(self):
       page1 = Page('This is content for page 1')
       page2 = Page('This is content for page 2')
       self.pages = [page1, page2]

class Page:
    def __init__(self, content):
        self.content = content

book = Book() # If I destroy this Book instance,
              # the Page instances are also destroyed
```

Aggregation

Aggregation: With an aggregation, the child can exist independently of the parent.
So thinking of a **Car** and an **Engine**, the **Engine** doesn't need to be destroyed when the **Car** is destroyed.

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

class Engine:
    def __init__(self):
        pass
engine = Engine()
car = Car(engine) # If I destroy this Car instance,
                  # the Engine instance still exists
```

classmethod

A class method receives the class as implicit first argument, just like an instance method receives the instance

- A class method is a method which is bound to the class and not the object of the class.

- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.

- It can modify a class state that would apply across all the instances of the class.

classmethod

@classmethod is a regular class method that has access to all the attributes of the class through which it was called. Hence, classmethod is a method that is bound to a class, not an instance of the class.

```
class MyClass():

    TOTAL_OBJECTS=0

    def __init__(self):
        MyClass.TOTAL_OBJECTS = MyClass.TOTAL_OBJECTS+1

    @classmethod
    def total_objects(cls):
        print("Total objects: ",cls.TOTAL_OBJECTS)
# Create objs
my_obj1 = MyClass()
my_obj2 = MyClass()
my_obj3 = MyClass()
# Call classmethod
MyClass.total_objects()
```

staticmethod

A static method does not receive an implicit first argument.

- A static method is also a method which is bound to the class and not the object of the class.

- A static method can’t access or modify class state.

- It is present in a class because it makes sense for the method to be present in class.

staticmethod

@staticmethod is like a normal function defined inside a class that doesn't have access to an instance, so it can be called without instantiating the class.

```python
import datetime


class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # a static method to check if a Person is adult or not.
    @staticmethod
    def is_adult(age):
        return age > 18


person1 = Person('mayank', 21)

# staticmethod called
print(Person.is_adult(22))  # True
```

property/getter/setter

property

- @property decorator can be used to make the method to be attribute-like accessible.
- @property decorator can be used to create getters & setters in pythonic way.
- Any kind of logic can be applied in getter or setter functions.

property

The @property decorator makes it easy to create properties in Python classes. Properties look like ordinary attributes (fields) of a class, but when they are read, a getter is called, when they are written, a setter is called, and when they are deleted, a deleter is called. Getter and divider are optional.

```
class Circle:
    def __init__(self, r):
        self.r = r
    @property
    def area(self):
        return 3.1415 * self.r**2
c = Circle(10)
print(c.area)
```

setattr/getattr

The setattr() function sets the value of the attribute of an object.

Syntax: setattr(object, name, value)

The getattr() method returns the value of the named attribute of an object.

If not found, it returns the default value provided to the function.

Syntax: getattr(object, name[, default]), which is equivalent to: object.name

setattr/getattr

```
 1 class Person:
 2     name = 'Adam'
 3
 4
 5 p = Person()
 6
 7 # setting attribute name to John
 8 setattr(p, 'name', 'John')
 9 # both 2 instructions will show the same output
10 print('Name is:', p.name)
11 print('Name is:', getattr(p, "name"))
12
13 # setting an attribute not present in Person
14 setattr(p, 'age', 23)
15 print('Age is:', p.age)  # Age is: 23
```

Slots

__slots__ allows to predefined set of attributes class instance will have. (avoid dynamically created attributes)

Reason to use:

1. Faster attribute access

2. Space savings in memory

When not to use:

- does not work for classes derived from some built-in types such as int, bytes and tuple.

- better not to use in case of multiple inheritance - brings extra complexity.

- definitely should not be used if dynamical attribute assignment should be performed.

Slots

```
 1 class SlotsClass:
 2     __slots__ = ('foo', 'bar')
 3
 4
 5 obj = SlotsClass()
 6 obj.foo = 5
 7 obj.__slots__
 8 # ('foo', 'bar')
 9 # if slots attribute is set, __dict__ dictionary will not be created
10 obj.__dict__
11 # Traceback (most recent call last):
12 #   File "python", line 8, in <module>
13 # AttributeError: 'SlotsClass' object has no attribute '__dict__'
```

Dataclasses

Data Class is a type of class that is used to store data without any functionality.

These data classes are just regular classes having the main purpose to store state and data without knowing the constraints and logic behind it.

Also it is easy to validate attribute values types.

Dataclasses

```python
import dataclasses


@dataclasses.dataclass
class Article:
    """A class for holding an article content"""

    # Attributes Declaration using Type Hints
    topic: str
    contributor: str
    language: str
    upvotes: int


# A DataClass object
article = Article("DataClasses", "nightfury1", "Python", 1)
print(article)  # Article(topic='DataClasses', contributor='nightfury1', ...)
print(article.upvotes)  # 1
# dataclass instance can be mutated
article.new_attr = 5
print(article.new_attr)  # 5
```

COPY

Frozen dataclass

Frozen dataclass looks very similar to regular dataclass, but it is immutable.

```
1  @dataclasses.dataclass(frozen=True)
2  class Book:
3      title: str
4      author: str
5
6
7  book = Book(title="Fahrenheit 451", author="Bradbury")
8
9  # frozen dataclass instance can not be mutated:
10 # dataclasses.FrozenInstanceError is raised
11 book.pages_num = 400
```

NamedTuple

The NamedTuple is a class that contains the data like a dictionary format stored under the ‘collections‘ module

Data is accessed using a specific key or can use the indexes.

```
import collections

# Declaring namedtuple()
Contributor = collections.namedtuple('Contributor', ['topic', 'author', 'post'])

# Adding values
c = Contributor('Difference between DataClass vs NamedTuple in Python',
                'night_fury1',
                'Technical Content Writer Intern')

# Access using index
print("The Article Topic : ", end="")
print(c[0])
# The Article Topic : Difference between DataClass vs NamedTuple in Python

# Access using name
print("The Article Contributor Name : ", end="")
print(c.author)
# The Article Contributor Name : night_fury1
```