POO

By: Guillermo Andres De Mendoza Corrales

Theory


OOP
Abstraction
Inheritance
Polymorphism
Encapsulation

Abstraction

Abstraction in Object-Oriented Programming (OOP) is a fundamental pillar that manages complexity by hiding unnecessary implementation details from the user, exposing only essential features. It models real-world objects by focusing on what an object does rather than how it does it, using abstract classes and interfaces to define templates for subclasses.

```
class Person:

    def __init__(self, name, gender, age, ethnicity):
        self.name = name
        self.gender = gender
        self.age = age
        self.ethnicity = ethnicity
```

```
p2 = Person("Laura", "F", 25, "Caucasian")
```

```
p1 = Person("Miyagi", "M", 45, "Asian")
```

Encapsulation

bundles data (attributes) and methods (behaviors) into a single unit, known as a class, while restricting direct access to some of an object's components. It acts as a protective shield, allowing data to be hidden and preventing unauthorized, direct modification by external code.

```python
class Person:

    def __init__(self, name, gender, age, ethnicity):
        self.__name = name
        self.__gender = gender
        self.__age = self.setAge(age)
        self.__ethnicity = ethnicity

    def setAge(self, age):
        if not isinstance(age, int):
            raise Exception("non valid age variable, only int")
        if age < 0 or age > 150:
            raise Exception("non valid age [0,150]")
        self.__age = age

    def getAge(self):
        return self.__age
```

```python
p1 = Person("Miyagi", "M", 45, "Asian")
```

```python
p1 = Person("Miyagi", "M", -10, "Asian")
```

```
Exception: non valid age [0,150]
```

```python
p1 = Person("Miyagi", "M", "10", "Asian")
```

```
Exception: non valid age variable, only int
```

Inheritance

Inheritance in object-oriented programming (OOP) is a core mechanism where a new class (subclass/child) is derived from an existing class (superclass/parent), inheriting its attributes and methods. This technique enables code reuse, establishes a natural "is-a" hierarchy, and allows child classes to extend or override parent functionality.

```python
class Person:

    def __init__(self, name, gender, age, ethnicity):
        self.name = name
        self.gender = gender
        self.age = age
        self.ethnicity = ethnicity

class Worker(Person):

    def __init__(self, name, gender, age, ethnicity, job_title, salary):
        super().__init__(name, gender, age, ethnicity)
        self.job_title = job_title
        self.salary = salary

    def work_info(self):
        return f"{self.name} tiene {self.age} años. Trabaja como {self.job_title} y gana ${self.salary}."
```

```python
w1 = Worker("Miyagi", "M", 45, "Asian", "manager", 6200000)
```

```python
w1.work_info()
```

```
'Miyagi tiene 45 años. Trabaja como manager y gana $6200000.'
```

Polymorphism

Polymorphism in Object-Oriented Programming (OOP) is the ability of different objects to respond to the same method call in their own unique way, enabling a single interface to represent multiple underlying forms. Derived from Greek for "many forms," it allows subclasses to redefine methods, enhancing code flexibility, reusability, and maintainability.

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

    def perform_task(self):
        return f"{self.name} está realizando una actividad general."

class Worker(Person):
    def __init__(self, name, job):
        super().__init__(name)
        self.job = job

    def perform_task(self):
        return f"{self.name} está trabajando como {self.job}."

class Student(Person):
    def perform_task(self):
        return f"{self.name} está estudiando para sus exámenes."
```

```python
people = [
    Worker("Carlos", "Carpintero"),
    Student("Sofía"),
    Person("Juan")
]

for p in people:
    print( p.perform_task() )
```

```
Carlos está trabajando como Carpintero.
Sofía está estudiando para sus exámenes.
Juan está realizando una actividad general.
```

POO
Special methods

__init__

Allow to make a new instance of a class (Object)

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
p1 = Person("Carlos", 15)
p2 = Person("Maria", 25)
p3 = Person("Luis",50)
```

__str__

Control the way the object will be printed

```python
class Person:

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

    def __str__(self):
        return f"Me llamo {self.name} y tengo {self.age} años"
```

```python
p1 = Person("Carlos", 15)
print( p1 )
```

```
Me llamo Carlos y tengo 15 años
```

__repr__

same as __str__ but works when the object is inside a collection

```
class Person:

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

    def __str__(self):
        return f"Me llamo {self.name} y tengo {self.age} años"

    def __repr__(self):
        return self.__str__()
```

```
p1 = Person("Carlos", 15)
p2 = Person("Maria", 25)
p3 = Person("Luis",50)

persons = [ p1, p2, p3 ]
print(persons)
```

```
[Me llamo Carlos y tengo 15 años, Me llamo Maria y tengo 25 años, Me llamo Luis y tengo 50 años]
```

__eq__

Compare if two objects should be considered the same

```
class Person:

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

    def __eq__(self, other):
        if not isinstance(other, Person):
            return False
        if self.age == other.age:
            if self.name == other.name:
                return True
        return False
```

```
p1 = Person("Max", 5)
p2 = Person("Max", 6)
p3 = Person("Max", 5)
```

```
print( p1 == p2 )
```

```
False
```

```
print( p1 == p3 )
```

```
True
```

__lt__ or __le__

Compare to other object for

lt: “less than”

le: “less of equals”

```
class Person:

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

    def __lt__(self, other):
        return self.age < other.age
```

```
p1 = Person("Carlos", 15)
p2 = Person("Maria", 25)

print( p1 < p2 )
```

```
True
```

__gt__ or __ge__

Compare to other object for

gt: “greather than”

ge: “great of equals”

```
class Person:

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

    def __gt__(self, other):
        return self.age > other.age
```

```
p1 = Person("Carlos", 15)
p2 = Person("Maria", 25)

print( p1 > p2 )
```

```
False
```

sorteable

Uses the implementacion of __lt__ and __gt__ to order a collection

```python
class Person:

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

    def __str__(self):
        return f"{self.age}:{self.name}"

    def __repr__(self):
        return self.__str__()

    def __gt__(self, other):
        if self.age == other.age:
            return self.name > other.name
        return self.age > other.age

    def __lt__(self, other):
        if self.age == other.age:
            return self.name < other.name
        return self.age < other.age
```

```python
p1 = Person("Carlos", 15)
p2 = Person("Maria", 25)
p3 = Person("Alejandro", 50)
p4 = Person("Pedro", 5)
p5 = Person("Estefania", 5)
p6 = Person("Max", 5)
p7 = Person("Guillermo", 30)

persons = [ p1, p2, p3, p4, p5, p6, p7 ]

print("Before: ", persons)

persons.sort()

print("After: ", persons)
```

```
Before:  [15:Carlos, 25:Maria, 50:Alejandro, 5:Pedro, 5:Estefania, 5:Max, 30:Guillermo]
After:  [5:Estefania, 5:Max, 5:Pedro, 15:Carlos, 25:Maria, 30:Guillermo, 50:Alejandro]
```

__len__

when called in a len() method

```
class Box:

    def __init__(self):
        self.elements = []
        self.count = 0

    def __len__(self):
        return self.count

    def add(self, element):
        self.elements.append( element )
        self.count += 1
```

```
box = Box()
box.add("Car")
box.add("Pencil")
box.add("Battery")
box.add("Clock")
box.add("Cable")

print( len(box) )
```

```
5
```

__getitem__

allow to call an element in the object by an index

```
class Box:

    def __init__(self):
        self.elements = []
        self.count = 0

    def __getitem__(self, index):
        return self.elements[index]

    def add(self, element):
        self.elements.append( element )
        self.count += 1
```

```
box = Box()
box.add("Car")
box.add("Pencil")
box.add("Battery")
box.add("Clock")
box.add("Cable")


print( box[2] )
```

```
Battery
```

__iter__

allow to call the object in a for loop

```
class Box:

    def __init__(self):
        self.elements = []
        self.count = 0

    def __iter__(self):
        for item in self.elements:
            yield item

    def add(self, element):
        self.elements.append( element )
        self.count += 1
```

```
box = Box()
box.add("Car")
box.add("Pencil")
box.add("Battery")
box.add("Clock")
box.add("Cable")

for b in box:
    print(b)
```

```
Car
Pencil
Battery
Clock
Cable
```

__contains__

Allow to ask if the object contains an element

```
class Box:

    def __init__(self):
        self.elements = []
        self.count = 0

    def __contains__(self, element):
        return element in self.elements

    def add(self, element):
        self.elements.append( element )
        self.count += 1
```

```
box = Box()
box.add("Car")
box.add("Pencil")
box.add("Battery")
box.add("Clock")
box.add("Cable")
```

```
print( "Sword" in box )
```

```
False
```

```
print( "Car" in box )
```

```
True
```

And many more (most importants):

1. Inicialización y Construcción

Controlan cómo nace y muere un objeto.

- `__init__`: Inicializador de la instancia.
- `__new__`: Constructor de la instancia (se ejecuta antes que `__init__`).
- `__del__`: Destructor (limpieza de memoria).

2. Representación (Strings)

- `__str__`: Para `print()` y `str()`. Orientado al usuario.
- `__repr__`: Para depuración y la consola. Orientado al desarrollador.
- `__format__`: Usado por f-strings y `format()`.
- `__bytes__`: Para `bytes(objeto)`.

3. Comparación (Ricos en lógica)

Estos devuelven booleanos y permiten usar operadores lógicos.

- `__eq__`: `==` (Igualdad)
- `__ne__`: `!=` (Desigualdad)
- `__lt__`: `<` (Menor que)
- `__le__`: `<=` (Menor o igual que)
- `__gt__`: `>` (Mayor que)
- `__ge__`: `>=` (Mayor o igual que)

4. Operaciones Aritméticas

Permiten que tus objetos sumen, resten o multipliquen.

Binarios	In-place (Asignación)	Unarios
`__add__` (+)	`__iadd__` (+=)	`__neg__` (-)
`__sub__` (-)	`__isub__` (-=)	`__pos__` (+)
`__mul__` (*)	`__imul__` (*=)	`__abs__` (abs())
`__truediv__` (/)	`__itruediv__` (/=)	`__invert__` (~)
`__floordiv__` (//)	`__ifloordiv__` (//=)	
`__mod__` (%)	`__imod__` (%=)	
`__pow__` (**)	`__ipow__` (**=)	

5. Contenedores y Colecciones

Hacen que tu objeto actúe como una lista, tupla o diccionario.

- `__len__`: `len(obj)`.
- `__getitem__`: `obj[key]` (obtener).
- `__setitem__`: `obj[key] = value` (asignar).
- `__delitem__`: `del obj[key]`.
- `__contains__`: `item in obj`.
- `__reversed__`: `reversed(obj)`.

6. Protocolo de Iteración

- `__iter__`: Devuelve el iterador (usado en `for`).
- `__next__`: Devuelve el siguiente elemento (usado por `next()`).

7. Manejo de Contexto (Context Managers)

- `__enter__`: Lo que ocurre al abrir `with`.
- `__exit__`: Lo que ocurre al cerrar `with` (gestión de errores).

8. Atributos y Reflexión

Controlan el acceso a las variables del objeto.

- `__getattr__`: Se ejecuta cuando un atributo **no** existe.
- `__getattribute__`: Se ejecuta **siempre** que accedes a un atributo.
- `__setattr__`: Al intentar escribir un atributo.
- `__delattr__`: Al intentar borrar un atributo.
- `__dir__`: Para `dir(obj)`.

9. Invocación y Otros

- `__call__`: Permite llamar al objeto como una función: `obj()`.
- `__hash__`: Para que el objeto sea "hasheable" (llave de dict o elemento de set).
- `__bool__`: Define si el objeto es `True` o `False`.
- `__copy__` / `__deepcopy__`: Para clonar objetos.