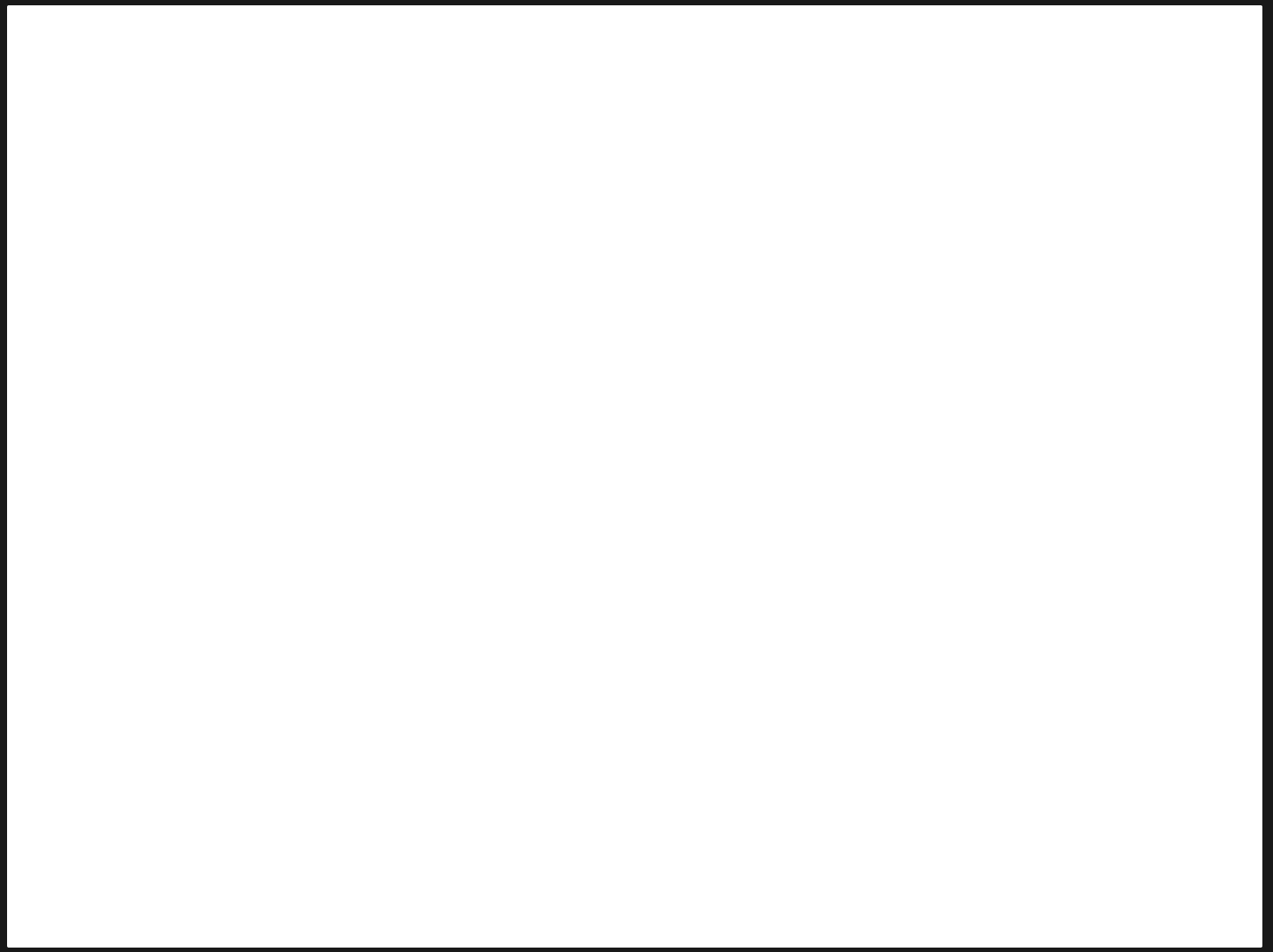
Los 4 pilares de OOP

Semana	Semana 12
Tipo	Lección
Descripción	Aprenda cuales son los 4 pilares fundamentales de todo código orientado a objetos.
Cuéntanos tu expe...	Cuéntanos tu experiencia
Migrado	☑

4 more properties



Qué son

- Los 4 pilares son 4 principios que todo código orientado a objetos debe de tener.

- Se consideran cultura general, así que es esencial aprenderlos y entenderlos a la perfección.
- Suelen ser preguntas de entrevista.

Herencia

Alek Castillo 04

Hablar del super

- Los humanos heredamos gran parte de nuestro comportamiento y físico de nuestros padres por genética.
- Al igual que los humanos, las clases pueden *heredar* características de otras clases.
  - Esto incluye sus `atributos` y `métodos`.
- A la clase que hereda cosas de otra le llamamos `clase hija`, mientras que a la clase de la que hereda le llamamos `clase padre`.

Por ejemplo, creemos una clase `Vehicle` que tenga 2 métodos: `turn_on` y `turn_off`:

```
class Vehicle: is_on = False wheel_number = 0 def turn_on(self): self.is_on = True
print(f"Vehicle with {self.wheel_number} wheels is on") def turn_off(self): self.is_on =
False print(f"Vehicle with {self.wheel_number} wheels is off")
```

Ahora creemos 2 clases que hereden de `Vehicle`:

```
class Car(Vehicle): wheel_number = 4
my_car = Car() my_car.turn_on()
my_car.turn_off()
```

```
Vehicle with 4 wheels is on
Vehicle with 4 wheels is off
```

```
class Bike(Vehicle): wheel_number = 2
my_bike = Bike() my_bike.turn_on()
my_bike.turn_off()
```

```
Vehicle with 2 wheels is on
Vehicle with 2 wheels is off
```

- Como podemos ver, las instancias de las clases `Car` y `Bike` tienen los mismos *métodos* y *atributos* que la clase `Vehicle` debido a que `Car` y `Bike` heredan de ella.
  - Son sus `clases hijas`, y `Vehicle` es la `clase padre`.
- Sin embargo, también tienen valores de `wheel_number` distintos porque sobre-escribieron el de `Vehicle`.

Herencia Multiple

- Python es de los pocos lenguajes en donde una clase puede `heredar` de **n cantidad de clases** y no solo de 1.

Por ejemplo, puedo tener tres clases padre:

```
class WalkerMixin: def walk(self): print("I'm walking!") class RunnerMixin: def
run(self): print("I'm running!") class FlyerMixin: def fly(self): print("I'm flying!")
```

Y una clase que herede de todas a la vez:

```
class SuperMan(WalkerMixin, RunnerMixin, FlyerMixin): pass clark_kent = SuperMan()
clark_kent.walk() clark_kent.run() clark_kent.fly()
```

```
I'm walking!
I'm running!
I'm flying!
```

- En caso de que dos (o mas) de estas clases tengan métodos o atributos con el mismo identificador, la primera tendrá prioridad y sobre-escribirá a las de las posteriores.

Por ejemplo:

```
class ClassA: name = "A" def my_method(): print("Hello") class ClassB: name = "B" def
my_method(): print("Goodbye")
```

Ahora veamos cual name es el que queda según el orden de herencia:

```
class ClassC(ClassA, ClassB): def
print_name(self): print(f"My name is
{self.name}") my_c = ClassC()
my_c.print_name() my_c.my_method()
```

```
My name is A
Hello
```

```
class ClassC(ClassB, ClassA): def
print_name(self): print(f"My name is
{self.name}") my_c = ClassC()
my_c.print_name() my_c.my_method()
```

```
My name is B
Goodbye
```

Clases Abstractas

- Otra manera de lograr esto es usando `clases abstractas`.
  - Estas son clases que no se pueden instanciar, sino que se usan *exclusivamente* para ser clases padre en una jerarquía de herencia.
  - Así mismo, pueden tener `métodos abstractos` que **deben ser sobre-escritos** por sus clases hijas.
    - Son como *contratos*: si heredas de mi, tienes que implementar este funcionamiento.

Por ejemplo, creemos una clase abstracta de `Animal` con un método abstracto de `reproduce`.

```
from abc import ABC, abstractmethod class Animal(ABC): def breath(self): pass def
born(self): pass @abstractmethod def reproduce(self): # Todas las especies de animales
deben reproducirse para sobrevivir # Pero lo pueden hacer de distintas maneras pass class
AsexualAnimal(Animal): def reproduce(self): print("Reproducing in an asexual manner")
class SexualAnimal(Animal): def reproduce(self, mate): print(f"Reproducing in a sexual
manner with {mate}") class OtherAnimal(Animal): pass asexual_animal = AsexualAnimal()
asexual_animal.reproduce() # -> Reproducing in an asexual manner sexual_animal_a =
SexualAnimal() sexual_animal_b = SexualAnimal() sexual_animal.reproduce(sexual_animal_b)
# -> Reproducing in a sexual manner with sexual_animal_b animal = Animal() # va a fallar
porque Animal es una clase abstracta other_animal = OtherAnimal() # va a fallar porque no
se sobre-escribió el método reproduce
```

```
from abc import ABC, abstractmethod class ReportEmailer(ABC): @abstractmethod def
generate_report(): # genera el reporte pass def email_report(): # envia el report por
email pass def generate_report_and_email(self): self.generate_report()
self.email_report()
```

Encapsulamiento

- En la mayoría de lenguajes de programación existe un nivel de acceso o de protección para los métodos y atributos.
  - `public` : pueden ser accesados desde cualquier lugar.
  - `protected` : pueden ser accesados desde los métodos de la clase y sus clases hijas.
  - `private` : pueden ser accesados solo desde los métodos de la clase.
- Esto se hace para proteger al código de un uso inadecuado por parte de los desarrolladores.

```
class BankAccount(): balance = 0 def __substract_balance(self, amount): self.balance -=
amount def __add_balance(self, amount): self.balance += amount def
send_money_to_account(self, account, amount): self.__substract_balance(amount)
account.__add_balance(amount) bank_account = BankAccount()
bank_account.__add_balance(5000)
```

- Python carece de esta funcionalidad, pero si *hipotéticamente*, funcionaría así:

```
class Person: name: string #public _date_of_birth: datetime #protected __sex: string
# private def __init__(self, name, date_of_birth, sex) self.name = name
self._date_of_birth = date_of_birth self.__sex = sex class Worker(Person): def
print_date_of_birth(self): print(self._date_of_birth) def print_sex(self):
print(self.__sex) my_person = Person("Juan", "2003/02/02", "Male")
print(my_person.name) # -> Juan print(my_person._date_of_birth) # -> error, ya que
_date_of_birth es un atributo protegido print(my_person.__sex) # -> error, ya que
__sex es un atributo privado my_worker = Worker("Joan", "1984/05/06", "Female")
my_worker.print_date_of_birth() # -> 1984/05/06 my_worker.print_sex() # -> error, ya
que __sex es un atributo privado
```

- Debido a que Python carece de esta protección, (y el ejemplo de arriba es hipotético - en realidad todos los casos funcionarían), lo que se usa *como estándar* es usar guiones bajos en sus identificadores para especificar su nivel de protección.
  - `name` es publico.
  - `_name` es protected.
  - `__name` es privado.
  - `get_age` es publico.
  - `_get_age` es protected.
  - `__get_age` es privado.

Abstracción

- La abstracción permite poder trabajar con objetos de manera efectiva sin necesidad de conocer todos los detalles internos de los mismos.
  - Por ejemplo, cuando manejamos un carro, no necesitamos saber cómo funciona su motor a nivel interno.
  - Solo necesitamos saber cómo usar los pedales y la manivela.
- Para ponerlo en palabras simples, es el acto de hacer que los métodos "*públicos*" de nuestros objetos sean lo más claros y sencillos posibles, y dividir su lógica entre métodos "*privados*". (uso comillas porque, reitero, en Python no existe tal cosa).

```
class Person: name: string _date_of_birth: datetime def __init__(self, name, date_of_birth) self.name = name self._date_of_birth = date_of_birth def get_age(self) return today() - self._date_of_birth`
```

Polimorfismo

- Viene de poli (varias) morfo (formas).
- Se refiere a que una sintaxis o método con el mismo `identificador` pueden tener comportamientos o formas distintas dependiendo del contexto en que se usen.
  - Por ejemplo, la función `range` dependiendo del numero de parámetros que tenga.
  - Otro ejemplo es el símbolo de `+`.
- En el caso de los objetos, podemos estandarizar ciertos identificadores para poder usarlos de manera intercambiable con clases que no estén relacionadas.
  - Por ejemplo, el método `reproduce` del ejemplo de arriba es un caso de polimorfísmo.

Aquí otro ejemplo, podemos tener una lista de objetos de tipo Vehículo y de tipo Computadora, y que todos tengan un método de “encender” y “apagar”:

```
class Vehiculo: is_on: bool ruedas: int def encender(self): self.is_on = True
```