PYTHON

Advanced OOP

OOP Advanced

OOP ADVANCED

Python core syntax

OOP ADVANCED

- Python core syntax
  - Python puede hacer operaciones sobre distintos tipos de datos:
    - Operador + aplicado a números o cadenas de caracteres.
    - Función len() aplicada a tuplas, listas, diccionarios o cadenas de caracteres.
    - *Python core syntax* cubre:
      - Operadores como '+', '-', '*', '/', '%' y otros similares;
      - Operadores como '==', '<', '>', '<=', 'in' y otros similares;
      - indexing, slicing, subscripting;
      - Funciones built-in functions como str(), len()
      - Reflexion – isinstance(), issubclass()

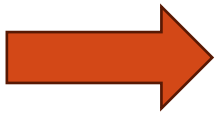
OOP ADVANCED

- Métodos mágicos o métodos de propósito especial → Responsables de realizar operaciones con objetos y susceptibles de ser programados .
- Los nombres de los métodos mágicos van 'rodeados' de dobles guiones bajos (double underscores o "dunder")

OOP ADVANCED

- Los métodos mágicos no se invocan directamente: se ejecutan como consecuencia de la evaluación de una expresión según las reglas de Python:
  - El operador + ejecuta el método __add__()
  - La función len() ejecuta el método __len__()

```
number = 10
print(number + 20)
```

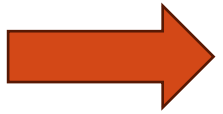


```
number = 10
print(number.__add__(20))
```

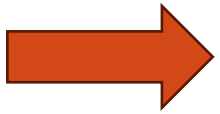
OOP ADVANCED

- Ejemplos de __add__ y __len__ para el operador + y la función len()

```
class Cosa:
    def __init__(self, valor):
        self.valor = valor
    def __add__(self, other):
        return self.valor + other.valor
    def __len__(self):
        return self.valor * 2

x = Cosa(5)
y = Cosa(8)
print(x+y)
print(len(x))
```



```
13
10
```



Si los métodos __add__ y/o __len__ no hubiesen estado programados, se produciría un TypeError indicando que el operador + y/o la función len() no están soportados

OOP ADVANCED

- Para obtener los métodos mágicos de un objeto con la función dir*(objeto o clase)* y con la función help(*objeto o clase*)
  - dir(*objeto o clase*) muestra la relación de métodos del objeto o de la clase.
  - help(*objeto o clase*) muestra toda la información sobre el objeto o la clase (incluyendo explicaciones detalladas).

OOP ADVANCED

Comparison methods

Function or operator	Magic method	Implementation meaning or purpose
==	__eq__(self, other)	equality operator
!=	__ne__(self, other)	inequality operator
<	__lt__(self, other)	less-than operator
>	__gt__(self, other)	greater-than operator
<=	__le__(self, other)	less-than-or-equal-to operator
>=	__ge__(self, other)	greater-than-or-equal-to operator

OOP ADVANCED

Unary operators and functions

Function or operator	Magic method	Implementation meaning or purpose
+	__pos__(self)	**unary** positive, like a = +b
-	__neg__(self)	**unary** negative, like a = -b
abs()	__abs__(self)	behavior for abs() function
round(a, b)	__round__(self, b)	behavior for round() function

OOP ADVANCED

Common, binary operators and functions

Function or operator	Magic method	Implementation meaning or purpose
+	__add__(self, other)	addition operator
-	__sub__(self, other)	subtraction operator
*	__mul__(self, other)	multiplication operator
//	__floordiv__(self, other)	integer division operator
/	__div__(self, other)	division operator
%	__mod__(self, other)	modulo operator
**	__pow__(self, other)	exponential (power) operator

OOP ADVANCED

Augumented operators and functions

By augumented assignment we should understand a sequence of unary operators and assignments like `a += 20`

Function or operator	Magic method	Implementation meaning or purpose
+=	__iadd__(self, other)	addition and assignment operator
-=	__isub__(self, other)	subtraction and assignment operator
*=	__imul__(self, other)	multiplication and assignment operator
//=	__ifloordiv__(self, other)	integer division and assignment operator
/=	__idiv__(self, other)	division and assignment operator
%=	__imod__(self, other)	modulo and assignment operator
**=	__ipow__(self, other)	exponential (power) and assignment operator

OOP ADVANCED

Type conversion methods

Python offers a set of methods responsible for the conversion of built-in data types.

Function	Magic method	Implementation meaning or purpose
int()	__int__(self)	conversion to integer type
float()	__float__(self)	conversion to float type
oct()	__oct__(self)	conversion to string, containing an octal representation
hex()	__hex__(self)	conversion to string, containing a hexadecimal representation

OOP ADVANCED

Object introspection

Python offers a set of methods responsible for representing object details using ordinary strings.

Function	Magic method	Implementation meaning or purpose
str()	__str__(self)	responsible for handling str() function calls
repr()	__repr__(self)	responsible for handling repr() function calls
format()	__format__(self, formatstr)	called when new-style string formatting is applied to an object
hash()	__hash__(self)	responsible for handling hash() function calls
dir()	__dir__(self)	responsible for handling dir() function calls
bool()	__nonzero__(self)	responsible for handling bool() function calls

OOP ADVANCED

Object retrospection

Following the topic of object introspection, there are methods responsible for object reflection.

Function	Magic method	Implementation meaning or purpose
isinstance(object, class)	__instancecheck__(self, object)	responsible for handling isinstance() function calls
issubclass(subclass, class)	__subclasscheck__(self, subclass)	responsible for handling issubclass() function calls

OOP ADVANCED

Object attribute access

Access to object attributes can be controlled via the following magic methods

Expression example	Magic method	Implementation meaning or purpose
object.attribute	__getattr__(self, attribute)	responsible for handling access to a non-existing attribute
object.attribute	__getattribute__(self, attribute)	responsible for handling access to an existing attribute
object.attribute = value	__setattr__(self, attribute, value)	responsible for setting an attribute value
del object.attribute	__delattr__(self, attribute)	responsible for deleting an attribute

OOP ADVANCED

Methods allowing access to containers

Containers are any object that holds an arbitrary number of other objects; containers provide a way to access the contained objects and to iterate over them. Container examples: list, dictionary, tuple, and set.

Expression example	Magic method	Implementation meaning or purpose
len(container)	__len__(self)	returns the length (number of elements) of the container
container[key]	__getitem__(self, key)	responsible for accessing (fetching) an element identified by the key argument
container[key] = value	__setitem__(self, key, value)	responsible for setting a value to an element identified by the key argument
del container[key]	__delitem__(self, key)	responsible for deleting an element identified by the key argument
for element in container	__iter__(self)	returns an iterator for the container
item in container	__contains__(self, item)	responds to the question: does the container contain the selected item?

OOP ADVANCED

- El listado completo de los métodos especiales de Python se puede consultar en el siguiente enlace:
  - https://docs.python.org/3/reference/datamodel.html#special-method-names

OOP ADVANCED

Inheritance and polymorphism

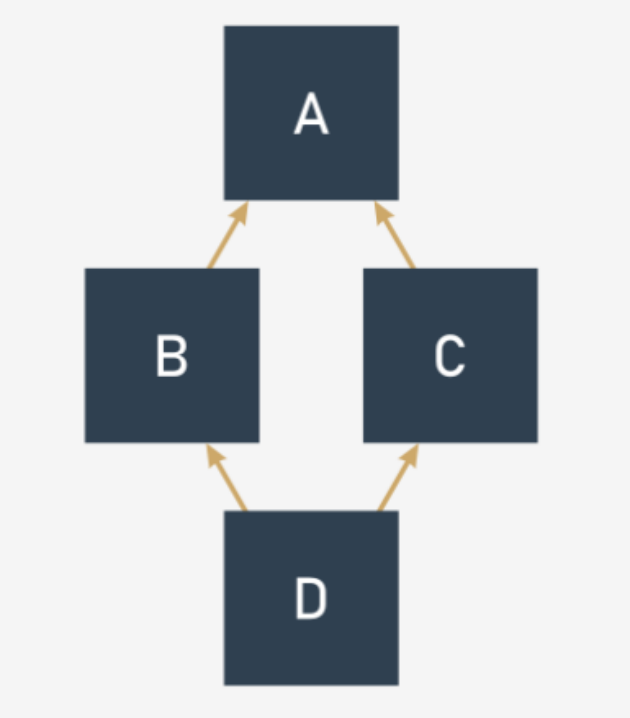
Inheritance is a pillar of OOP

OOP ADVANCED

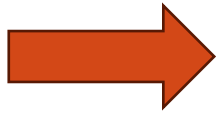
- ¿Qué es la herencia?
- Sintaxis: class A(B) → La clase A hereda de B.
- Python admite herencia múltiple:
  - class A(B,C)
  - Diferencias entre la herencia simple y la múltiple:
    - La herencia simple siempre es más sencilla, segura y fácil de entender y mantener.
    - La herencia múltiple puede generar complejidad en la sobreescritura de métodos y ambigüedad en el uso de super()
    - Es muy probable que implementando la herencia múltiple se viole el principio de responsabilidad única.
  - Recomendación: no utilizar la herencia múltiple y utilizar la composición en su lugar.

OOP ADVANCED

- MRO – Method Resolution Order
  - De abajo a arriba y de izquierda a derecha.



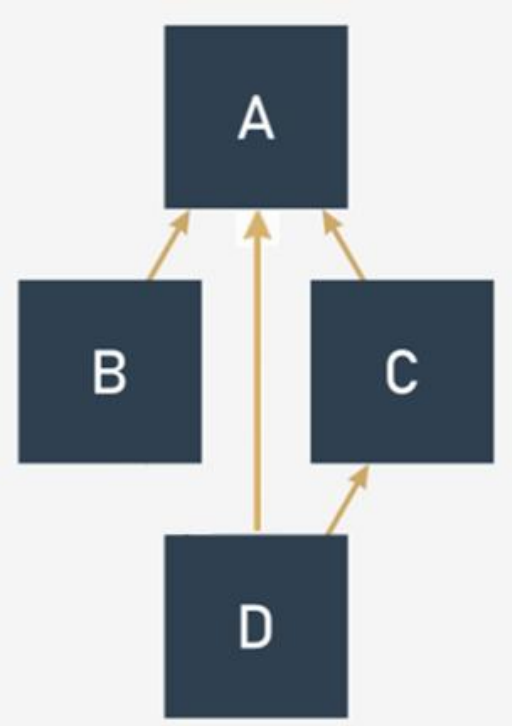

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

D().info()
```



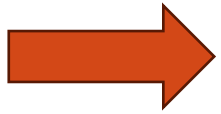
```
Class B
```

OOP ADVANCED

- MRO – Method Resolution Order
  - Inconsistencias.




```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(A, C):
    pass

D().info()
```

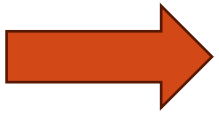


```
TypeError: Cannot create a
consistent method resolution
order (MRO) for bases A, C
```

OOP ADVANCED

- MRO – Method Resolution Order
  - Ejercicio:

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

class E(C, B):
    pass

D().info()
E().info()
```



¿?

OOP ADVANCED

- Polimorfismo
  - ¿Qué es el polimorfismo?
  - Relación entre el polimorfismo y los métodos especiales.
  - Relación entre el polimorfismo y la herencia. Ventajas de su uso combinado:
    - Permite la reutilización de código.
    - Mejora la estructura del código.
    - La forma en la que se invoca un método es uniforme entre objetos de distinto tipo.

OOP ADVANCED

- Polimorfismo
  - El polimorfismo en ausencia de herencia de clases debe ir acompañado de mecanismos de seguridad, ante la posible inexistencia de los métodos.
  - El error que provoca la invocación de un método inexistente es **AttributeError**

```
class Wax:
    def melt(self):
        print("Wax can be used to form a tool")

class Cheese:
    def melt(self):
        print("Cheese can be eaten")

class Wood:
    def fire(self):
        print("A fire has been started!")

for element in Wax(), Cheese(), Wood():
    try:
        element.melt()
    except AttributeError:
        print('No melt() method')
```

OOP ADVANCED

Extended function argument syntax

OOP ADVANCED

- Las funciones se pueden invocar:
  - Sin argumentos
  - Con un número determinado de argumentos en un orden concreto (argumentos posicionales)
  - Con algunos argumentos con un valor por defecto, que pueden no ser asignado en la invocación
  - Sin un orden determinado, asignando el nombre del argumento (keyword) en la llamada. En este caso, los argumentos posicionales irán al principio de la llamada.

OOP ADVANCED

- Las funciones se pueden invocar:
  - Con un número arbitario de argumentos posicionales y de *keyword* (palabra clave o con nombre).

```
def combiner(a, b, *args, **kwargs):
    print(a, type(a))
    print(b, type(b))
    print(args, type(args))
    print(kwargs, type(kwargs))


combiner(10, '20', 40, 60, 30, argument1=50, argument2='66')
```

OOP ADVANCED

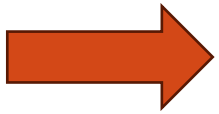
- Las funciones se pueden invocar:
  - Con un número arbitario de argumentos posicionales y de *keyword* (palabra clave o con nombre).
    - *args. Recoge todos los argumentos posicionales no recogidos.
      - Debe indicarse al final de los parámetros de la función, antes de **kwargs, excepto con parámetros con valor por defecto.
    - **kwargs. Recoge todos los argumentos *keyword* no recogidos.
      - Debe indicarse al final de los parámetros de la función, después de *args.

```
def f(*args, x=8, **kwargs):
    pass
f(3, 4, x=1, a=1, b=8)
```

OOP ADVANCED

- Las funciones se pueden invocar:
  - Con un número arbitario de argumentos posicionales y de *keyword* (palabra clave o con nombre).
    - Reenvío de argumentos a otra función.

```
def combiner(a, b, *args, **kwargs):
    super_combiner(*args, **kwargs)

def super_combiner(*my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_kwargs', my_kwargs)

combiner(10, '20', 40, 60, 30, argument1=50, argument2='66')
```

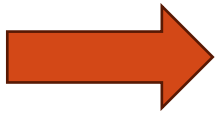


```
my_args: (40, 60, 30)
my_kwargs {'argument1': 50, 'argument2': '66'}
```

OOP ADVANCED

- Las funciones se pueden invocar:
  - Con un número arbitario de argumentos posicionales y de *keyword* (palabra clave o con nombre).
    - Reenvío de argumentos a otra función (sin asteriscos). Se recogen todos los parámetros en *args.

```
def combiner(a, b, *args, **kwargs):
    super_combiner(args, kwargs)

def super_combiner(*my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_kwargs', my_kwargs)

combiner(10, '20', 40, 60, 30, argument1=50,
argument2='66')
```

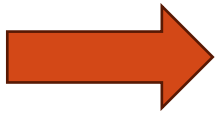


```
my_args: ((40, 60, 30), {'argument1': 50, 'argument2': '66'})
my_kwargs {}
```

OOP ADVANCED

- Las funciones se pueden invocar:
  - Con un número arbitario de argumentos posicionales y de *keyword* (palabra clave o con nombre).
    - Ejemplo de combinación de parámetros.

```
def combiner(a, b, *args, c=20, **kwargs):
    super_combiner(c, *args, **kwargs)
def super_combiner(my_c, *my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_c:', my_c)
    print('my_kwargs', my_kwargs)
combiner(1, '1', 1, 1, c=2, argument1=1, argument2='1')
```



```
my_args: (1, 1)
my_c: 2
my_kwargs {'argument1': 1, 'argument2': '1'}
```

OOP ADVANCED

Decoradores

OOP ADVANCED

- Un decorador es un patrón de diseño consistente en envolver un componente de software con otro que lo 'decora' o dota de una funcionalidad que no tenía.
- Python permite 'decorar' funciones, métodos y clases.
- Un decorador envuelve el elemento (típicamente una función) con otra función (o clase). La función 'decoradora' recibe la función 'decorada' como parámetro y devuelve una nueva función que podrá ser invocada posteriormente.
- La función decoradora puede tomar parámetros de la función decorada.

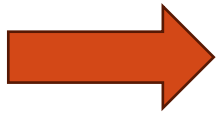
OOP ADVANCED

- Los decoradores se utilizan en:
  - la validación de argumentos;
  - la modificación de argumentos;
  - la modificación de objetos devueltos;
  - la medición del tiempo de ejecución;
  - el registro de mensajes;
  - la sincronización de subprocesos;
  - la refactorización de código;
  - el almacenamiento en caché.

OOP ADVANCED

- Ejemplo de decorador (versión simple):

```
def simple_decorator(function):
    print('We are about to call "{}"'.format(function.__name__))
    return function

def simple_hello():
    print("Hello from simple function!")


decorated = simple_decorator(simple_hello)
decorated()
```



```
We are about to call "simple_hello"
Hello from simple function!
```

OOP ADVANCED

- Ejemplo de decorador con *azúcar sintático*:

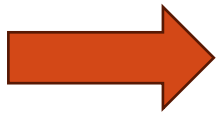
```
def simple_decorator(own_function):

    def internal_wrapper(*args, **kwargs):
        print('"{}" was called with the following
arguments'.format(own_function.__name__))
        print('\t{}\n\t{}\n'.format(args, kwargs))
        own_function(*args, **kwargs)
        print('Decorator is still operating')

    return internal_wrapper


@simple_decorator
def combiner(*args, **kwargs):
    print("\tHello from the decorated function; received
arguments:", args, kwargs)

combiner('a', 'b', exec='yes')
```

```
"combiner" was called with the following arguments
        ('a', 'b')
        {'exec': 'yes'}

        Hello from the decorated function; received arguments:
('a', 'b') {'exec': 'yes'}
Decorator is still operating
```

OOP ADVANCED

- Ejemplo de decorador que recibe argumentos:

```
def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            print('<strong>*</strong> Wrapping items from {}
with {}'.format(our_function.__name__, material))
            our_function(*args)
            print()
        return internal_wrapper
    return wrapper

@warehouse_decorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)

@warehouse_decorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)

pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
```

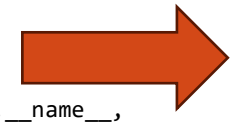


```
<strong>*</strong> Wrapping items from pack_books with kraft
We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')

<strong>*</strong> Wrapping items from pack_toys with foil
We'll pack toys: ('doll', 'car')
```

OOP ADVANCED

- Apilado de decoradores (influye el orden):

```
def big_container(collective_material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print('--The whole order would be packed with', collective_material)
            print()
        return internal_wrapper
    return wrapper

def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print('** Wrapping items from {} with {}'.format(our_function.__name__,
material))
        return internal_wrapper
    return wrapper

@big_container('plain cardboard')
@warehouse_decorator('bubble foil')
def pack_books(*args):
    print("We'll pack books:", args)

@warehouse_decorator('foil')
@big_container('colourful cardboard')
def pack_toys(*args):
    print("We'll pack toys:", args)

pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
```



```
We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')
** Wrapping items from pack_books with bubble foil
--The whole order would be packed with plain cardboard

We'll pack toys: ('doll', 'car')
--The whole order would be packed with colourful cardboard

** Wrapping items from internal_wrapper with foil
```

OOP ADVANCED

- Decoración de funciones con clases:
  - En Python un decorador puede ser una función o una clase.
  - El método especial de una clase que hace la función del decorador es __call__ y recibe los parámetros self, *args y **kwargs.
  - El método __call__ convierte a los objetos de una clase en invocables *(callable).*

```
class Motor:
    def __call__(self, *args, **kwargs):
        print("Arrancando...")

motor = Motor()
motor()
```

OOP ADVANCED

- Decoración de funciones con clases (**sin wrapper**):
  - Ventajas: la posibilidad de aprovechar las características de la POO (como la herencia) y la flexibilidad de poder dar soporte a la 'decoración' con los métodos de la clase.
  - El constructor de la clase **recibe la función a decorar**.
  - El método __call__ recibe **los argumentos de la función a decorar**.

```
class SimpleDecorator:
    def __init__(self, own_function):
        self.func = own_function

    def __call__(self, *args, **kwargs):
        print('Decorando: Antes de la llamada a la función decorada')
        self.func(*args, **kwargs)
        print('Decorando: Después de la llamada a la función
decorada')

@SimpleDecorator
def combiner(*args, **kwargs):
    print("Función decorada", args, kwargs)

combiner('a', 'b', exec='yes')
```

OOP ADVANCED

- Decoración de funciones con clases y argumentos (**con wrapper**):
  - El constructor de la clase recibe **los argumentos de la decoración**.
  - El método __call__ que recibe **la función a decorar**.
  - La función interna (wrapper) recibe **los argumentos de la función a decorar.**

```
class WarehouseDecorator:
    def __init__(self, material):
        self.material = material

    def __call__(self, own_function):
        def internal_wrapper(*args, **kwargs):
            print('<strong>*</strong> Wrapping items from {} with {}'.format(own_function.__name__,
self.material))
            own_function(*args, **kwargs)
            print()
        return internal_wrapper

@WarehouseDecorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)

@WarehouseDecorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)

pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
```

OOP ADVANCED

- Decoración clases:

```
def contador(clase):
    clase.atributos = clase.__getattribute__

    def detector(self, name):
        if name == 'kilometros':
            print('Se ha leído el kilometraje')
        return clase.atributos(self, name)

    clase.__getattribute__ = detector
    return clase

@contador
class Coche:
    def __init__(self, matricula):
        self.kilometros = 0
        self.matricula = matricula

coche = Coche('M-7797-HH')
print('El kilometraje es', coche.kilometros)
print('La matrícula es', coche.matricula)
```

OOP ADVANCED

Different faces of Python methods

OOP ADVANCED

- Tipos de métodos:
  - Métodos de instancia:
    - Los métodos de instancia reciben **self** (convención) como primer argumento y operan sobre la instancia en la que se encuentran.
  - Métodos de clase:
    - Pertenecen a la clase, no al objeto.
    - Se declaran con el decorador **@classmethod**
    - El primer parámetro es **cls** (convención)
  - Métodos estáticos:
    - No pertenecen a la clase (no reciben **cls**) por lo que no pueden modificar ni el estado del objeto ni el de la clase
    - Se declaran con el decorador **@staticmethod**

OOP ADVANCED

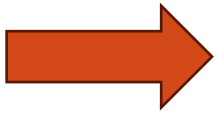
- Ejemplo de método de clase:
  - Nota: en *__init__* se utiliza **Example** y en *get_internal* **cls**

```
class Example:
    __internal_counter = 0

    def __init__(self, value):
        Example.__internal_counter +=1

    @classmethod
    def get_internal(cls):
        return '# of objects created: {}'.format(cls.__internal_counter)

print(Example.get_internal())

example1 = Example(10)
print(Example.get_internal())

example2 = Example(99)
print(Example.get_internal())
```

OOP ADVANCED

- Ejemplo de método de clase para crear un constructor alternativo:

```
class Car:
    def __init__(self, vin):
        print('Ordinary __init__ was called for', vin)
        self.vin = vin
        self.brand = ''

    @classmethod
    def including_brand(cls, vin, brand):
        print('Class method was called')
        _car = cls(vin) #Crea la instancia
        _car.brand = brand
        return _car

car1 = Car('ABCD1234')
car2 = Car.including_brand('DEF567', 'NewBrand')

print(car1.vin, car1.brand)
print(car2.vin, car2.brand)
```



```
Ordinary __init__ was called for ABCD1234
Class method was called
Ordinary __init__ was called for DEF567
ABCD1234
DEF567 NewBrand
```

OOP ADVANCED

- Ejemplo de método estático:

```
class Bank_Account:
    def __init__(self, iban):
        print('__init__ called')
        self.iban = iban

    @staticmethod
    def validate(iban):
        if len(iban) == 20:
            return True
        else:
            return False


account_numbers = ['8' * 20, '7' * 4, '2222']

for element in account_numbers:
    if Bank_Account.validate(element):
        print('We can use', element, ' to create a bank account')
    else:
        print('The account number', element, 'is invalid')
```

OOP ADVANCED

- Comparativa de métodos de clase y métodos estáticos:
  - a class method requires 'cls' as the first parameter and a static method does not;
  - a class method has the ability to access the state or methods of the class, and a static method does not; ¿CIERTO?
  - a class method is decorated by '@classmethod' and a static method by '@staticmethod';
  - a class method can be used as an alternative way to create objects, and a static method is only a utility method.

OOP ADVANCED

Abstract classes

OOP ADVANCED

- ¿Qué es una clase abstracta?
- El módulo de Python que proporciona la clase auxiliar (**ABC**) para definir clases abstractas es **abc**.
- Para definir una clase abstracta:
  - Importar el módulo **abc**
  - Heredar la clase de **abc.ABC**
  - Decorar los métodos abstractos con el decorador **@abstractmethod**
  - Si la clase derivada no implementa los métodos abstractos produce un error de tipo **TypeError** al intentar crear una instancia de esta.
  - Si hay herencia múltiple de varias clases abstractas se deben implementar todos los métodos abstractos de todas las clases base.

OOP ADVANCED

- Ejemplo de clase abstracta:

```
import abc

class BluePrint(abc.ABC):
    @abc.abstractmethod
    def hello(self):
        pass

class GreenField(BluePrint):
    def hello(self):
        print('Welcome to Green Field!')


gf = GreenField()
gf.hello()
```

OOP ADVANCED

Attribute encapsulation

OOP ADVANCED

- ¿Qué es la encapsulación?
  - *__atributo* → Hace 'invisible' el atributo (**Attribute Error**)
    - Se puede acceder a él como *_Clase__atributo.*

```
class Cosa():
    def __init__(self) -> None:
        self.__atributo1 = "Uno"

cosa = Cosa()
print(cosa.__atributo1) #Error
print(cosa._Cosa__atributo1) #Correcto
```

OOP ADVANCED

- ¿Qué es la encapsulación?
  - *__atributo* → Hace 'invisible' el atributo (**Attribute Error**)
    - Se puede acceder a él como *_Clase__atributo.*

```
class Cosa():
    def __init__(self) -> None:
        self.__atributo1 = "Uno"

cosa = Cosa()
#print(cosa.atributo1) #AttributeError
#print(cosa.__atributo1) #AttributeError
print(cosa._Cosa__atributo1) #Correcto
```

OOP ADVANCED

- ¿Qué es la encapsulación?
  - La función **property()** y el correspondiente decorador **@property** permiten leer un atributo encapsulado →**getter**

```
class Cosa():
    def __init__(self) -> None:
        self.__atributo1 = "Uno"

    @property
    def atributo1(self):
        return self.__atributo1

cosa = Cosa()
print(cosa.atributo1) #Correcto
print(cosa.__atributo1) #AttributeError
print(cosa._Cosa__atributo1) #Correcto
```

OOP ADVANCED

- ¿Qué es la encapsulación?
  - Los decoradores ***@atributo.setter*** y ***@atributo.deleter*** encapsulan la funcionalidad de modificación y eliminación del atributo.

```
class TankError(Exception):
    pass

class Tank:
    def __init__(self, capacity):
        self.capacity = capacity
        self.__level = 0

    @property
    def level(self):
        return self.__level

    @level.setter
    def level(self, amount):
        self.__level = amount

    @level.deleter
    def level(self):
        if self.__level > 0:
            print('Eliminado level')
        self.__level = None

tanque = Tank(1000)
print(tanque.level) #0
tanque.level=500
print(tanque.level) #500
delattr(tanque, "level") #Eliminado level
print(tanque.level) #None
```

OOP ADVANCED

Composition vs Inheritance - two ways to the same destination: Inheritance

OOP ADVANCED

- El principal objetivo de la herencia es la reutilización de código.
- Un mal uso de la herencia (especialmente de la múltiple) puede generar una estructura de clases enorme y compleja, con código difícil de entender, depurar y ampliar →Explosión de clases (*class explosion*) → Antipatrón (*AntiPattern*).
- Alternativa→Composición.
- Herencia → ***Is a relation***
- Composición → ***Has a relation***

OOP ADVANCED

- **Inheritance extends a class's capabilities by adding new components and modifying existing ones**; in other words, the complete recipe is contained inside the class itself and all its ancestors; the object takes all the class's belongings and makes use of them;
- **Composition projects a class as a container (called a composite) able to store and use other objects (derived from other classes) where each of the objects implements a part of a desired class's behavior**. It's worth mentioning that blocks are **loosely coupled** with the composite, and those blocks could be exchanged any time, even during program runtime.

OOP ADVANCED

- Ejemplo de composición:

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


class GasEngine:
    def __init__(self, horse_power):
        self.hp = horse_power

    def start(self):
        print('Starting {}hp gas engine'.format(self.hp))


class DieselEngine:
    def __init__(self, horse_power):
        self.hp = horse_power

    def start(self):
        print('Starting {}hp diesel engine'.format(self.hp))


my_car = Car(GasEngine(4))
my_car.engine.start()
my_car.engine = DieselEngine(2)
my_car.engine.start()
```

OOP ADVANCED

- La composición proporciona mayor **flexibilidad** → las clases se construyen uniendo componentes que podrían ampliarse o reducirse.
- La composición responde mejor a los **cambios de requisitos**.
- Inconveniente: el desarrollador debe asegurar que las clases que componen el objeto proporcionan una interfaz común.

OOP ADVANCED

- ¿Herencia o composición?
  - No son excluyentes.
  - Si la relación es del tipo "es un"→Herencia.
  - Si la relación es del tipo "tiene un"→Composición.

OOP ADVANCED

- Ejemplo de uso combiando de herencia y composición (1/2):

```
class Base_Computer:
    def __init__(self, serial_number):
        self.serial_number = serial_number


class Personal_Computer(Base_Computer):
    def __init__(self, sn, connection):
        super().__init__(sn)
        self.connection = connection
        print('The computer costs $1000')


class Connection:
    def __init__(self, speed):
        self.speed = speed

    def download(self):
        print('Downloading at {}'.format(self.speed))
```

OOP ADVANCED

- Ejemplo de uso combiando de herencia y composición (2/2):

```
class DialUp(Connection):
    def __init__(self):
        super().__init__('9600bit/s')

    def download(self):
        print('Dialling the access number ... '.ljust(40), end='')
        super().download()


class ADSL(Connection):
    def __init__(self):
        super().__init__('2Mbit/s')

    def download(self):
        print('Waking up modem  ... '.ljust(40), end='')
        super().download()
```

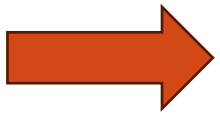
OOP ADVANCED

Inheriting properties from built-in classes

OOP ADVANCED

- Ejemplo: diccionario que registra las fechas de acceso.

```
from datetime import datetime

class MonitoredDict(dict):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.log = list()
        self.log_timestamp('MonitoredDict created')

    def __getitem__(self, key):
        val = super().__getitem__(key)
        self.log_timestamp('value for key [{}] retrieved'.format(key))
        return val

    def __setitem__(self, key, val):
        super().__setitem__(key, val)
        self.log_timestamp('value for key [{}] set'.format(key))

    def log_timestamp(self, message):
        timestampStr = datetime.now().strftime("%Y-%m-%d (%H:%M:%S.%f)")
        self.log.append('{} {}'.format(timestampStr, message))

kk = MonitoredDict()
kk[10] = 15
kk[20] = 5

print('Element kk[10]:', kk[10])
print('Whole dictionary:', kk)
print('Our log book:\n')
print('\n'.join(kk.log))
```



```
Element kk[10]: 15
Whole dictionary: {10: 15, 20: 5}
Our log book:

2024-09-08 (19:49:25.741076) MonitoredDict created
2024-09-08 (19:49:25.741076) value for key [10] set
2024-09-08 (19:49:25.741076) value for key [20] set
2024-09-08 (19:49:25.741076) value for key [10] retrieved
```

OOP ADVANCED

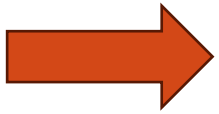
- Ejemplo: diccionario que controla la inserción de códigos IBAN

```
import random

class IBANValidationError(Exception):
    pass

def validateIBAN(iban):
    iban = iban.replace(' ', '')
    if not iban.isalnum():
        raise IBANValidationError("You have entered invalid characters.")
    elif len(iban) < 15:
        raise IBANValidationError("IBAN entered is too short.")
    elif len(iban) > 31:
        raise IBANValidationError("IBAN entered is too long.")
    else:
        iban = (iban[4:] + iban[0:4]).upper()
        iban2 = ''
        for ch in iban:
            if ch.isdigit():
                iban2 += ch
            else:
                iban2 += str(10 + ord(ch) - ord('A'))
        ibann = int(iban2)
        if ibann % 97 != 1:
            raise IBANValidationError("IBAN entered is invalid.")
        return True
```

OOP ADVANCED

- Ejemplo: diccionario que controla la inserción de códigos IBAN

```
class IBANDict(dict):
    def __setitem__(self, _key, _val):
        if validateIBAN(_key):
            super().__setitem__(_key, _val)

    def update(self, *args, **kwargs):
        for _key, _val in dict(*args, **kwargs).items():
            self.__setitem__(_key, _val)

my_dict = IBANDict()
keys = ['GB72 HBZU 7006 7212 1253 00', 'FR76 30003 03620 00020216907 50', 'DE02100100100152517108']

for key in keys:
    my_dict[key] = random.randint(0, 1000)

print('The my_dict dictionary contains:')
for key, value in my_dict.items():
    print("\t{} -> {}".format(key, value))

try:
    my_dict.update({'dummy_account': 100})
except IBANValidationError:
    print('IBANDict has protected your dictionary against incorrect data insertion')
```



```
The my_dict dictionary contains:
        GB72 HBZU 7006 7212 1253 00 -> 792
        FR76 30003 03620 00020216907 50 -> 958
        DE02100100100152517108 -> 40
IBANDict has protected your dictionary against incorrect data insertion
```