



we design and
create
professional
quality software

Language - Agnostic

Language-agnostic programming

La programación o secuencias de comandos independientes del lenguaje es un paradigma de desarrollo de software en el que se elige un lenguaje en particular debido a su idoneidad para una tarea en particular, y no simplemente por el conjunto de habilidades disponibles dentro de un equipo de desarrollo.

Modern Portfolio Designed

Te quedo? Listo
(Rodrigo Bueno)



Programación Orientada a Objetos

PYTHON

POO

INTRODUCCION A LAS POO EN PYTHON.

Programación Orientada a Objetos

 python Repaso

Se trata de un paradigma de programación introducido en los años 1970s, pero que no se hizo popular hasta años más tarde.

Este modo o paradigma de programación nos permite organizar el código de una manera que se asemeja bastante a como pensamos en la vida real, utilizando las famosas **clases**. Estas nos permiten agrupar un conjunto de variables y funciones que veremos a continuación.

Cosas de lo más cotidianas como un perro o un coche pueden ser representadas con clases. Estas clases tienen diferentes características, que en el caso del perro podrían ser la edad, el nombre o la raza. Llamaremos a estas características, atributos.

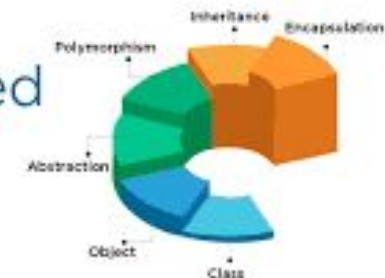
Por otro lado, las clases tienen un conjunto de funcionalidades o cosas que pueden hacer. En el caso del perro podría ser andar o ladrar. Llamaremos a estas funcionalidades métodos.

Por último, pueden existir diferentes tipos de perro. Podemos tener uno que se llama Toby o el del vecino que se llama Laika. Llamaremos a estos diferentes tipos de perro objetos. Es decir, el concepto abstracto de perro es la clase, pero Toby o cualquier otro perro particular será el objeto.

La programación orientada a objetos está basada en 6 principios o pilares básicos:

- Herencia
- Cohesión
- Abstracción
- Polimorfismo
- Acoplamiento
- Encapsulamiento

Object Oriented
Programming
with **Python**



Motivación POO

Repaso

Una vez repasada la programación orientada a objetos puede parecer bastante lógica, pero no es algo que haya existido siempre, y de hecho hay muchos lenguajes de programación que no la soportan.

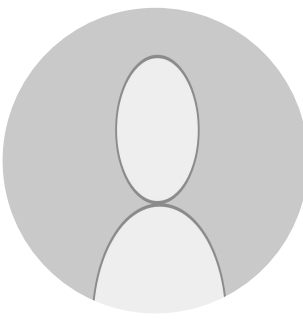
En parte surgió debido a la creciente complejidad a la que los programadores se iban enfrentando conforme pasaban los años. En el mundo de la programación hay gran cantidad de aplicaciones que realizan tareas muy similares y es importante identificar los patrones que nos permiten no reinventar la rueda. La programación orientada a objetos intentaba resolver esto.

Uno de los primeros mecanismos que se crearon fueron las **funciones**, que permiten agrupar bloques de código que hacen una tarea específica bajo un nombre. Algo muy útil ya que permite también reusar esos módulos o funciones sin tener que copiar todo el código, tan solo la llamada.

Las funciones resultaron muy útiles, pero no eran capaces de satisfacer todas las necesidades de los programadores. Uno de los problemas de las funciones es que sólo realizan unas operaciones con unos datos de entrada para entregar una salida, pero no les importa demasiado conservar esos datos o mantener algún tipo de estado. Salvo que se devuelva un valor en la llamada a la función o se usen variables globales, todo lo que pasa dentro de la función queda olvidado, y esto en muchos casos es un problema.

Imaginemos que tenemos un juego con naves espaciales moviéndose por la pantalla. Cada nave tendrá una posición (x,y) y otros parámetros como el tipo de nave, su color o tamaño. Sin hacer uso de clases y POO, tendremos que tener una variable para cada dato que queremos almacenar: coordenadas, color, tamaño, tipo. El problema viene si tenemos 10 naves, ya que nos podríamos juntar con un número muy elevado de variables. Todo un desastre.

En el mundo de la programación existen tareas muy similares al ejemplo con las naves, y en respuesta a ello surgió la programación orientada a objetos. Una herramienta perfecta que permite resolver cierto tipo de problemas de una forma mucho más sencilla, con menos código y más organizado. Agrupa bajo una clase un conjunto de variables y funciones, que pueden ser reutilizadas con características particulares creando objetos.





POO INTRODUCCION A LAS POO EN PYTHON.

Powered By Chat GPT

Stage 1

Definiendo clases

Definiendo Clases

Repasada ya la parte teórica, vamos a ver como podemos hacer uso de la programación orientada a objetos en Python. Lo primero es crear una clase, para ello usaremos el ejemplo de perro.

Creando una clase vacía

```
class Perro:  
    pass
```

Se trata de una clase vacía y sin mucha utilidad práctica, pero es la mínima clase que podemos crear. Nótese el uso del **pass** que no hace realmente nada, pero daría un error si después de los **:** no tenemos contenido.

Ahora que tenemos la clase, podemos crear un objeto de la misma. **Podemos hacerlo como si de una variable normal se tratase.**

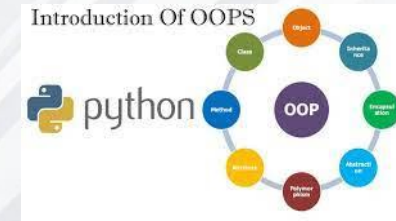
Nombre de la variable igual a la clase con (). Dentro de los paréntesis irían los parámetros de entrada si los hubiera.

```
# Creamos un objeto de la clase perro  
mi_perro = Perro()
```



Powered By
ChatGPT & Python

AI ChatGPT Python code



Definiendo atributos

A continuación vamos a añadir algunos atributos a nuestra clase.

*Antes de nada es importante distinguir que **existen dos tipos de atributos en Python:***

- **Atributos de instancia:** Pertenecen a la instancia de la clase o al objeto. Son atributos particulares de cada instancia (Objeto), en nuestro caso de cada perro.
- **Atributos de clase:** Se trata de atributos que pertenecen a la clase, por lo tanto serán comunes para todos los objetos o Instancias de la Clase.

```
class Perro:
```

```
    # Atributo de clase
```

```
    especie = 'mamífero'
```

```
    # El método __init__ es llamado al crear el objeto (constructor)
```

```
    def __init__(self, nombre, raza):
```

```
        print(f"Creando perro {nombre}, {raza}")
```

```
    # Atributos de instancia
```

```
    self.nombre = nombre
```

```
    self.raza = raza
```



Veámoslo en mas detalle..



Definiendo atributos

Atributos de Instancia

Empecemos creando un par de atributos de **instancia** para nuestro perro, el nombre y la raza.

Para ello creamos un método `__init__` que será llamado automáticamente cuando creemos un objeto. Se trata del constructor !!!.

class Perro:

El método `__init__` es llamado al crear el objeto

def `__init__(self, nombre, raza):`

`print(f"Creando perro {nombre}, {raza}")`

Atributos de instancia

`self.nombre = nombre`

`self.raza = raza`



Powered By
ChatGPT & Python

AI GPT: En Python, los atributos de instancia son variables que pertenecen a una instancia específica de una clase. Cada instancia de la clase puede tener valores diferentes para sus atributos de instancia.

Ahora que hemos definido el método `init` con dos parámetros de entrada, podemos crear el objeto pasando el valor de los atributos. Usando `type()` podemos ver como efectivamente el objeto es de la clase Perro.

`mi_perro = Perro("Toby", "Bulldog")`

`print(type(mi_perro))`

Creando perro Toby, Bulldog

`<class '__main__.Perro'>`

Seguramente te hayas fijado en el **self** que se pasa como parámetro de entrada del método.

Es una variable que representa la instancia de la clase, y deberá estar siempre ahí.

El uso de `__init__` y el doble `__` no es una coincidencia. Cuando veas un método con esa forma, significa que está reservado para un uso especial del lenguaje.

En este caso sería lo que se conoce como constructor. Hay gente que llama a estos métodos mágicos.

Objetos que se creen de la clase perro compartirán ese atributo de la instancia de la clase, ya que pertenecen a la misma.



Definiendo atributos

Atributos de Clase

A diferencia de los atributos de instancia que son visibles en el nivel de objeto, los atributos de clase siguen siendo los mismos para todos los objetos.

Consulte el siguiente ejemplo para demostrar el uso de atributos de nivel de clase.

```
class BookStore:  
    instances = 0  
    def __init__(self, attrib1, attrib2):  
        self.attrib1 = attrib1  
        self.attrib2 = attrib2  
        BookStore.instances += 1
```

```
b1 = BookStore("", "")  
b2 = BookStore("", "")
```

```
print("BookStore.instances:", BookStore.instances)
```

En este ejemplo, "instances" es un atributo de nivel de clase. Puede acceder a él utilizando el nombre de la clase. Tiene el total no. de instancias creadas. Hemos creado dos instancias de la clase <Librería>. Por lo tanto, la ejecución del ejemplo debe imprimir "2" como salida.

```
# output  
BookStore.instances: 2
```



Powered By
ChatGPT & Python

Demostración de la clase Python

Aquí hay un ejemplo en el que estamos construyendo una clase BookStore y creando instancias de su objeto con diferentes valores.

Crear una clase BookStore en Python

```
class BookStore:
```

```
    noOfBooks = 0
```

```
    def __init__(self, title, author):
```

```
        self.title = title
```

```
        self.author = author
```

```
        BookStore.noOfBooks += 1
```

```
    def bookInfo(self):
```

```
        print("Book title:", self.title)
```

```
        print("Book author:", self.author, "\n")
```

```
# Create a virtual book store
```

```
b1 = BookStore("Great Expectations", "Charles Dickens")
```

```
b2 = BookStore("War and Peace", "Leo Tolstoy")
```

```
b3 = BookStore("Middlemarch", "George Eliot")
```

```
# call member functions for each object
```

```
b1.bookInfo()
```

```
b2.bookInfo()
```

```
b3.bookInfo()
```

```
print("BookStore.noOfBooks:", BookStore.noOfBooks)
```

Puede abrir IDLE o cualquier otro IDE de Python, guardar el código anterior en algún archivo y ejecutar el programa. En este ejemplo, hemos creado tres objetos de la clase BookStore, es decir, b1, b2 y b3. Cada uno de los objetos es una instancia de la clase BookStore.



Python Class & OOP Fundamentals

Class

self Reference to an object

__init__ Constructor method

class attrib Same for all objects

instance attrib Object specific data

```
class BookStore:
    instances = 0
    def __init__(self, attrib1, attrib2):
        self.attrib1 = attrib1
        self.attrib2 = attrib2
        BookStore.instances += 1
```

```
# output
```

```
Book title: Great Expectations
```

```
Book author: Charles Dickens
```

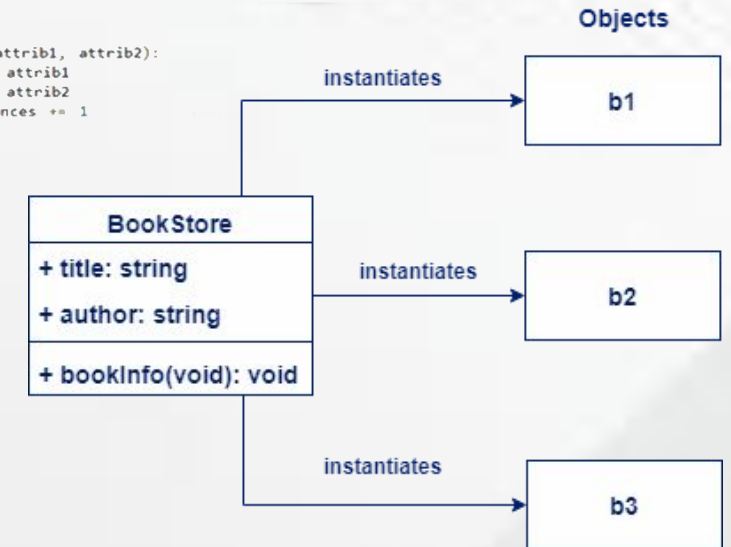
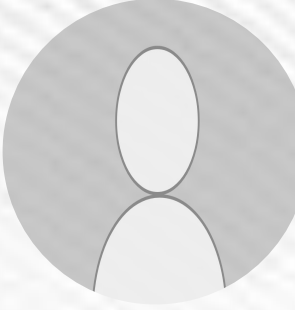
```
Book title: War and Peace
```

```
Book author: Leo Tolstoy
```

```
Book title: Middlemarch
```

```
Book author: George Eliot
```

```
BookStore.noOfBooks: 3
```



Python Class and Object Creation

Diagrama UML de la clase BookStore
El diagrama UML del código anterior es el siguiente.

Clase y objetos de Python (diagrama UML)
Después de ejecutar el código del ejemplo, debería ver el siguiente resultado.





POO METODOS EN PYTHON



Definiendo métodos

En realidad cuando usamos `__init__` anteriormente ya estábamos definiendo un método, solo que uno especial. A continuación vamos a ver como definir métodos que le den alguna funcionalidad interesante a nuestra clase, siguiendo con el ejemplo de perro.

```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

    # Atributos de instancia
    self.nombre = nombre
    self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")
```

Vamos a codificar dos métodos, ladrar y caminar. El primero no recibirá ningún parámetro y el segundo recibirá el número de pasos que queremos andar. Como hemos indicado anteriormente `self` hace referencia a la instancia de la clase. Se puede definir un método con `def` y el nombre, y entre `()` los parámetros de entrada que recibe, donde siempre tendrá que estar `self` el primero.

Por lo tanto si creamos un objeto `mi_perro`, podremos hacer uso de sus métodos llamándolos con `.` y el nombre del método. Como si de una función se tratase, pueden recibir y devolver argumentos.

```
mi_perro = Perro("Toby", "Bulldog")
mi_perro.ladra()
mi_perro.camina(10)
```

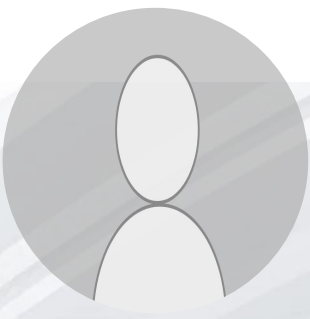
```
# Creando perro Toby, Bulldog
# Guau
# Caminando 10 pasos
```



Powered By
ChatGPT & Python



Métodos en Python: instancia, clase y estáticos



En otros apartados hemos visto como se pueden crear métodos con `def` dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia. Pues bien, haciendo uso de los **decoradores**, es posible crear diferentes tipos de métodos:

- Los métodos de **instancia** “normales” que ya hemos visto como `metodo()`
- Métodos de **clase** usando el decorador `@classmethod`
- Métodos **estáticos** usando el decorador `@staticmethod`

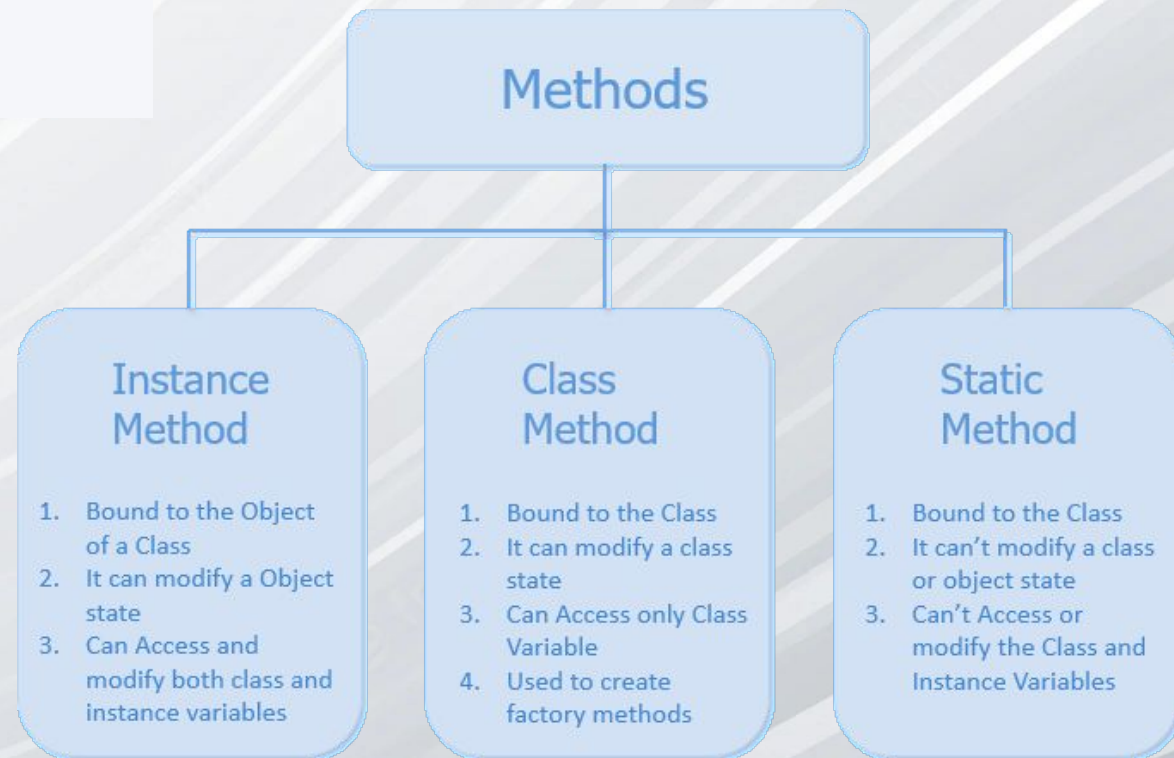
class Clase:

```
def metodo(self):  
    return 'Método normal', self
```

```
@classmethod  
def metododeclase(cls):  
    return 'Método de clase', cls
```

```
@staticmethod  
def metodoestatico():  
    return "Método estático"
```

Veamos su comportamiento en detalle uno por uno.



Métodos en Python: instancia, clase y estáticos

Métodos de instancia

Los métodos de instancia son los métodos normales, de toda la vida, que hemos visto anteriormente. Reciben como parámetro de entrada `self` que hace referencia a la instancia que llama al método. También pueden recibir otros argumentos como entrada.

Para saber más: El uso de "`self`" es totalmente arbitrario. Se trata de una convención acordada por los usuarios de Python, usada para referirse a la instancia que llama al método, pero podría ser cualquier otro nombre. Lo mismo ocurre con "`cls`", que veremos a continuación.

```
class Clase:  
    def metodo(self, arg1, arg2):  
        return 'Método normal', self
```

Y como ya sabemos, una vez creado un objeto pueden ser llamados.

```
mi_clase = Clase()  
mi_clase.metodo("a", "b")  
# ('Método normal', <__main__.Clase at 0x10b9daa90>)
```

En vista a esto, los métodos de instancia:

Pueden acceder y modificar los atributos del objeto.

Pueden acceder a otros métodos.

Dado que desde el objeto `self` se puede acceder a la clase con ``self.class``, también pueden modificar el estado de la clase

Instance Method

1. Bound to the Object of a Class
2. It can modify a Object state
3. Can Access and modify both class and instance variables

Métodos en Python: instancia, clase y estáticos

Métodos de clase (classmethod)

A diferencia de los métodos de instancia, los métodos de clase reciben como argumento cls, que hace referencia a la clase. Por lo tanto, pueden acceder a la clase pero no a la instancia.

```
class Clase:  
    @classmethod  
    def metododeclase(cls):  
        return 'Método de clase', cls
```

Se pueden llamar sobre la clase.

```
Clase.metododeclase()  
# ('Método de clase', __main__.Clase)
```

Pero también se pueden llamar **sobre el objeto**.

```
mi_clase = Clase()  
mi_clase.metododeclase()  
# ('Método de clase', __main__.Clase)
```

Por lo tanto, los métodos de clase:

No pueden acceder a los atributos de la instancia.
Pero si pueden modificar los atributos de la clase.

Class Method

1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods

Métodos en Python: instancia, clase y estáticos



Métodos de clase (classmethod) cont...

Los métodos de clase son métodos que se usan para realizar operaciones en el ámbito de clase y NO al generar una instancia. Para crear un método de clase, debemos usar el decorador `@classmethod` y en el método debemos añadir como primer parámetro `cls`.

Al igual que pasaba con `self`, podemos emplear cualquier nombre, aunque por convención se recomienda `cls` que viene de `class`. Este parámetro se emplea para acceder a los atributos de la clase y a otros métodos, siempre y cuando también sean métodos de clase o estáticos.

```
class Perro:
    peso = 30    #Atributo de Clase

    def __init__(self, peso):    #Atributo de Instancia
        self.peso = peso

    @classmethod    #Metodo de Clase
    def get_peso_promedio(cls):
        return cls.peso

labrador = Perro(25)
print(f'El peso de un perro labrador es {labrador.peso} kilos')
# El peso de un perro labrador es 25 kilos
print(f'El peso promedio de un perro es {Perro.get_peso_promedio()} kilos')
# El peso promedio de un perro es 30 kilos
```

Class Method

1. Bound to the Class
2. It can modify a class state
3. Can Access only Class Variable
4. Used to create factory methods

En este ejemplo, se ha creado una clase `Perro` que tiene un atributo llamado `peso` con un valor por defecto, pero que al crear una instancia lo reasignamos con el valor enviado por parámetro.

Después generamos el método de clase `get_peso_promedio` que retornará el atributo `peso`. Pues bien, si creamos una instancia de `Perro` e imprimimos el valor de `peso`, podemos ver que recuperamos el valor al crear la instancia.

Sin embargo, al acceder al método de clase, obtendremos siempre el valor por defecto asignado a `peso`.

Como puedes ver, la diferencia entre uno y otro es que gracias al método de clase podemos acceder al valor inicial del atributo.

Métodos en Python: instancia, clase y estáticos

Métodos estáticos (staticmethod)

Por último, los métodos estáticos se pueden definir con el decorador `@staticmethod` y no aceptan como parámetro ni la instancia ni la clase. Es por ello por lo que no pueden modificar el estado ni de la clase ni de la instancia. Pero por supuesto pueden aceptar parámetros de entrada.

```
class Clase:
    @staticmethod
    def metodoestatico():
        return "Método estático"
```

```
mi_clase = Clase()
Clase.metodoestatico()
mi_clase.metodoestatico()
```

```
# 'Método estático'
# 'Método estático'
```

Por lo tanto el uso de los métodos estáticos pueden resultar útil para indicar que un método no modificará el estado de la instancia ni de la clase. Es cierto que se podría hacer lo mismo con un método de instancia por ejemplo, pero a veces resulta importante indicar de alguna manera estas peculiaridades, evitando así futuros problemas y malentendidos.

En otras palabras, los métodos estáticos se podrían ver como funciones normales, con la salvedad de que van ligadas a una clase concreta.

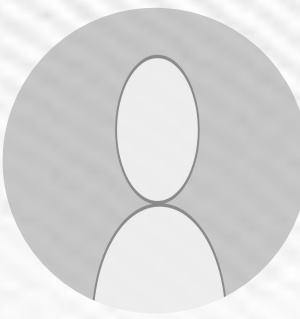


Static Method

1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables

Veamos un ejemplo...

Métodos en Python: instancia, clase y estáticos



Métodos estáticos (staticmethod) cont...

Los métodos estáticos pertenecen a la clase, aunque no dependen ni de la clase ni de una instancia de esta. Por lo tanto, en este caso no necesitaremos ningún parámetro principal como sucede con los métodos normales o los de clase. Este tipo de métodos se utilizan normalmente cuando están ligados de alguna forma a la clase, por lo que no vale la pena tenerlos en un módulo separado de esta.

Para declarar un método estático, solo necesitamos usar el decorador `@staticmethod` y nuestro método estático. Un ejemplo podría ser una clase que se encargase de realizar operaciones matemáticas sencillas:

```
class Math:
```

```
    @staticmethod
    def sumar(num1, num2):
        return num1 + num2
```

```
    @staticmethod
    def restar(num1, num2):
        return num1 - num2
```

```
    @staticmethod
    def multiplicar(num1, num2):
        return num1 * num2
```

```
    @staticmethod
    def dividir(num1, num2):
        return num1 / num2
```

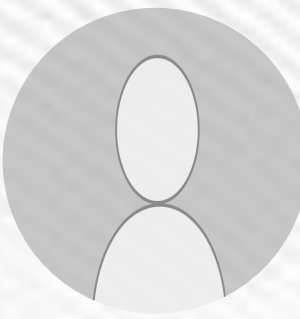
```
print(Math.sumar(5, 7))
print(Math.restar(9, 3))
print(Math.multiplicar(15, 9))
print(Math.dividir(10, 2))
```

“Como se puede ver, en ningún caso necesitamos crear una instancia de la clase y si tuviéramos métodos de otro tipo en la clase, podríamos acceder a estos sin problema.”

Static Method

1. Bound to the Class
2. It can't modify a class or object state
3. Can't Access or modify the Class and Instance Variables

Métodos en Python: instancia, clase y estáticos



En otros posts hemos visto como se pueden crear métodos con def dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia. Pues bien, haciendo uso de los decoradores, es posible crear diferentes tipos de métodos:

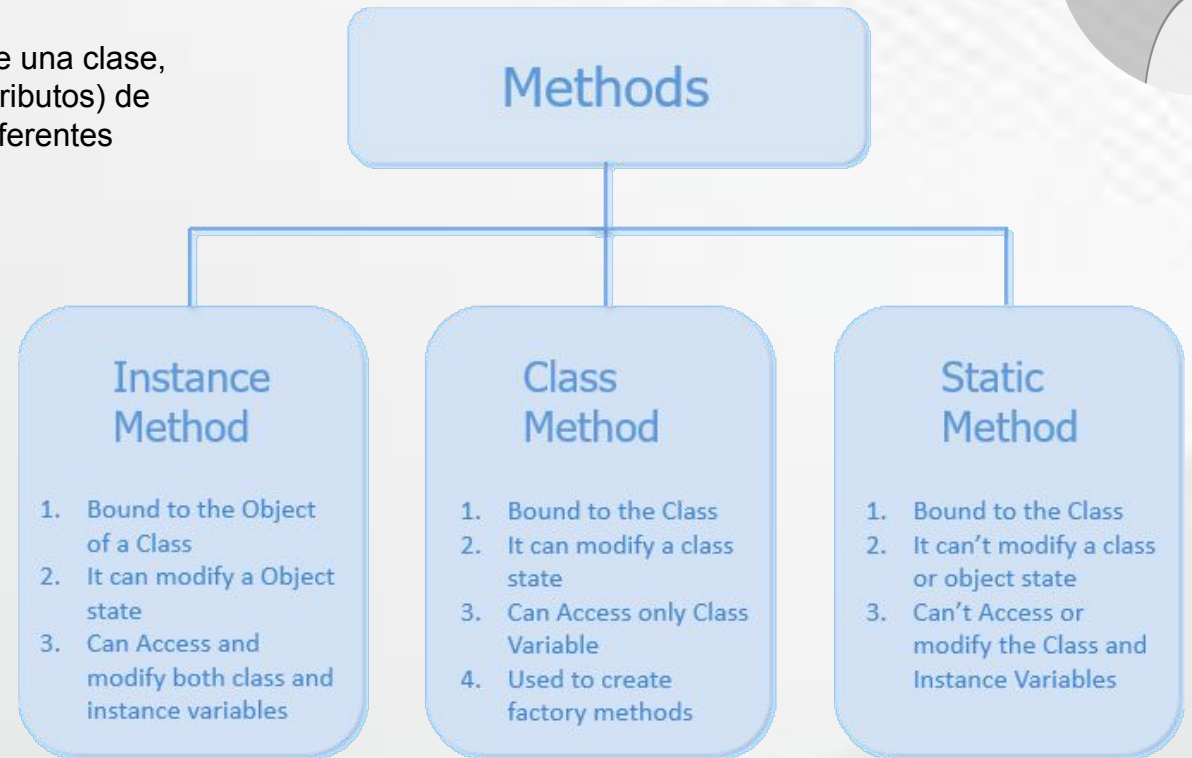
- Los métodos de **instancia** “normales” que ya hemos visto como metodo()
- Métodos de **clase** usando el decorador @classmethod
- Métodos **estáticos** usando el decorador @staticmethod

ass Clase:

```
def metodo(self):  
    return 'Método normal', self
```

```
@classmethod  
def metododeclase(cls):  
    return 'Método de clase', cls
```

```
@staticmethod  
def metodoestatico():  
    return "Método estático"
```




ask ai
Powered By
ChatGPT & Python



13 Py





Language - Agnostic

Language-agnostic programing

La programación o secuencias de comandos independientes del lenguaje es un paradigma de desarrollo de software en el que se elige un lenguaje en particular debido a su idoneidad para una tarea en particular, y no simplemente por el conjunto de habilidades disponibles dentro de un equipo de desarrollo.

Modern Portfolio Designed



THANK YOU

Let's Python Code...