
GeeksHubs
academy _

GeeksHubs academy _

Desarrollamos { talento }

Paradigmas de Programación

¿Qué es un Paradigma de Programación?

Un **paradigma de programación** es un estilo de desarrollo de programas. Es decir, un modelo para resolver problemas computacionales.

Los lenguajes de programación se encuadran en uno o varios paradigmas a la vez.

De lejos, el más popular y usado es el paradigma de ***programación orientado a objetos.***

GeeksHubs academy _

Desarrollamos { talento }

Programación Orientada a Objetos

¿Qué es la Programación Orientada a Objetos?

La **Programación Orientada a Objetos** (POO u OOP según sus siglas en inglés) es un *paradigma de programación*. Es una forma especial de programar, ya que pasamos a usar objetos como unidad mínimas de código y las relaciones que tienen entre sí.

Los Objetos son representaciones de elementos del mundo real. Por ejemplo, una persona, un coche, un martillo. A estos objetos le programamos unas propiedades y acciones necesarios para resolver los problemas de nuestra aplicación.

Example of an Object

Properties	Methods
Colour	Start, Stop
Transmission Type	Accelerate
Max Speed	Change Transmission

Ventajas de POO

1. **Descomponer un problema:** la programación orientada a objetos permite la división del problema en partes pequeñas.

2. **Programas más fáciles de mantener:** Sumado a su buena legibilidad, el concepto de los objetos favorece el mantenimiento de un programa.

3. **Orden y Legibilidad:** Las clases y objetos son fáciles de identificar a primer golpe de vista y gracias a su sintaxis es muy fácil generar código que de otra forma sería muy complejo.

Los Objetos en Python

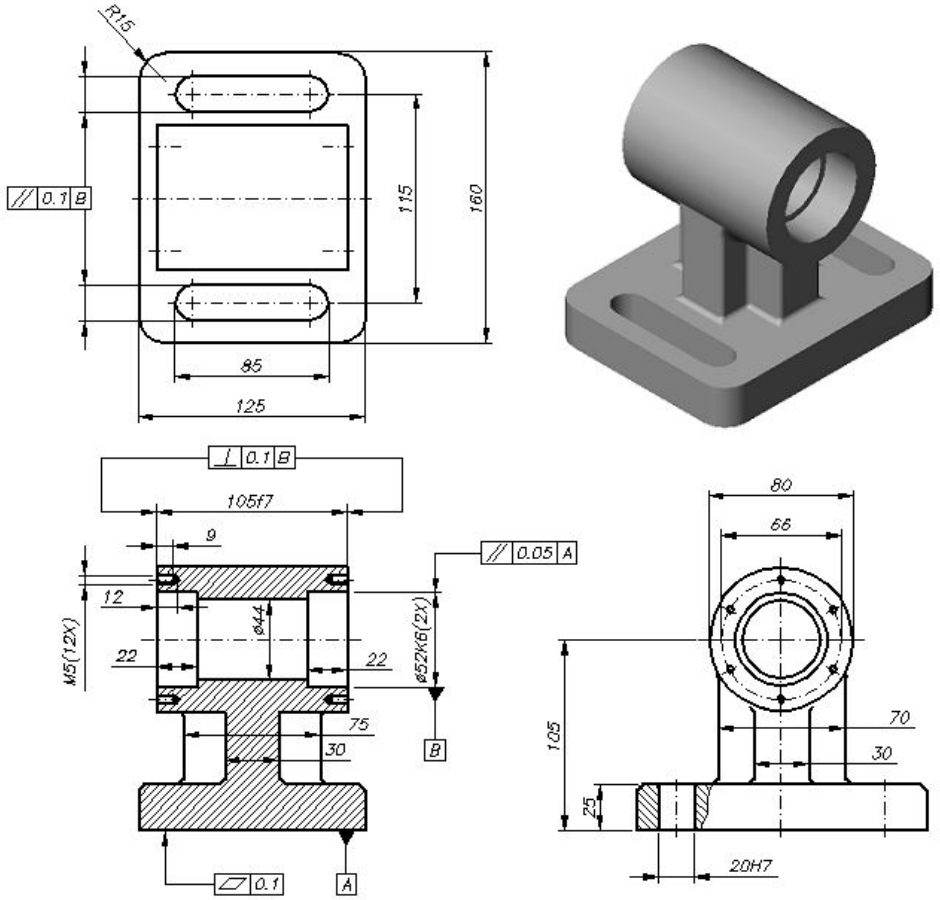
La ***Programación Orientada a Objetos***, se puede diferenciar en dos “fases” a la hora de aplicarlo en nuestro algoritmos:

Fase 1: La declaración de la Clase.
Fase 2: La instanciación del Objeto.

Utilizando una definición no-técnica:

La **clase** es un plano que te permite crear el Objeto

La **instanciación** es la acción que simboliza la creación de un objeto a partir de un “plano” (clase).



Anatomía de una Clase de Python

Entonces, lo primero que necesitamos es declarar la clase. Para ello, tenemos a **class** una palabra reservada en Python, y se usa como una estructura.

Una clase se divide en dos partes:

Las Propiedades: Son las variables encapsuladas dentro de la clase/objeto. Son las características o propiedades del objeto real.

Los Métodos: Son las funciones encapsuladas dentro de la clase/objeto. Son las acciones que pueden hacer el objeto real.

```
class Persona():
    # Propiedades
    Nombre = ''
    Edad = 0

    # Métodos
    def Saludar(self):
        print(f'Hola, soy {self.Nombre}')


Jorge = Persona()

Jorge.Nombre = 'Manuel S.'
Jorge.Saludar()
```

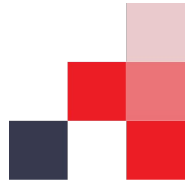
La palabra reservada self

Las propiedades de una clase no tiene mucho misterio, son variables internas de la clase, pero los métodos tiene algo diferente.

Como primer parametro del metodo, usamos la palabra reservada ***self***. Este self nos da acceso a las propiedades y métodos de la clase.

En el ejemplo vemos cómo usamos la propiedad Nombre desde el **self**.

```
# Métodos
def Saludar(self):
    print(f'Hola, soy {self.Nombre}')
```



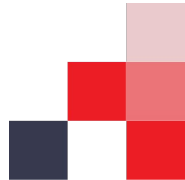
Generar el Objeto desde una Clase

Como hemos dicho anteriormente, la clase es solo “el plano” que nos permite crear el objeto en cuestión.

En Python, tenemos que usar el operador de asignación (=) para poder instanciar (construir) un objeto a esa variable.

Luego podemos ver como esa variable tiene las propiedades y métodos de dicha clase.

```
class Persona():
    # Propiedades
    Nombre = ''
    Edad = 0

    # Métodos
    def Saludar(self):
        print(f'Hola, soy {self.Nombre}')


Jorge = Persona()

Jorge.Nombre = 'Manuel S.'
Jorge.Saludar()
```



Constructor

Vale, ya tenemos una clase que nos hace de plano para crear objetos, pero tal y como está ahora es demasiado estático.

¿Y si pudiéramos dar valores por defecto a ese objeto en su creación?

Para eso tenemos los constructores, en Python se representa con la función **__init__** y usamos los self para apuntar a las propiedades internas de la clase.

Con este constructor podremos dar valores por defectos o invocar una función desde su asignación a la variable.

```
class Persona():
    # Propiedades
    Nombre = ''
    Edad = 0

    def __init__(self, nombre, edad):
        self.Nombre = nombre
        self.Edad = edad

    # Métodos
    def Saludar(self):
        print(f'Hola, soy {self.Nombre}')


Jorge = Persona('Jorge', 25)
Jorge.Saludar()
```

Kata I

KATA I: Evaluación de Alumnos

Vamos a desarrollar un programa donde nos facilite la evaluación de los alumnos. Para ello, vamos a usar POO.

Alumno

+ Nombre
+ Apellidos
+ DNI
+ Edad
+ Nota

- Saludar
- Añadir Nota (entre 0 - 10)
- Cumplir años (+1 a la edad)

```python
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```

Composición

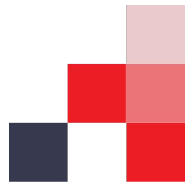
Composición

La **Composición** es cuando usamos un objeto como propiedad de otro objeto.

Si recordamos, todo en Python es un objeto, y los primeros tipos de datos (int, str y bool) fueron los primeros objetos que vimos.

Pues la composición es cuando usamos un objeto creado por nosotros para que sea parte de otro objeto en forma de propiedad.

```
class Team():
    players = []

    def add_player(self, player):
        self.players.append(player)

    def show_player(self):
        for player in self.players:
            print(player.name)

class Player():
    # Propiedades
    name = ''
    yearOld = 0
    rol = ''

    def __init__(self, name, yearOld, rol):
        self.name = name
        self.yearOld = yearOld
        self.rol = rol
```



Kata II

KATA II: Asignatura

Usando la Kata anterior, vamos a crear una clase llamada Asignatura, para modularizar nuestro programa

Asignatura
- \+ Nombre
- \+ Nota
- ● Añadir Nota

Alumno (Modificación)
- \+ Asignaturas
- ● Añadir Asignaturas
- ● Eliminar Asignaturas

```
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```

Abstracción

Abstracción

La Abstracción es el proceso el cual **dividimos una gran funcionalidad** o programa, en otras funciones más pequeñas.

Por ejemplo, si queremos una calculadora, que te haga todas las operaciones en el constructor. Pues generamos una función por cada operación y la invocamos desde el constructor.

De esa forma el código queda más legible y organizado.

```
class Calculator():

    def __init__(self, num1, num2):
        self.add(num1, num2)
        self.subs(num1, num2)
        self.multiply(num1, num2)
        self.divide(num1, num2)

    def add(self, num1, num2):
        print(num1 + num2)

    def subs(self, num1, num2):
        print(num1 - num2)

    def multiply(self, num1, num2):
        print(num1 * num2)

    def divide(self, num1, num2):
        print(num1 / num2)
```

Kata III

KATA III: Clase

Usando la Kata anterior, vamos a crear una clase llamada “Clase” donde crearemos la estructura de una clase con su profesor, alumnos y asignaturas con las mejores prácticas posibles.

Clase
- \+ profesor
- \+ alumnos
- \+ asignaturas
- ● añadir/borrar alumnos
- ● añadir/borrar asignatura

```python
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```

Encapsulación

Encapsulación

La encapsulación consiste en controlar el acceso a las propiedades y métodos internos de la clase cuando esta sea un objeto.

Python, es un lenguaje muy flexible y libre, pero eso no es siempre una ventaja. Esa libertad puede hacer que nuestro programa no funcione como debería o producir bugs bastante peligrosos.

```
class BankAcount:
    incumbent = None
    balance = 0

    def __init__(self, incumbent):
        self.incumbent = incumbent

    def add_balance(self, money):
        self.balance += money
        print(f'Balance: {self.balance}')
```

Accesibilidad

En la encapsulación tiene dos tipos de accesibilidad en las propiedades y métodos:

Públicos: Es la forma que esta por defecto y tal como hemos estado trabajando. No tienen ninguna restricción a la hora de usar propiedades o métodos en nuestro objeto.

Privados: Denegamos el acceso de la propiedad o método al exterior. La declaración privada se realiza con doble guión bajo al inicio (__). Para controlar ese acceso tenemos los getters y setters.

```
class BankAcount:
    incumbent = None
    __balance = 0

    def __init__(self, incumbent):
        self.__change_incumbent(incumbent)

    def __change_incumbent(self, new_incumbent):
        self.incumbent = new_incumbent

    def add_balance(self, money):
        self.__balance += money
        print(f'Balance: {self.__balance}')
```

Getter / Setter

Los getters y setters son dos funciones que nos permite acceder a una propiedad privada en concreto.

Recordemos que las propiedades privadas solo es se puede acceder desde la misma clase.

Al encapsular el acceso en estas funciones, podemos complementar la lectura / escritura de las variables con más código.

Por ejemplo, validar la asignación o añadir el nuevo dato a la lista de propiedad.

```
class BankAcount:
    incumbent = None
    __balance = 0

    def __init__(self, incumbent):
        self.__change_incumbent(incumbent)

    def get_balance(self):
        return self.__balance

    def set_balance(self, balance):
        self.__balance = balance

    def __change_incumbent(self, new_incumbent):
        self.incumbent = new_incumbent

    def add_balance(self, money):
        self.__balance += money
        print(f'Balance: {self.__balance}')
```

Getter / Setter Avanzado

Gracias al uso de decoradores, podemos definir el getter con el decorador @property y el setter con @propiedad.setter

De esta forma, vemos a simple vista dónde está los getter y setter en nuestra clase y el uso del getter es más intuitivo.

```
class BankAcount:
    incumbent = None
    __balance = 0

    def __init__(self, incumbent):
        self.__change_incumbent(incumbent)

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def set_balance(self, balance):
        self.__balance = balance

    def __change_incumbent(self, new_incumbent):
        self.incumbent = new_incumbent

    def add_balance(self, money):
        self.__balance += money
        print(f'Balance: {self.__balance}')
```

Kata IV

KATA IV: Encapsulando

Usando la Kata anterior, vamos poner en privado los siguientes métodos y propiedades:

Clase
+ id (solo getter)

Asignatura
+ id (solo getter)
+ nota(getter/setter)

Alumno
+ dni(solo getter)

```
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```

Herencia

Herencia Simple

La herencia es un mecanismo que sirve para crear clases nuevas a partir de otras clases preexistentes.

Se heredan las propiedades y métodos de las clases padres junto a las de la clase hija. De esta forma dividimos el código y reutilizamos las partes que sean iguales de otra clase.

Para ello, declaramos la clase padre en los paréntesis de la clase hija.

```
class Point2D():
    x = 0
    y = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def coord2D(self):
        return [self.x, self.y]

class Point3D(Point2D):
    z = 0

    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def coord3D(self):
        p2 = super().coord2D()
        p2.append(self.z)
        return p2
```

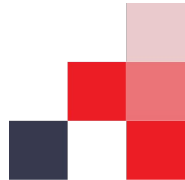
Super()

Para que la herencia esté completa, necesitamos añadir en nuestro constructor una palabra clave llamada **super().**

Esta palabra reservada podrá llamar al constructor del padre e invocarlo en el constructor del hijo.

Después podemos acceder a las propiedades y métodos del padre desde el self como si fuera de la misma clase hija.

```
class Point2D():
    x = 0
    y = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def coord2D(self):
        return [self.x, self.y]

class Point3D(Point2D):
    z = 0

    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def coord3D(self):
        p2 = super().coord2D()
        p2.append(self.z)
        return p2
```



Kata V

KATA V: Usuario / Profesor

Usando la Kata anterior, genera la clase Usuario y usa la herencia para que desde Usuario, se cree Alumno y Profesor, reutilizando todo lo que veas oportuno

```
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```

Herencia Múltiple

Herencia Múltiple

La Herencia Múltiple es una característica muy poco habitual en los POO, y es cuando una clase puede heredar las propiedades y atributos de varias clases padres a la vez.

La forma de funcionar es la misma, separando por comas cada padre en los paréntesis, y en vez de usar super(), podemos usar el nombre de la clase padre.

```
class Colour():
    black = 'BLACK'
    red = 'RED'
    green = 'GREEN'
    yellow = 'YELLOW'

    def __init__(self):
        pass

class Point3D(Point2D, Colour):
    z = 0

    def __init__(self, x, y, z):
        Point2D.__init__(self, x, y)
        Colour.__init__(self)
        self.z = z

    def coord3D(self):
        return [self.x, self.y, self.z, self.black]

p3 = Point3D(1,2,3)
print(p3.coord3D())
```

Sobrecarga de Métodos

Otra característica de la Herencia, es la sobrecarga o sobrescritura de métodos.

¿Qué pasa si un padre e hijo comparten el mismo nombre en un método? Pues el método que queda es el método del hijo, es decir, estamos sobreescribiendo el método del padre con el del hijo.

```
class Point2D():
  x = 0
  y = 0

  def __init__(self, x, y):
      self.x = x
      self.y = y
   def coord(self):
      return [self.x, self.y]

class Point3D(Point2D):
  z = 0

  def __init__(self, x, y, z):
      super().__init__(x, y)
      self.z = z

  def coord(self):
      p2 = super().coord()
      p2.append(self.z)
      return p2
```

GeeksHubs academy _

Desarrollamos { talento }

Kata VI

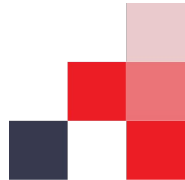
KATA VI: Animales

Vamos a generar un programa de clasificación de animales. Para ello partiremos de una clase base llamada animal, que luego se usará para crear León y Perro.

Las propiedades serán, especie, peso, altura, y métodos como ***comer, cazar o dormir***.

Perro como es un animal doméstico, también heredada de una clase Mascota, donde tendrá acceso a un nombre y trucos como sentarse y tumbarse

```
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```



Métodos Mágicos

Métodos Mágicos

Los métodos mágicos son unos métodos que están por defecto en todas las clases de Python. Que tienen su comportamiento por defecto, sino lo llamamos.

El método __init__ es uno de ellos, que le llamamos constructor. Existen muchos de estos métodos que nos permite modificar el comportamiento de estas clases cuando se someten a los operadores, como la suma, resta, comparaciones, toString.... etc

```
class Person():
    name = ''
    yearOld = 0

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

    def __add__(self, person):
        family = Family()
        family.add(self)
        family.add(person)
        return family

class Family():
    membrers = []

    def new_memb(self, person):
        self.membrers.append(person)

    def __sub__(self, person):
        self.membrers.remove(person)
        return self
```

Ejemplos de Métodos Mágicos

__add__ = Modificamos el operador +
__sub__ = Modificamos el operador -
__str__ = Creamos un cast de objetos a string

Más información:

https://www.tutorialsteacher.com/python/magic-methods-in-python

```python
class Person():
    name = ''
    yearOld = 0

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

    def __add__(self, person):
        family = Family()
        family.add(self)
        family.add(person)
        return family
```

Kata VII

KATA VIII: Familia

Usando el ejemplo anterior, usa los métodos mágicos para que si a una familia le resta miembros y queda solo 1, pasa automáticamente a ser una persona.

Y si queda a 0 o menos, devolverá un None

```
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```

Polimorfismo

Polimorfismo

El **polimorfismo** es la capacidad de realizar una acción en un objeto independientemente de su tipo.

Esto es posible si varias clases comparte el mismo padre, por lo cual nos aseguramos una base de propiedades y métodos.

```
class Shape():
    slides = 0
    area = 0
    perimeter = 0

    def __init__(self, slides):
        self.slides = slides

    def perimeter_calc(self, length = 0):
        self.perimeter = self.slides * length
        return self.perimeter

    def area_calc(self, ap = 0):
        self.area = (self.perimeter * ap) / 2
        return self.area
```

Kata VIII

KATA VIII: Figuras

A partir del ejemplo de figuras, crea una clase para Triángulo y Cuadrado, sobre escribiendo sus métodos de perímetro e área.

```
class Alumno():
    # Propiedades


    def __init__(self):
        pass

    # Métodos
```

GeeksHubs
academy _

/

Community Lead - Manuel S. Lemos
@manuels_lemos

www.geekshubsacademy.com
@geekshubs