KIBERNUM
IT Academy

Módulo 2 – Fundamentos de Programación Python

Orientación a Objetos en Python

Objetivos

- Reconocer los conceptos fundamentales de la programación orientada a objetos.
- Utilizar librerías con orientación a objetos para resolver un problema.

KIBERNUM

Contenido

1. Características de la POO.
2. Clases y objetos.
3. Implementación en Python.
4. Accesadores y mutadores.
5. Herencia y polimorfismos.

KIBERNUM

Características de la POO

Programación Orientada a Objetos

La programación orientada a objetos (POO) es una forma de escribir programas de computadora donde los datos y las operaciones que se realizan con esos datos están organizados en unidades llamadas "objetos". Cada objeto tiene características (llamadas atributos) y comportamientos (llamados métodos).

La POO nos permite organizar nuestro código de manera más clara y modular, lo que hace que sea más fácil de entender, mantener y reutilizar. Además, facilita la colaboración en equipos de desarrollo, ya que cada clase define un tipo de objeto concreto y sus funcionalidades asociadas.

```python
class Saludo:
    def __init__(self, mensaje):
        self.mensaje = mensaje

    def mostrar_saludo(self):
        print(self.mensaje)

# Crear una instancia de la clase Saludo
saludo = Saludo("¡Hola Mundo!")
```

KIBERNUM

Programación Orientada a Objetos

Marca

Color Modelo

Arrancar Acelerar

Detenerse

Piensa en un objeto como algo que puedes ver y manipular en el mundo real, como un coche. El coche tendría **características** como el color, la marca y el modelo, y **comportamientos** como arrancar, detenerse y acelerar.

En la programación orientada a objetos, creamos clases para definir diferentes tipos de objetos. Por ejemplo, podríamos tener una clase llamada "Coche" que define cómo se crea y cómo funciona un coche en nuestro programa. Luego, podemos crear múltiples objetos basados en esa clase, cada uno con sus propios datos y comportamientos, como diferentes coches en la vida real.

Programación Orientada a Objetos

El siguiente programa calcula el área de un círculo. Nótese que en la imagen de la derecha se ha utilizado la POO.

Programación Estructurada

```python
import math

def calcular_area_circulo(radio):
    return math.pi * radio**2

radio = 5
area = calcular_area_circulo(radio)
print("El área del círculo es:", area)
```

Programación con Orientación a Objetos

```python
import math

class Circulo:
    def __init__(self, radio):
        self.radio = radio

    def calcular_area(self):
        return math.pi * self.radio**2

radio = 5
mi_circulo = Circulo(radio)
area = mi_circulo.calcular_area()
print("El área del círculo es:", area)
```

KIBERNUM

¿Por qué utilizar POO?

La POO permite crear software seguro y fiable. Muchas librerías y módulos de Python utilizan este paradigma para construir su código base. A continuación, algunas razones de por qué es conveniente utilizar la POO.

- La POO te ayuda a organizar tu código de una manera más fácil de entender, como si estuvieras ordenando tus juguetes en diferentes cajas.
- La POO facilita la reutilización del código, Es como tener un conjunto de bloques de construcción. Puedes usar los mismos bloques una y otra vez para crear diferentes cosas.
- La POO te permite hacer cambios en tu código más fácilmente, como si estuvieras cambiando las piezas de un rompecabezas sin afectar las otras piezas.
- La POO mejora el análisis de cualquier situación, ya que los objetos son abstracciones de la realidad.
- La POO hace que tu código sea más fácil de leer y entender para otras personas, como si estuvieras escribiendo una historia con personajes bien definidos y acciones claras
- La POO te permite dividir grandes problemas en partes más pequeñas y manejables, como si estuvieras dividiendo una tarea difícil en pasos más simples.

Programación Estructurada v/s POO

POO	Programación Estructurada
Se centra en objetos con atributos y comportamientos	Se centra en funciones y procedimientos
Pequeños trozos de código reutilizados en muchos lugares	Trozos grandes de código con poca reutilización
Muy poco código repetido	Es común encontrar código repetido
Más fácil de depurar y encontrar errores	Más difícil de depurar y encontrar errores
Utilizado en grandes proyectos	Utilizado en programas sencillos
Más fácil de entender un código escrito por otra persona	Más complicado de entender un código escrito por otra persona

Clases y Objetos

Clases y Objetos

Clases

Imagina que una clase es como un molde para hacer galletas de jengibre. El molde define cómo tiene que ser cada galleta. En programación, una clase define cómo debe ser un objeto.

Una clase tiene características (llamadas atributos) y comportamientos (llamados métodos).

En este ejemplo, vamos a definir atributos tales como el color de la decoración.

Clases y Objetos

Objetos

Ahora, piensa en un objeto como una galleta hecha con el molde. Cada galleta hecha con el mismo molde tiene las mismas características, pero puede tener diferentes valores para esas características.

En programación, un objeto es una instancia de una clase, que se crea en memoria y tiene valores definidos en sus atributos. En este caso, ambas galletas fueron hechas con el mismo molde, pero decoradas de forma distinta cada una de ellas.

Clases y Objetos

El molde o clase, es lo que codificamos en nuestro entorno de desarrollo (notebook jupyter, script Python). Mientras que cuando ejecutamos el programa, se ejecutan las instrucciones y los objetos son creados en memoria. Cuando finaliza la ejecución, se vacía la memoria y los objetos son destruidos.

Codificación del Programa

Ejecución del Programa

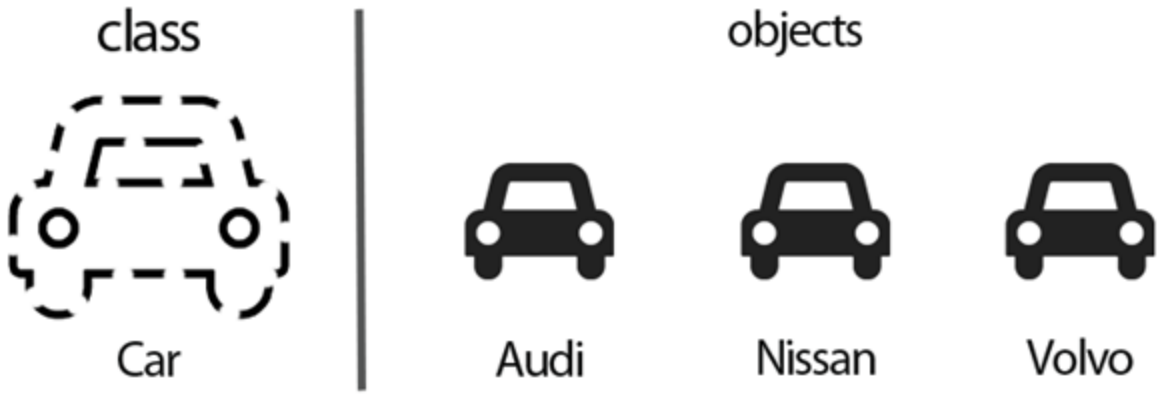
KIBERNUM

Abstracción de Características

En términos simples, la abstracción se refiere a la capacidad **de enfocarse en los aspectos importantes de un objeto o idea**, mientras se omiten los detalles menos relevantes o complicados.

Por ejemplo, podemos crear una clase "Coche" con atributos como color, marca y modelo, y métodos como "encender" y "acelerar“.

Esto nos permite trabajar con coches en nuestro código sin preocuparnos por los detalles internos de cómo funcionan esos métodos o cómo se implementan realmente.



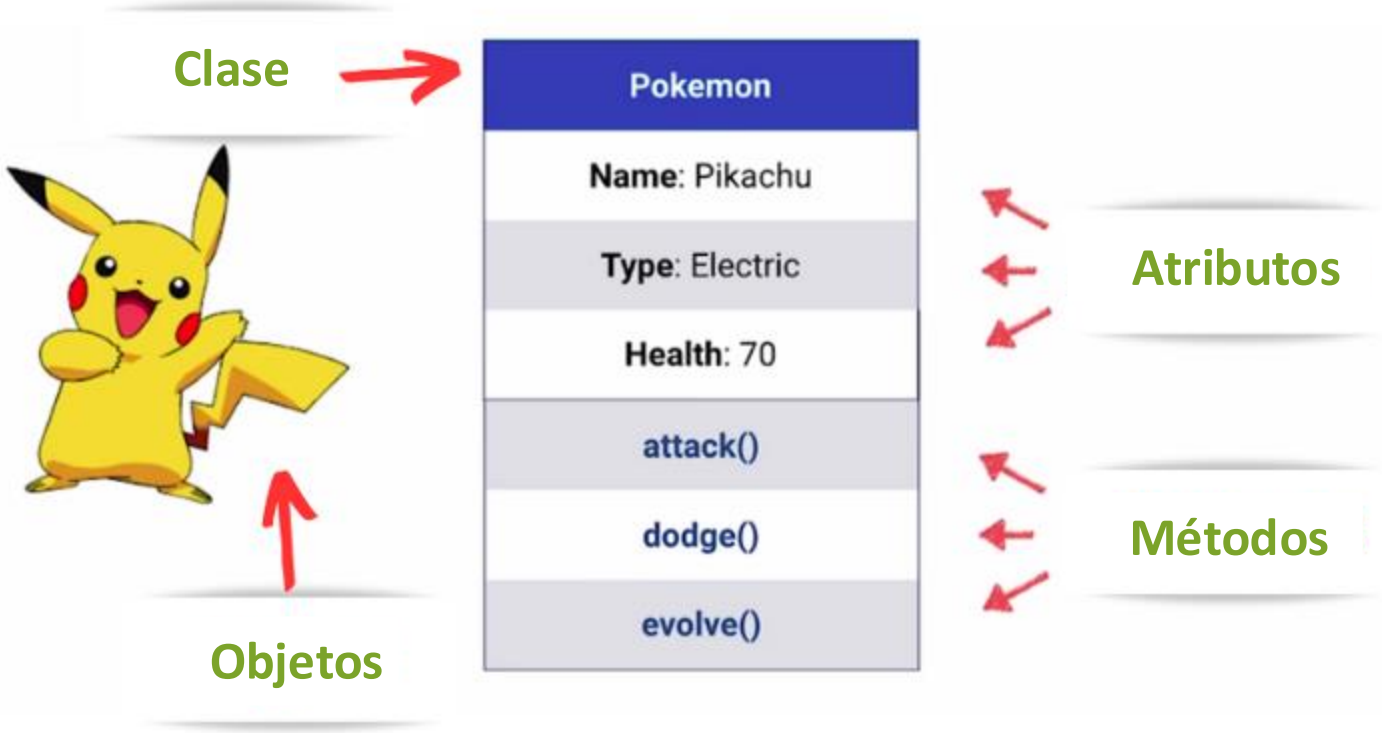
KIBERNUM

Atributos y Métodos

Atributos

Los atributos son como características o propiedades de un objeto. Puedes pensar en ellos como cosas que describen al objeto. Son como etiquetas que ayudan a definir qué es el objeto y cómo es.

En este ejemplo, se definen los siguientes atributos para definir a un Pokemon específico: Nombre, Tipo de Poder, y Salud.

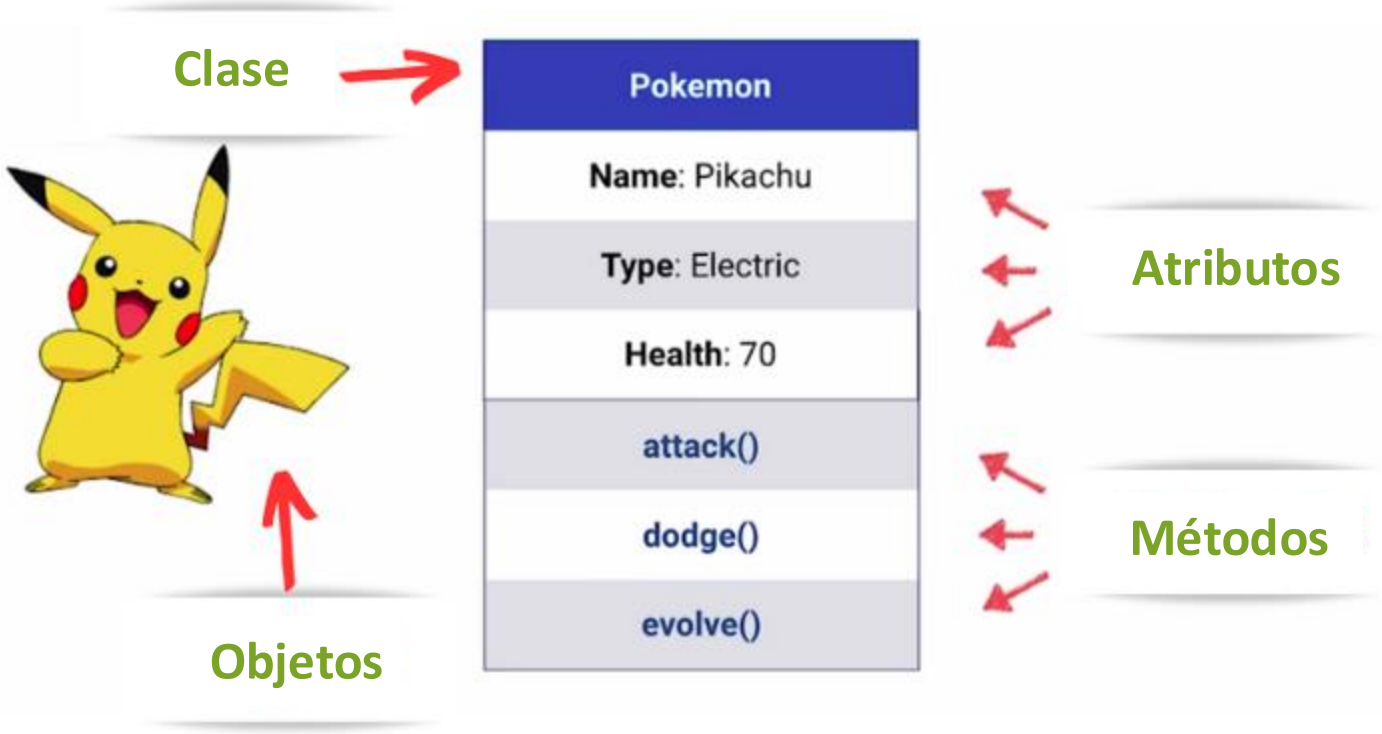


Atributos y Métodos

Métodos

Los métodos son como las acciones que puede realizar un objeto. Son como cosas que el objeto puede hacer. Siguiendo con el ejemplo, se han definido los métodos Atacar, Esquivar, y Evolucionar, para el caso de la clase Pókemon.

Los métodos definen el comportamiento del objeto y cómo interactúa con su entorno.



Estado de un objeto

Durante la ejecución del programa, los objetos se crearán en la memoria con valores concretos en sus atributos. A esto le llamamos **Estado de un Objeto**.

Durante la ejecución del programa, los objetos **pueden ir cambiando su estado** de acuerdo con lo que se haya definido en el molde en sus métodos.

Los métodos definen el comportamiento que tendrán los objetos durante la ejecución. Dicho **comportamiento es el que tiene la capacidad de mutar el estado del objeto**.

Nombre: Pikachu
Poder: Eléctrico
Salud: 100

Nombre: Bulbasaur
Poder: Embestida
Salud: 100

Nombre: Charizard
Poder: Fuego
Salud: 100

KIBERNUM

Implementación en Python

La clase más simple

- Las clases son definidas utilizando la palabra reservada **class**. En este ejemplo, se ha implementado una clase vacía, la más simple que podamos crear.

- Al definir el nombre de una clase, se recomienda utilizar la convención de nombre de tipo **camel**, similar a Java. Es decir, **NombreDeLaClase**, en vez de **nombre_de_la_clase**.

```
class Pokemon:
    pass
```

```
pok1 = Pokemon()
```

```
print(type(Pokemon))
print(type(pok1))
```

```
<class 'type'>
<class '__main__.Pokemon'>
```

KIBERNUM

Definiendo atributos de clase

En el siguiente ejemplo se definen 3 atributos de clase, con valores por defecto.

```
class Pokemon:
    name = 'Pikachu'
    ot = 'Electric'
    health = 70
```

```
pok1 = Pokemon()

print(pok1.name)
print(pok1.ot)
```

```
Pikachu
Electric
```

Creando un atributo en una instancia

Un objeto creado a partir de una clase (instancia), permite la definición de nuevos atributos los que tendrán validez sólo para el objeto en donde se define.

```
# sea la siguiente clase con un atributo definido
class Pokemon:
    name = 'Pikachu'
```

```
# definimos el atributo color en el objeto pok1
pok1 = Pokemon()
pok1.color = 'Amarillo'
print(pok1.color)
```

```
Amarillo
```

Nótese que al objeto pok1 se le ha definido un atributo color con valor amarillo.

```
# el objeto pok2 no conoce el atributo color
pok2 = Pokemon()
print(pok2.color)
```

```
---------------------------------------------------------------
---------------
AttributeError                            Traceback (most re
cent call last)
<ipython-input-35-bbe8323c6960> in <module>
      1 pok2 = Pokemon()
----> 2 print(pok2.color)

AttributeError: 'Pokemon' object has no attribute 'color'
```

Nótese que el objeto pok2 no conoce este atributo, pues está definido sólo en el objeto pok1.

KIBERNUM

Creando métodos en una clase

En una clase, se pueden definir métodos que serán compartidos eventualmente por cada instancia de la clase (objeto). Dentro de un método de la clase, podemos referenciar a la instancia con la palabra reservada **self**.

```
class Pokemon:
    name = 'Pikachu'
    ot = 'Electric'
    health = 70

    # creamos un método que accede un atributo de clase
    def firePower(self):
        return self.health / 2
```

```
pok1 = Pokemon()

# invocamos el metodo creado
print(pok1.firePower())

# otra forma de invocar el metodo
print(Pokemon.firePower(pok1))
```

```
35.0
35.0
```

KIBERNUM

Inicializando una clase

A continuación, se inicializa una clase con valores por defecto. Nótese que en este caso se han creado los atributos de la clase directamente en el método inicializador. En otros lenguajes, el método inicializador es llamado **constructor**.

```
class Pokemon:

    # Constructor sin parámetros
    def __init__(self):
        # atributos de la clase
        self.name = 'Pikachu'
        self.ot = 'Electric'
        self.health = 70
```

```
pok2 = Pokemon()
print(pok2.name)
print(pok2.health)
```

```
Pikachu
70
```

Acá hemos creado un objeto sin parámetros en el constructor.

```
pok2 = Pokemon(name='Bulbasor')
print(pok2.name)
print(pok2.health)
```

```
Bulbasor
70
```

En este caso sólo hemos pasado el parámetro name al momento de crear el objeto

Inicializando una clase con parámetros

La inicialización de una clase puede llevar parámetros, los cuales podrían tener valores por defecto, de forma análoga a como se define la firma de una función.

```
class Pokemon:

    # Constructor con parámetros con valores por defecto
    def __init__(self, name='Pikachu',
                 object_type='Electric', health=70):
        self.name = name
        self.ot = object_type
        self.health = health
```

```
pok2 = Pokemon()
print(pok2.name)
print(pok2.health)
```

```
Pikachu
70
```

En este caso, hemos inicializado la clase sin utilizar parámetros, por lo tanto, el objeto toma los valores de atributos de la clase

```
pok2 = Pokemon(name='Bulbasaur')
print(pok2.name)
print(pok2.health)
```

```
Bulbasaur
70
```

En este caso, se ha proporcionado un parámetro en la inicialización de la clase

Modificadores de Acceso

En Python no existen modificadores de acceso reales (como private o protected en Java). Lo que se usa son convenciones de nombres:

Público (sin guiones bajos): Accesible desde cualquier parte.

```
self.name = "Pokemon Warrior"    # público
```

Protegido (_atributo): el prefijo con un solo guion bajo (_) indica convención: el atributo debería usarse solo dentro de la clase subclases, pero no está restringido.

```
self._ot = "Electric"    # protegido por convención
```

Privado (__atributo): el doble guion bajo (__) activa el mecanismo de name mangling. Python cambia internamente el nombre del atributo para incluir el nombre de la clase (_Clase__atributo). Esto evita accesos accidentales, pero no impide acceder si se conoce el nombre transformado.

```
self.__health = 100     # privado por convención
```

Modificadores de Acceso

- En el caso de los métodos, funciona de forma análoga.

```
class Pokemon:
    # variable privada
    __health = 0

    def __init__(self,health):
        self.__health = health

    # metodo privado
    def __reset_health():
        __health = 100
```

```
pok1 = Pokemon(50)
print(pok1.reset_health())
```

```
---------------------------------------------------------
----------------------
AttributeError                            Traceback
(most recent call last)
<ipython-input-135-2899b43376bc> in <module>
      1 pok1 = Pokemon(50)
----> 2 print(pok1.reset_health())

AttributeError: 'Pokemon' object has no attribute 'r
eset_health'
```

KIBERNUM

Accesadores y Mutadores

Accesadores y Mutadores

En la programación orientada a objetos, los métodos accesadores y mutadores permiten que atributos definidos, como privados y protegidos, puedan ser obtenidos o modificados por otros objetos, de forma controlada. A continuación, se muestra un ejemplo, utilizando la nomenclatura de **getters** y **setters**.

```
class Pokemon:

    def __init__(self, health):
        self.__health = health

    # accesador health
    def get_health(self):
        return self.__health

    # mutador health
    def set_health(self, health):
        if 0 <= health <= 100:
            self.__health = health
        else:
            # se podria mejorar con raise Error
            print('Healt debe estar entre 0 y 100')
```

```
pok1 = Pokemon(150)
# utilizamos accesador
print('Health es', pok1.get_health())

# utilizamos mutador
pok1.set_health(80)
print('Health es', pok1.get_health())
```

```
Health es 150
Health es 80
```

En este caso, al accesar directamente la variable se levanta un error, puesto que está definida como privada.

```
# utilizamos la variable directa
print('Health es' , pok1.health)
```

```
---------------------------------------------------------
----------------------
AttributeError                            Traceback
(most recent call last)
<ipython-input-32-1cc6cc4f12a1> in <module>
      1 # utilizamos la variable directa
----> 2 print('Health es' , pok1.health)

AttributeError: 'Pokemon' object has no attribute 'h
ealth'
```

KIBERNUM

Definiendo atributos de clase

- Un Decorador en Python, es una función que toma otra función como argumento y que retorna otra función. Los decoradores son extremadamente útiles al momento de extender una función sin la necesidad de modificar la función original. No entraremos en mayor detalle respecto a los decoradores, pero revisaremos algunos decoradores convenientes al momento de definir una clase.

- El decorador Property ayuda a definir clases con métodos accesadores y mutadores (getters y setters).

A continuación, un ejemplo:

```
class Pokemon:
    def __init__(self, health):
        self._health = health

    @property
    def health(self):
        return self._health

    @health.setter
    def health(self, health):
        if 0 <= health <= 100:
            self._health = health
        else:
            # se podria mejorar con raise Error
            print('Healt debe estar entre 0 y 100')
```

```
pok1 = Pokemon(50)
print('Health es', pok1.health)
# asignamos un valor invalido
pok1.health=120
print('Health es', pok1.health)
# asignamos un valor valido
pok1.health=90
print('Health es', pok1.health)
```

```
Health es 50
Health debe estar entre 0 y 100
Health es 50
Health es 90
```

KIBERNUM

Definiendo atributos de clase

- Un Decorador en Python, es una función que toma otra función como argumento y que retorna otra función. Los decoradores son extremadamente útiles al momento de extender una función sin la necesidad de modificar la función original. No entraremos en mayor detalle respecto a los decoradores, pero revisaremos algunos decoradores convenientes al momento de definir una clase.

- El decorador Property ayuda a definir clases con métodos accesadores y mutadores (getters y setters).

A continuación, un ejemplo:

```python
class Pokemon:
    def __init__(self, health):
        self._health = health

    @property
    def health(self):
        return self._health

    @health.setter
    def health(self, health):
        if 0 <= health <= 100:
            self._health = health
        else:
            # se podria mejorar con raise Error
            print('Healt debe estar entre 0 y 100')
```

```python
pok1 = Pokemon(50)
print('Health es', pok1.health)
# asignamos un valor invalido
pok1.health=120
print('Health es', pok1.health)
# asignamos un valor valido
pok1.health=90
print('Health es', pok1.health)
```

```
Health es 50
Healt debe estar entre 0 y 100
Health es 50
Health es 90
```

KIBERNUM

Herencia y Polimorfismo

KIBERNUM

Herencia y Polimorfismo

Uno de los objetivos de la orientación a objetos es la reutilización del código. La herencia, permite extender una funcionalidad existente en una clase definiendo una nueva clase que hereda la funcionalidad de una clase existente.

KIBERNUM

Herencia

Cuando hablamos de herencia, estamos definiendo una relación entre objetos del tipo “Es-un”.

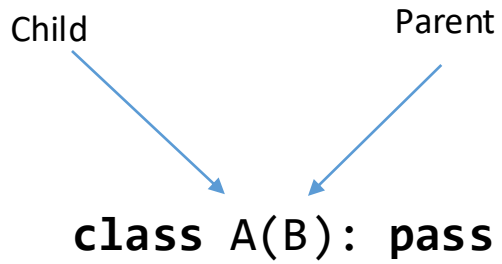
```python
# definimos una clase pokemon
class Pokemon:
    health = 100
    pass

# definimos una clase ElectricPokemon
class ElectricPokemon(Pokemon): # Es-un Pokemon
    pass

# definamos una clase FirePokemon
class FirePokemon(Pokemon): # Es-un Pokemon
    pass
```

```python
pok1 = Pokemon()
pok2 = ElectricPokemon()
pok3 = FirePokemon()
```

```python
for pok in [pok1,pok2,pok3]:
    print(pok.__class__.__name__, 'health: ',pok.health)
```

```
Pokemon healt:  100
ElectricPokemon health: 100
FirePokemon healt:  100
```

ElectricPokemon “Es-un” Pokemon, por lo tanto, hereda el atributo health. Por otra parte, se puede decir que Pokemon es la **clase base** de ElectricPokemon.



KIBERNUM

Chequeando clases y subclases

```
# chequeando si los objetos son instancias de Pokemon
print(isinstance(pok1,Pokemon))
print(isinstance(pok2,Pokemon))
print(isinstance(pok3,Pokemon))
```

```
True
True
True
```

```
# ElectricPokemon es subclase de Pokemon
print(issubclass(ElectricPokemon,Pokemon))

# FirePokemon es subclase de Pokemon
print(issubclass(FirePokemon,Pokemon))

# Pokemon No es subclase de FirePokemon
print(issubclass(Pokemon,FirePokemon))

# Pokemon es subclase de sí misma
print(issubclass(Pokemon,Pokemon))
```

```
True
True
False
True
```

Nótese que la clase Pokemon es subclase de sí misma

Las funciones isinstance() e issubclass() permiten verificar si una instancia corresponde a una clase, y si una clase es subclase de otra, respectivamente.

Accesando la clase base

Supongamos tenemos una subclase que posee las mismas variables que su clase base. En el código de la izquierda, se inicializa la subclase definiendo los mismos atributos de clase que su ancestro. Esto no es una buena práctica, puesto que se estarían duplicando los parámetros y tendríamos dos sets de instrucciones haciendo lo mismo.

```python
# clase base que inicializa dos variables
class Pokemon:
    def __init__(self,name,health):
        self.name = name
        self.health = health

# subclase que inicializa las mismas variables que
# su ancestro
class FirePokemon(Pokemon):
    def __init_(self,name,health,fire_temp):
        self.name = name
        self.health = health
        self.fire_temp = fire_temp
```

```python
# clase base que inicializa dos variables
class Pokemon:
    def __init__(self,name,health):
        self.name = name
        self.health = health

# esta es una mejor forma de inicializar la subclase
class FirePokemon(Pokemon):
    def __init_(self,name,health,fire_temp):

        # llamamos al inicializador de la clase ancestra
        Pokemon.__init__(self,name,health)

        # incorporamos las demas variables
        self.fire_temp = fire_temp
```

Nótese cómo en este código estamos llamando al inicializador de la clase base al momento de inicializar la subclase. Esta es una mejor práctica. Sin embargo, si cambiamos el nombre de la clase Pókemon, debemos cambiar también, la forma en que inicializamos la subclase.

Accesando la clase base

Para no referenciar el nombre de una clase al momento de inicializar la subclase, la buena práctica consiste en utilizar la función super(), como se muestra a continuación.

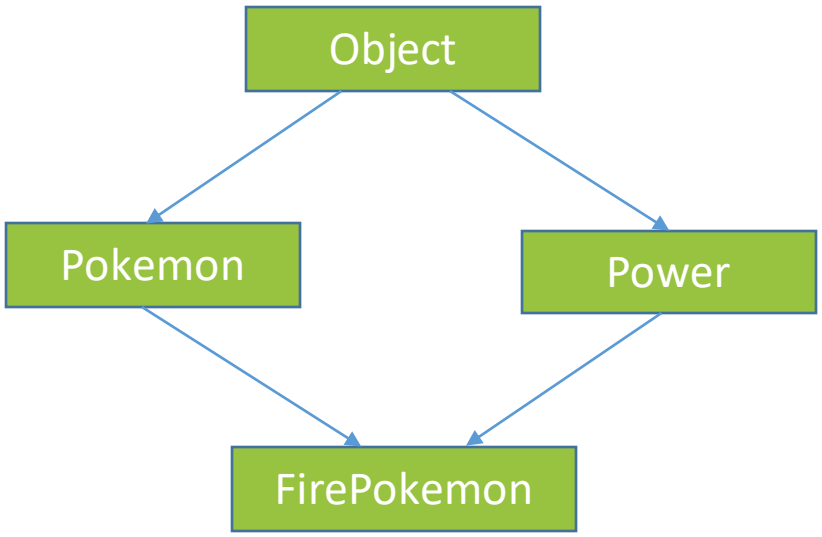
```python
# clase base que inicializa dos variables
class Pokemon:
    def __init__(self,name,health):
        self.name = name
        self.health = health

# esta es una mejor forma de inicializar la subclase
class FirePokemon(Pokemon):
    def __init__(self,name,health,fire_temp):

        # llamamos al inicializador de la clase ancestra
        # en este caso, utilizamos una forma más general
        super().__init__(name,health)

        # incorporamos las demas variables
        self.fire_temp = fire_temp
```

```python
pok1 = FirePokemon('Growlithe',100,700)
print(pok1.name)
```

```
Growlithe
```

Herencia múltiple

Una característica distintiva de Python es que permite la herencia múltiple, cosa que otros lenguajes no lo permiten (por ejemplo, Java). La herencia múltiple consiste en que una clase puede tener dos clases bases o ancestros.




```python
# clase base
class Pokemon:
    def __init__(self,name,health):
        self.name = name
        self.health = health
# clase base
class Power:
    def firePower(self, intensity):
        print(f'Disparando poder con intensidad {intensity}')

# subclase que hereda de Pokemon y Power
class ElectricPokemon(Pokemon,Power):
    def __init__(self,name,health,fire_temp):
        super().__init__(name,health)
        self.fire_temp = fire_temp
```

```python
pok1 = ElectricPokemon('Bulbasaur',100,700)
print(pok1.name)
pok1.firePower(50)
```

```
Bulbasaur
Disparando poder con intensidad 50
```

KIBERNUM

Orden de resolución de métodos

- Cuando se busca un atributo en un objeto, si no es encontrado, Python sigue buscando a nivel de clase. Lo mismo sucede con los métodos, primero busca en el objeto y si no existiera, continúa su búsqueda por las clases ancestras.
- Sin embargo, cuando hay herencia múltiple eso se torna un poco más dificultoso. Par este efecto, Python provee una vía para siempre conocer el orden en el cual las clases son buscadas, el cual se le conoce como **Method Resolution Order (MRO)**.
- Nótese en este caso el orden de búsqueda es clase **D-B-C-A-object**. El método mro() permite visualizar el orden de resolución.

```
class A:
    label='A'

class B(A):
    label='B'

class C(A):
    label='C'

class D(B,C):
    pass

d = D()

print(d.label)
print(d.__class__.mro()) # imprime MRO
```

```
B
[<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'objec
t'>]
```

KIBERNUM

Orden de resolución de métodos

- En este ejemplo, hemos comentado el atributo label en la clase B. Ahora el orden de búsqueda es D-B-C-A-object, por lo que en pantalla se imprime el valor C para el atributo label.
- En el día a día es probable que no sea común enfrentarse con este tipo de problema, pero es conveniente conocer este fenómeno puesto que en oportunidades se estará utilizando librerías o frameworks que podrían tener un comportamiento en donde se agradecerá haber leído acerca de él.

```
class A:
    label='A'

class B(A):
    #label='B'
    pass

class C(A):
    label='C'

class D(B,C):
    pass

d = D()

print(d.label)
print(d.__class__.mro())
```

```
C
[<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'objec
t'>]
```

KIBERNUM

Polimorfismo

Polimorfismo está relacionado con tener varias formas. En programación, el polimorfismo puede ser implementado de diversas formas. Algunos lenguajes como Java lo implementan a partir de herencia o de interfaces, en Python, el polimorfismo es implícito.

A continuación, algunos ejemplos de cómo implementarlo:

- Por ejemplo, una función que puede ser utilizada con argumento de tipo lista o de tipo **string**, pensemos en la función **len()**. En este caso, es la misma función con distinta firma.

- Otro caso puede ser implementado con métodos de clases.

- Y una tercera forma podría ser implementada con herencia, en donde una clase hija implementa de forma distinta un método de la clase padre, mediante sobrescritura.

Polimorfismo mediante herencia

- En Python, polimorfismo nos permite definir métodos en una clase hija que tiene el mismo nombre de método en su clase ancestra. En herencia, la clase hija hereda los métodos de la clase padres. Sin embargo, es posible modificar un método de la clase hija que fue heredado.

- Esto es especialmente útil cuando un método heredado de la clase padre no se ajusta del todo a las necesidades particulares de la clase hija. En este caso, se reimplementa el método en la clase hija, lo cual se le denomina Sobrescritura.

```python
class Pokemon:
    def __init__(self, name, health, ot):
        self.__name = name
        self.__health = health
        self.__ot = ot

    def fire_power(self):
        return 'Disparando poder'

class ElectricPokemon(Pokemon):
    def fire_power(self):
        return 'Lanzando el poder electrico'

class WaterPokemon(Pokemon):
    def fire_power(self):
        return 'Lanzando todo el poder del agua'
```

```python
pok1 = Pokemon('Generic',100,'Electric')
print(pok1.fire_power())

pok2 = WaterPokemon('Acquachu',100,'Water')
print(pok2.fire_power())

pok3 = ElectricPokemon('Pikachu',100,'Electric')
print(pok3.fire_power())
```

```
Disparando poder
Lanzando todo el poder del agua
Lanzando el poder electrico
```

KIBERNUM

Dudas y consultas

Fin presentación

KIBERNUM