

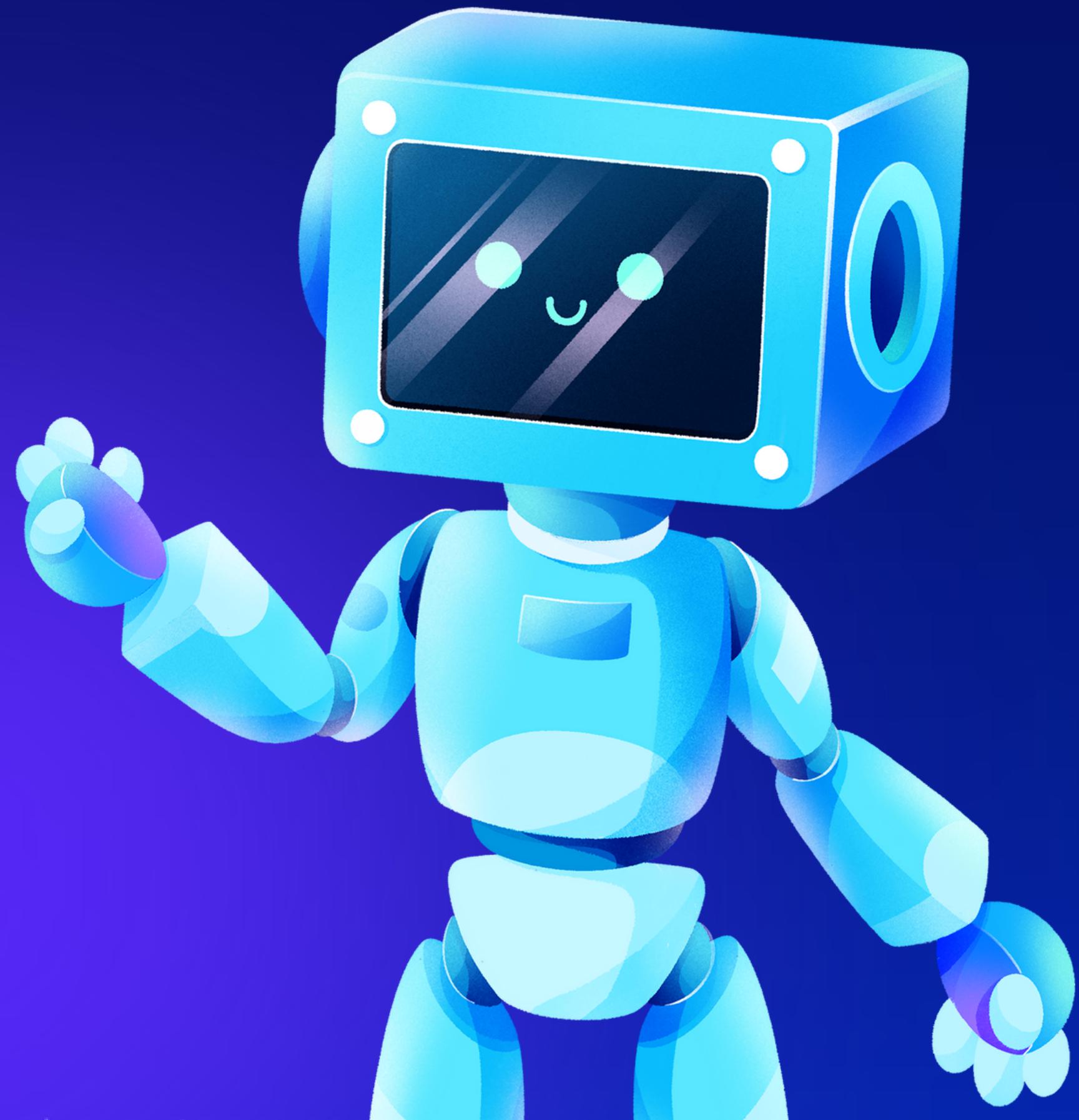
ALGORITMOS 2

JAVA

A

PYTHON

Licenciatura en Ciencias de Datos



INDICE

- Paradigma OO 01
- Clases y Enum 02
- Atributos y Métodos 03
- POO - Herencia 04
- POO - Polimorfismo 05
- Clase Abstracta e Interfaz -
Decorator 06
- Main y Constructor 07
- Property - Funciones Internas 08
- Type Hints 09





CLASES Y METODOS ABSTRACTOS

Una **clase abstracta** es una clase que no se puede instanciar directamente, no podemos crear un objeto directamente a partir de una clase abstracta.

Sirven como plantillas o modelos para otras clases que las extienden. Estas clases derivadas o subclases heredan la estructura y el comportamiento de la clase abstracta

Son útiles cuando deseamos definir una estructura común para un grupo de clases que comparten ciertas características o comportamientos similares. Al hacerlo podemos evitar duplicar código y promover la reutilización y la coherencia en nuestro diseño. A su vez, nos permiten establecer contratos o reglas que las subclases deben seguir si las definimos con métodos abstractos.

Un **método abstracto** es un método que se declara su firma pero no contiene una implementación en esa clase. No tiene un cuerpo de código definido para esa clase. La responsabilidad de proporcionar la implementación recae en las subclases que heredan de la clase abstracta o implementan la interfaz.

Reglas a recordar:

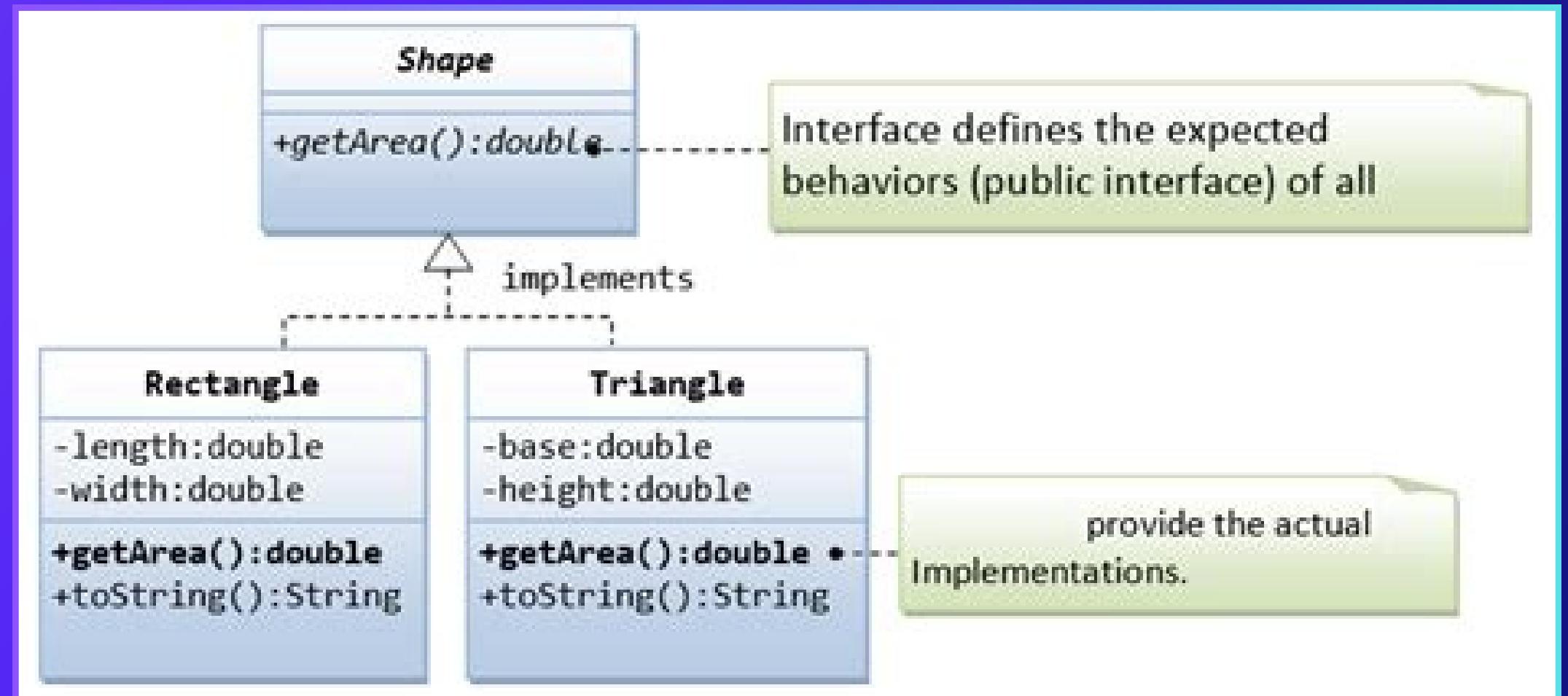
- Una clase abstracta puede tener definidos métodos abstractos.
- Si una clase tiene definidos métodos abstractos, entonces debe ser definida como clase abstracta.
- Si una clase hereda algún método abstracto, debe sobreescribirlo con su propia implementación para poder ser una clase concreta.



INTERFAZ

Una **interfaz** es un contrato que define el comportamiento mínimo que debe cumplir una clase que la implementa u otra interfaz que la extiende.

Básicamente podríamos decir que es una colección de métodos abstractos (sin implementación) que define un conjunto de acciones o comportamientos que una clase debe proporcionar. Al igual que con las clases abstractas, una interfaz no puede instanciarse.





INTERFAZ

Las interfaces son solo un **conjunto de métodos**
característicos de diversas clases, independientemente de
la relación que dichas clases mantengan entre sí.

Las interfaces son útiles por varias razones:

- **Establecen contratos:** Las interfaces definen contratos que las clases deben seguir, lo que garantiza que ciertos métodos estén disponibles y tengan la misma firma en todas las clases que implementen la interfaz.
- **Promueven la abstracción:** Las interfaces permiten la abstracción, ya que los detalles de implementación no se especifican en la interfaz, sino solo las firmas de los métodos. Esto ayuda a separar la especificación de la implementación.
- **Facilitan la interoperabilidad:** Diferentes clases pueden implementar la misma interfaz, lo que facilita la sustitución de objetos y la construcción de sistemas más flexibles y extensibles.



CLASE ABSTRACTA VS INTERFAZ

Considerar Clases abstractas si:

- Necesitamos compartir implementación de métodos entre clases bien relacionadas.
- Necesitamos mantener un estado o estructura en común para objetos de clases relacionadas (definir atributos o métodos de instancia).

Considerar Interfaces si:

- Esperamos aplicar cierto comportamiento a clases no relacionadas.
- Necesitamos sólo definir un comportamiento para cierto tipo de dato.



DECORADORES

En Python, los decoradores son una característica que permite modificar o extender el comportamiento de funciones o clases de una manera concisa y reutilizable. Se utilizan para agregar funcionalidades adicionales a las funciones o métodos sin modificar su código interno. Los decoradores son funciones que reciben otra función como argumento y retornan una nueva función con la transformación aplicada. Así es posible incorporar funcionalidad antes y después de la evaluación de la función pasada como argumento.

```
def decorador(funcion):
    def funcion_decorada():
        print("Antes de llamar a la función")
        funcion()
        print("Después de llamar a la función")
    return funcion_decorada

@decorador
def saludar():
    print("Hola mundo")

saludar() | # Esto imprimirá: Antes de llamar a la función, Hola mundo, Después de llamar a la función
```

En este ejemplo, `decorador()` es una función que toma otra función como argumento, define una nueva función `funcion_decorada()` que envuelve la función original y devuelve esta nueva función. Al usar `@decorador` encima de la definición de la función `saludar()`, estamos aplicando el decorador a la función `saludar()`, lo que significa que cuando llamamos a `saludar()`, en realidad estamos llamando a la función decorada `funcion_decorada()`.



CLASE ABSTRACTA EN PYTHON

Podemos definir este tipo de clases a través del módulo abc, llamadas Abstract Base Classes. En Python no tenemos un mecanismo por el cual evitar instanciar una clase abstracta, para forzar este comportamiento debemos agregar al menos un método abstracto utilizando el decorador @abstractmethod.

```
from abc import ABC, abstractmethod

class Vehiculo(ABC):
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    @abstractmethod
    def mostrar_info(self):
        raise NotImplementedError

vehiculo = Vehiculo("Toyota", "Corolla")    # TypeError: Can't instantiate abstract class Vehiculo with abstract method mostrar_
```



INTERFAZ EN PYTHON

```
from zope.interface import Interface, implementer

class IFlyable(Interface):
    def fly():
        "Fly method"

@implementer(IFlyable)
class Bird:
    def fly(self):
        return "I'm flying!"
```

En este ejemplo, `IFlyable` es una interfaz que define un método `fly`. La clase `Bird` implementa esta interfaz proporcionando su propia implementación del método `fly`. El decorador `@implementer` se utiliza para marcar una clase como implementadora de una interfaz.



MAIN EN JAVA

Cuando una clase tiene definido un método especial main, podremos ejecutarla y así se invocará a este método como punto de entrada de la ejecución. La firma del método es la siguiente:

```
public static void main(String[] args)
```

El parámetro args permite aceptar argumentos de entrada al programa cuando se ejecuta la clase. En general sólo implementaremos este método en sólo una clase en nuestro programa, la cual será la clase a ejecutar.

```
public class Prueba {  
    public static void main(String[] args) {  
        for (int i=0; i < args.length; i++) {  
            System.out.println("Argumento " + i + ": " + args[i]);  
        }  
    }  
}
```



CONSTRUCTORES EN JAVA

Los constructores en Java son métodos especiales que se utilizan para inicializar objetos de una clase. Se llaman automáticamente cuando se crea una nueva instancia de la clase.

Para definir un constructor en Java, debemos seguir las siguientes reglas:

- El nombre del constructor debe coincidir exactamente con el nombre de la clase.
- Los constructores no tienen un tipo de retorno, ni siquiera void.

```
Persona persona1 = new Persona("Juana", 30);
```

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    // Constructor con parámetros  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}
```

- Si no definimos ningún constructor en una clase, el compilador proporcionará un constructor por defecto sin argumentos automáticamente. Este constructor por defecto inicializa los campos con valores predeterminados (por ejemplo, 0 para enteros, null para objetos, etc.) y simplemente invoca al constructor de la superclase (super();).



MAIN EN PYTHON

En Python, no existe una función "main" estricta como en algunos otros lenguajes de programación como Java. Sin embargo, es común seguir una convención similar para organizar el código de un script.

La función `main()` se define para encapsular la lógica del programa.() se define para encapsular la lógica del programa.

La declaración `if __name__ == "__main__":` asegura que la función `main()` solo se llame si el script se ejecuta directamente. Esta estructura ayuda a organizar el código de manera lógica y facilita entender dónde comienza la ejecución del script.

El bloque `if __name__ == "__main__":` es una convención común en Python para distinguir entre el código que debe ejecutarse cuando el script se ejecuta directamente y el código que no debe ejecutarse cuando se importa como un módulo.

```
def main():
    # Your main logic goes here
    print("Hello, this is the main function!")

# Check if the script is being run directly (not imported as a module)
if __name__ == "__main__":
    # Call the main function when the script is executed
    main()
```



POO EN PYTHON

En Python las variables no tienen asociado un tipo de dato y es un lenguaje de tipado dinámico. Gracias al uso de type hints de tipo en nuestro código, podemos utilizar librerías no nativas que permiten hacer una verificación de tipos, similar a un type checker de un lenguaje tipado como Java. Una variable en Python es simplemente una etiqueta a una referencia en memoria.

EN PYTHON TODO ES UN OBJETO! clases, instancias de clases, funciones, módulos.

Cuando se genera una variable a través de la asignación =, estamos asociando esta etiqueta (nombre de la variable) al objeto asignado. A partir de ese momento, podemos acceder al objeto en memoria a través de esta etiqueta. Si la variable luego se asigna a otro nuevo objeto y no quedan variables que refieran al objeto previo, el recolector de basura se encargará de liberar de la memoria a ese objeto.

```
nombre = "Emma"  
nombre2 = nombre  
id(nombre)          # 2365055303536  
id(nombre2)         # 2365055303536
```



CONSTRUCTOR EN PYTHON

Cuando extendemos una clase debemos tener presente los argumentos que recibe su constructor, ya que si la superclase y subclase tienen un constructor definido, debemos invocar al primero explícitamente en el constructor de nuestra subclase.

Para invocar un miembro de la superclase debemos accederlo con la referencia `super()`

```
class Persona:  
    def __init__(self, nombre, apellido):  
        self.nombre = nombre  
        self.apellido = apellido  
  
class Estudiante(Persona):  
    def __init__(self, nombre, apellido, matricula):  
        super().__init__(nombre, apellido) # Invoca constructor de Persona  
        self.matricula = matricula  
  
juana = Estudiante("Juana", "Lopez", 1234)
```



ACCESIBILIDAD A MIEMBROS DE CLASE

En Python no disponemos de un mecanismo que nos permita modificar la visibilidad de los elementos de una clase, tal como teníamos en Java con los modificadores de acceso private, protected, etc. En general se distingue sólo entre miembros públicos y no públicos, definidos usualmente mediante una convención del nombre.

Un miembro de una clase con un nombre que comienza con `_`, se asume es no público pero Python NO LO RESTRINGE.

```
class MiClase:  
    def __init__(self):  
        self.x = 1  
        self._y = 2  
        self.__z = 3  
  
mi_objeto = MiClase()  
print(dir(mi_objeto))          # ['__MiClase__z', ..., '_y', 'x']  
print(mi_objeto.x)            # 1  
print(mi_objeto._y)           # 2  
print(mi_objeto.__MiClase__z)  # 3  
mi_objeto.__MiClase__z = 9  
print(mi_objeto.__MiClase__z)  # 9
```



ATRIBUTOS -> PROPIEDADES

Una alternativa que ofrece Python para mejorar el encapsulamiento y consistencia de nuestras clases es a través de la conversión de los atributos en propiedades. Esta funcionalidad que viene incorporada en el lenguaje permite definir getters y setters para operar con la estructura interna.

```
# getter
@property
def nombre_atributo(self):
    return self._nombre_atributo

# setter
@nombre_atributo.setter
def nombre_atributo(self, valor):
    self._nombre_atributo = valor
```



FUNCIONES INTERNAS

Funciones definidas dentro de otras funciones. Tienen acceso al ámbito local de la función externa, lo que significa que pueden acceder a las variables locales y los parámetros de la función externa. Las funciones internas pueden ser utilizadas para modularizar el código y encapsular la lógica

```
def funcion_externa():
    def funcion_interna():
        return "Esta es una funcion interna."
    return funcion_interna()

print(funcion_externa()) # "Esta es una funcion interna."
print(funcion_interna()) # NameError: name 'funcion_interna' is not defined
```

```
class Greeting:
    def __init__(self, name):
        self.name = name

    def greet(self):
        def get_greeting():
            return f"Hello, {self.name}!"
        return get_greeting()

# Example usage
greeting = Greeting("Alice")
print(greeting.greet()) # Output: Hello, Alice!
```



TYPE HINTS

Se trata de una especie de extensión al lenguaje que nos permitirá trabajar con Python como si fuese un lenguaje tipado, es decir, incorporar un sistema de tipos estático.

Los type hints o anotaciones de tipos en Python son una característica introducida en Python 3.5. Estas anotaciones de tipo opcionales que se pueden añadir a las variables, parámetros de funciones y valores de retorno para indicar el tipo de datos que se espera en una determinada posición en el código.

Beneficios:

- Documentación del código: Los type hints proporcionan información adicional sobre los tipos de datos que se esperan.
- Mejora la legibilidad: se hace más evidente el propósito y la función de las variables, parámetros y valores de retorno en el código.
- Facilita la detección de errores: pueden ayudar a detectar errores de tipo en tiempo de desarrollo, incluso antes de ejecutar el código. Recordemos que Python es un lenguaje interpretado, por lo cual carece de una etapa de compilación donde se podrían verificar estas condiciones.



TYPE HINTS

Sintaxis

Se agregan anotaciones de tipo utilizando dos puntos (:) después del nombre de la variable o parámetro, seguido del tipo de dato esperado. En el caso de anotar un tipo de dato de retorno, se utiliza -> al final de la firma de la operación.

```
def potencia(base: float, exponente: int) -> float:  
    return base ** exponente  
  
potencia(10, 2)      # 100
```

La función espera dos parámetros, el primero de tipo float y el segundo de tipo int y devuelve un valor de tipo float. Notemos que aún si invocamos la operación con tipos incompatibles con los declarados en la firma, por ejemplo con `potencia(10, 2.4)`, Python aún permite la evaluación de la función.



TYPE HINTS

Sintaxis

Colecciones

De la misma forma que podemos utilizar type hints para tipos de datos primitivos, también podemos utilizarlos para declarar tipos de datos de colecciones de objetos.

Listas

En general las listas están compuestas por elementos de un mismo tipo de dato

```
var: list[int]  
var=[1,2,3,4,5]
```



TYPE HINTS

Sintaxis

Colecciones

Conjuntos

Los conjuntos son similares a las listas, sólo que son una colección desordenada y de elementos sin duplicar.

```
s1: set[int] = {1, 2, 4}
s2: set[str] = {'a', 'b', 'c'}
s3: set[list[int]] = {[1,2], [3,5]}      # TypeError: unhashable type: 'list'
s3: set[tuple[int]] = {(1,2), (3,5)}      # ok
```

Un set requiere que sus elementos sean hashables para poder determinar su identidad, por lo cual el tipo debe implementar el método `__hash__()`. El tipo list no lo tiene implementado y por lo tanto no puede ser utilizado en un set, pero sí tuple que es inmutable.



TYPE HINTS

Sintaxis

Colecciones

Diccionarios

Los diccionarios pueden definirse con dos tipos de datos, el primero corresponde al tipo de dato de las claves y el segundo al tipo de dato de los valores.

```
d1: dict[str, float] = {'a': 2.1, 'b': 3.4}
d2: dict[int, list[str]] = {1: ['a','b'], 2: ['c','d']}
d3: dict[list[int], int] = {[1,2]: 0, [3,5]: 1}      # TypeError: unhashable type: 'list'
d3: dict[tuple[int, ...], int] = {(1,2): 0, (3,5): 1}  # ok
```

Similar al caso previo, los tipos dict necesitan valores hashables como claves