

# Checkpoint 6

## 1. ¿Para qué usamos clases en Python?

En Python, las clases son estructuras fundamentales que nos permiten organizar y estructurar nuestro código de manera más modular y orientada a objetos. Una clase en Python es como un plano o una plantilla para crear objetos. Los objetos son instancias de una clase, y cada objeto tiene sus propias características (atributos) y comportamientos (métodos).

Usamos clases en Python por varias razones:

- **Abstracción:** Las clases nos permiten representar objetos del mundo real en nuestro código de una manera más fácil de entender. Por ejemplo, podemos tener una clase Coche que representa diferentes coches en nuestro programa.
- **Reutilización de código:** Al encapsular datos y funcionalidades relacionadas en clases, podemos reutilizar este código en diferentes partes de nuestro programa. Por ejemplo, si tenemos una clase Animal con métodos y atributos comunes a todos los animales, podemos reutilizar esta clase para crear diferentes tipos de animales, como Perro o Gato.
- **Modularidad:** Las clases nos permiten organizar nuestro código en componentes independientes y cohesivos. Esto hace que nuestro código sea más fácil de entender, mantener y escalar, ya que cada clase se encarga de una parte específica de la funcionalidad.
- **Herencia:** La herencia nos permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos. Esto nos permite crear jerarquías de clases donde las clases hijas heredan comportamientos de las clases padre.
- **Polimorfismo:** El polimorfismo nos permite usar un mismo nombre de método en diferentes clases, pero con comportamientos diferentes. Esto nos brinda flexibilidad en nuestro código y nos permite escribir código más genérico y reutilizable.

## Ejemplo

```
class Bienvenido:
    def saludo(self):
        return "Bienvenido a la clase de matematicas"

matematicas = Bienvenido()
print(matematicas.saludo())
```

Run

Bienvenido a la clase de matematicas

- Se define la clase Bienvenido usando la palabra clave class.
- Dentro de la clase, se define un método llamado saludo utilizando la palabra clave def.  
Este método toma un parámetro self, que hace referencia a la instancia de la clase.
- Dentro del método, se devuelve la cadena "Bienvenido a la clase de matemáticas".
- Se crea una instancia de la clase Bienvenido llamada matemáticas utilizando el constructor Bienvenido().
- Se llama al método saludo en la instancia matemáticas usando la sintaxis de punto (matematicas.saludo()).
- El método saludo devuelve la cadena "Bienvenido a la clase de matemáticas".  
La cadena devuelta se imprime en la consola usando la función print().

**\*En resumen, este código crea una clase llamada Bienvenido con un método saludo que devuelve un mensaje de bienvenida. Luego, se crea una instancia de la clase y se llama al método saludo en esa instancia para imprimir el mensaje de bienvenida en la consola.**

## 2. ¿Qué método se ejecuta automáticamente cuando se crea una instancia de una clase?

El método que se ejecuta automáticamente cuando se crea una instancia de una clase en Python es el método `__init__()`, que se conoce como el constructor de la clase.

Este método se utiliza para inicializar los atributos de la instancia y puede tomar argumentos para configurar el estado inicial del objeto.

## Ejemplo



```
class Bienvenido:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludo(self):
        return f'Bienvenido {self.nombre} tienes {self.edad} años puedes ingresar'

matematicas = Bienvenido('Thomas', 19)
print(matematicas.saludo())
```

Run

Bienvenido Thomas tienes 19 años puedes ingresar

- La clase `Bienvenido` tiene un método `__init__` que se ejecuta automáticamente cuando se crea una instancia de la clase. Este método toma tres parámetros: `self`, que hace referencia a la instancia misma, `nombre`, que representa el nombre del estudiante, y `edad`, que representa la edad del estudiante. Dentro del método, los valores `nombre` y `edad` se asignan a los atributos `self.nombre` y `self.edad`, respectivamente.
- El método `saludo` toma un parámetro `self` y devuelve un mensaje de bienvenida personalizado utilizando el nombre y la edad proporcionados al crear la instancia. El mensaje se construye utilizando una cadena de formato (f-string), que permite incrustar variables dentro de una cadena.
- Se crea una instancia de la clase `Bienvenido` llamada `matemáticas`, pasando los valores `'Thomas'` y `19` como argumentos para `nombre` y `edad`, respectivamente.
- Se llama al método `saludo` en la instancia `matemáticas` y se imprime el mensaje de bienvenida personalizado en la consola utilizando la función `print()`.

### 3. ¿Cuáles son los tres verbos de API?

Los tres verbos principales de una API son GET, POST y DELETE.

- **GET:** Se utiliza para recuperar información del servidor.  
Por ejemplo, al realizar una solicitud GET a una URL de una API de noticias, podemos obtener los titulares de las noticias actuales.  
Por ejemplo:

```
GET https://api.example.com/news
```

**Respuesta GET:**

```
{
  "headlines": [
    "Últimas noticias: COVID-19",
    "Nuevos avances en inteligencia artificial",
    "Evento de lanzamiento de productos tecnológicos"
  ]
}
```

- **POST:** Se utiliza para enviar datos al servidor para crear un nuevo recurso.  
Por ejemplo:

```
POST https://api.example.com/users
Content-Type: application/json

{
  "username": "usuario123",
  "email": "usuario123@example.com",
  "password": "contraseña123"
}
```

**Respuesta POST:**

```
{
  "message": "Usuario creado exitosamente"
}
```

- **DELETE:** para eliminar un recurso de una API  
Por ejemplo:

```
DELETE https://api.example.com/posts/123
```

**Respuesta DELETE:**

```
{  
  "message": "La publicación se ha eliminado correctamente"  
}
```

**\*Estos son sólo ejemplos ilustrativos de cómo se pueden utilizar los verbos GET, POST y DELETE en las solicitudes a una API para realizar diferentes acciones, como obtener datos, enviar información y eliminar recursos, respectivamente.**

## 4. ¿Qué es MongoDB una base de datos o noSQL?

MongoDB es una base de datos NoSQL orientada a documentos.

Se utiliza para almacenar volúmenes masivos de datos.

A diferencia de una base de datos relacional SQL tradicional, **MongoDB** no se basa en tablas y columnas. Los datos se almacenan como colecciones y documentos.

### Ejemplo 🙌

- Ejemplo de cómo insertar un documento en una colección llamada person en MongoDB.

```

db.person.insertOne(
  {
    name: 'Thomas',
    company: 'Healthetia',
    telephone: [
      { home: '378910112' },
      { work: '555789047' }
    ]
  }
);

```

- `db.person.insertOne({ ... })`:  
Esto inserta un nuevo documento en la colección `person`.

El documento que se inserta tiene tres campos:

**name**: El nombre es "Thomas".

**company**: La compañía es "Healthetia".

**telephone**: Este campo es un array que contiene dos objetos.

Cada objeto representa un número de teléfono, uno para el home y otro para el work.

\*Por lo tanto, este comando insertará un documento en la colección `person` con el nombre "Thomas", la compañía "Healthetia" y dos números de teléfono: uno para el home (378910112) y otro para el work (555789047).

- Ejemplo ejecutando una consulta que devuelva todos los resultados de la colección `person`:

```
db.person.find({});
```

El comando `db.person.find({})`; se utiliza en MongoDB para recuperar todos los documentos de la colección `person`.

**db.person**: Esto indica que queremos realizar la operación en la colección llamada `person`.

**.find({})**: Este es el método utilizado para buscar documentos en la colección.

El {} dentro de los paréntesis indica que no hay criterios de búsqueda específicos, lo que significa que recuperará todos los documentos de la colección.

Por lo tanto, al ejecutar este comando, se recuperarán todos los documentos de la colección person en la base de datos MongoDB.

- Ejemplo de resultado de la colección person

```
"_id" : ObjectId("62442429854636a03f6b8534"),
name: 'Thomas',
company: 'Healthetia',
telephone : [
  { home: '378910112' },
  { work: '555789047' }
]
```

**\*En resumen, este documento representa la información de una persona llamada Thomas, que trabaja en Healthetia y tiene dos números de teléfono: uno para el home y otro para el work.**

👉 [Mongodb](#)

## 5. ¿Qué es una api?

Una API (**Application Programming Interface**) es un conjunto de reglas y definiciones que permite que diferentes aplicaciones se comuniquen entre sí. En el contexto del desarrollo de software, una API generalmente se refiere a un conjunto de funciones, protocolos y herramientas que permiten a los desarrolladores de software interactuar con cierto software o plataforma de manera programática.

Las API pueden tener diferentes formas y propósitos, pero su función principal es proporcionar una interfaz estandarizada que permite a los desarrolladores acceder a ciertas funcionalidades o datos de una aplicación o servicio de manera controlada y segura.

**Por ejemplo:**

- Las API de redes sociales como la API de Twitter o la API de Facebook permiten a los desarrolladores acceder a datos como publicaciones, perfiles de usuario y estadísticas de uso de sus

plataformas para integrar esta información en sus propias aplicaciones.

- Las API de servicios de pago como Stripe o PayPal permiten a los desarrolladores procesar pagos en línea desde sus propias aplicaciones.
- Las API de mapas como Google Maps permiten a los desarrolladores integrar funcionalidades de mapas y geolocalización en sus aplicaciones.

**\*En resumen, las API son herramientas fundamentales en el desarrollo de software moderno, ya que permiten la integración y la interoperabilidad entre diferentes sistemas y servicios de manera eficiente y segura.**

## 6. ¿Qué es postman?

Postman es una plataforma de colaboración para el desarrollo de API (**Application Programming Interface**) que permite a los desarrolladores construir, probar y documentar API de manera más eficiente.

Es una herramienta ampliamente utilizada en el desarrollo de software para realizar diversas tareas relacionadas con el manejo de API.

Las principales características y funcionalidades de Postman incluyen:

- **Creación de solicitudes HTTP:** Postman permite a los usuarios enviar solicitudes HTTP, como GET, POST, PUT, DELETE, etc., a diferentes endpoints de una API para interactuar con ella y probar su funcionamiento.
- **Organización de solicitudes en colecciones:** Los desarrolladores pueden organizar y agrupar solicitudes relacionadas en colecciones, lo que facilita la gestión y la ejecución de pruebas de API.



- **Variables y entornos:** Postman permite definir y utilizar variables y entornos para personalizar y parametrizar las solicitudes de API, lo que facilita la creación de pruebas y la automatización de flujos de trabajo.
- **Automatización de pruebas:** Postman ofrece capacidades de automatización que permiten a los desarrolladores ejecutar pruebas automatizadas en sus API y validar su funcionamiento de manera programática.
- **Documentación de API:** Los desarrolladores pueden generar documentación interactiva para sus API directamente desde Postman, lo que facilita la comprensión y el uso de la API por parte de otros desarrolladores.

**\*En resumen, Postman es una herramienta poderosa y versátil que proporciona a los desarrolladores una amplia gama de capacidades para construir, probar y documentar API de manera efectiva y eficiente. Es ampliamente utilizada en equipos de desarrollo de software en todo el mundo para simplificar el proceso de desarrollo y asegurar la calidad de las API.**

👉 [Postman](#)

## 7. ¿Qué es el polimorfismo?

El polimorfismo en Python trata de cómo objetos de distintas clases pueden reaccionar de manera única cuando se les pide realizar una misma acción. En otras palabras, podemos usar un método en diferentes objetos sin preocuparnos de su tipo específico, ya que cada objeto sabrá cómo responder adecuadamente al método según su propia definición de clase. Esto hace que nuestro código sea más flexible y adaptable, ya que podemos tratar diferentes objetos de manera uniforme incluso si tienen comportamientos diferentes.

Un ejemplo común de polimorfismo en Python es cuando tenemos una clase base con un método, y luego diferentes subclases que sobrescriben ese método para proporcionar su propia implementación. Esto nos permite llamar al mismo método en diferentes objetos y obtener resultados específicos de acuerdo con el tipo de objeto.

## Ejemplo

```
class Html:
    def __init__(self, content):
        self.content = content

class Titulo(Html):
    def crear_html(self):
        return f'<h1>{self.content}</h1>'

class Subtitulo(Html):
    def crear_html(self):
        return f'<h2>{self.content}</h2>'

class Parrafo(Html):
    def crear_html(self):
        return f'<p>{self.content}</p>'

#Instancias
titulo = Titulo('Bienvenido a la clase')
subtitulo = Subtitulo('Primera lección de Matemáticas')
parrafo = Parrafo('En esta lección aprenderemos a sumar')

print(titulo.crear_html())
print(subtitulo.crear_html())
print(parrafo.crear_html())
```

Run

```
<h1>Bienvenido a la clase</h1>
<h2>Primera lección de Matemáticas</h2>
<p>En esta lección aprenderemos a sumar</p>
```

En este ejemplo vemos cómo se utiliza el concepto de herencia y polimorfismo en Python para generar diferentes etiquetas HTML a partir de una estructura común. Aquí hay una explicación paso a paso:

- **Definición de la clase base Html:**

La clase Html actúa como la clase base para otras clases específicas de elementos HTML.

Tiene un constructor `__init__` que inicializa el contenido común de todos los elementos HTML.

- **Definición de las clases Titulo, Subtitulo y Parrafo:**

Estas clases heredan de la clase Html, lo que significa que heredan su comportamiento y atributos.

Cada una de estas clases implementa su propio método `crear_html()`, que es específico para el tipo de elemento HTML que representa.

Por ejemplo, Titulo tiene su propio método `crear_html()` que devuelve una cadena HTML con etiquetas `<h1>`.

- **Creación de instancias de las clases Titulo, Subtitulo y Parrafo:**

Se crean instancias de estas clases con contenido específico para cada tipo de elemento HTML.

Cada instancia tiene acceso al método `crear_html()` definido en su clase base, pero lo implementa de manera diferente según la clase específica.

- **Llamadas a los métodos `crear_html()`:**

Se invoca el método `crear_html()` en cada instancia para generar el contenido HTML correspondiente.

Cada llamada al método devuelve una cadena HTML que representa el elemento específico (título, subtítulo o párrafo).

- **Impresión de los resultados:**

Se imprimen los resultados de las llamadas a los métodos `crear_html()`, mostrando el contenido HTML generado para cada tipo de elemento.

\*En resumen, este ejemplo ilustra cómo se pueden definir clases especializadas que heredan comportamiento de una clase base común, y cómo el polimorfismo permite llamar a métodos específicos de cada clase a través de una interfaz común.

## 8. ¿Qué es el método dunder?

El término "método dunder" en Python se refiere a los métodos especiales que comienzan y terminan con doble guión bajo.

Estos métodos son implementados por clases para realizar operaciones específicas que Python reconoce y utiliza internamente en ciertos contextos.

## Ejemplos comunes de métodos dunder 🙋

- `__init__` (para inicialización)

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

# Crear una instancia de la clase Persona
persona1 = Persona("Thomas", 19)

# Acceder a los atributos de la instancia
print("Nombre:", persona1.nombre)
print("Edad:", persona1.edad)
```

Run

Nombre: Thomas  
Edad: 19

En este ejemplo, el método `__init__` se utiliza para inicializar los atributos nombre y edad de la clase Persona cuando se crea una nueva instancia de esa clase.

Cuando se crea el objeto `persona1`, se llama automáticamente al método `__init__` con los argumentos proporcionados ("Thomas" y 19) para inicializar los atributos nombre y edad de la instancia `persona1`. Luego, podemos acceder a estos atributos utilizando la sintaxis de punto (`persona1.nombre` y `persona1.edad`).

- `__str__` (para la representación en cadena)

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"Nombre: {self.nombre}, Edad: {self.edad}"

# Crear una instancia de la clase Persona
thomas = Persona("Thomas", 19)

# Imprimir la representación legible del objeto
print(thomas)
```

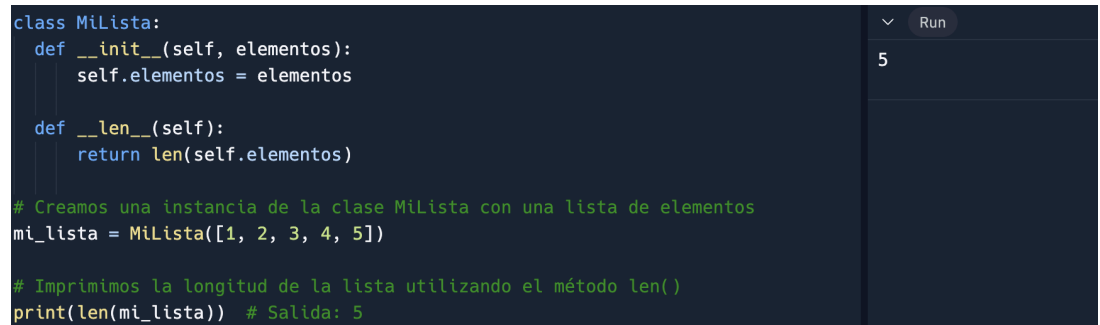
Run

Nombre: Thomas, Edad: 19

En este ejemplo, la clase Persona tiene un método `__str__` que devuelve una cadena formateada con el nombre y la edad de la persona.

Cuando llamamos a `print(thomas)`, Python invoca automáticamente el método `__str__` para obtener la representación legible del objeto `thomas`, que luego se imprime en la consola.

- `__len__`(para la obtención de la longitud)



```
class MiLista:
    def __init__(self, elementos):
        self.elementos = elementos

    def __len__(self):
        return len(self.elementos)

# Creamos una instancia de la clase MiLista con una lista de elementos
mi_lista = MiLista([1, 2, 3, 4, 5])

# Imprimimos la longitud de la lista utilizando el método len()
print(len(mi_lista)) # Salida: 5
```

The screenshot shows a code editor with the above Python code. To the right, there is a 'Run' button and a console output area displaying the number '5'.

En este ejemplo, la clase `MiLista` tiene un método `__len__` definido, que devuelve la longitud de la lista de elementos almacenada en la instancia. Cuando llamamos a `len(mi_lista)`, Python invoca automáticamente este método para obtener la longitud de la lista, que luego se imprime en la consola.

## 9. ¿Qué es un decorador python?

En Python, un decorador es una función que toma otra función como argumento y devuelve una nueva función, generalmente extendiendo o modificando el comportamiento de la función original de alguna manera.

El decorador **@property** se utiliza comúnmente para definir métodos que actúan como atributos de solo lectura en una clase. Esto significa que podemos acceder al método como si fuera un atributo normal, pero no podemos modificarlo directamente.

## Ejemplo

```
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre

    @property
    def nombre(self):
        return self._nombre

# Crear una instancia de la clase Persona
persona1 = Persona("Thomas")

# Acceder al atributo de solo lectura 'nombre' como si fuera un atributo normal
print(persona1.nombre) # Salida: Thomas

# Intentar modificar el atributo 'nombre' directamente generará un error
# persona1.nombre = "Carlos" # Esto generaría un error
```

Run

Thomas

En este ejemplo, **@property** se aplica al método nombre de la clase Persona, lo que nos permite acceder a él como si fuera un atributo normal (persona1.nombre). Sin embargo, no podemos modificarlo directamente debido a la naturaleza de solo lectura del decorador **@property**.