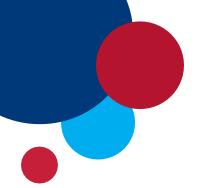


# Material de lectura



# 1

# Contenido

ESTRU	CTURAS Y PROPIEDADES DEL LENGUAJE PYTHON	2
2.1	DEFINIR LOS ATRIBUTOS Y MÉTODOS DE CREACIÓN DE PAQUETES	2
2.1.1.	ATRIBUTOS DE UN PAQUETE	2
2.1.2.	MÉTODOS DE UN PAQUETE	4
EJEMF	PLO EN PYTHON	6
RFFFR	FNCIAS BIBLIOGRÁFICAS	19





# ESTRUCTURAS Y PROPIEDADES DEL LENGUAJE PYTHON

# 2.1 Definir los Atributos y Métodos de Creación de Paquetes

En la creación de paquetes en el contexto de logística, envíos o software, los **atributos** y **métodos** definen las propiedades y comportamientos del paquete. A continuación, se describe cómo estructurar una clase que represente un paquete.

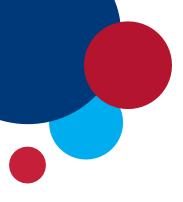
# 2.1.1. Atributos de un Paquete

Los atributos representan las características del paquete. Estos pueden incluir:

- 1. **ID del paquete**: Identificador único para el paquete.
  - o Ejemplo: "PKG12345"
- 2. **Descripción**: Información sobre el contenido del paquete.
  - o Ejemplo: "Electrodomésticos"
- 3. **Peso**: Peso del paquete en kilogramos o libras.



- o Ejemplo: 2.5 kg.
- 4. **Dimensiones**: Largo, ancho y alto del paquete (en cm).
  - o Ejemplo: Largo=50, Ancho=30, Alto=20
- 5. **Destino**: Dirección o ubicación de entrega.
  - o Ejemplo: "Asunción, Paraguay"
- 6. **Estado**: Estado actual del paquete.
  - o Ejemplo: "Pendiente", "En tránsito", "Entregado"
- 7. Fecha de creación: Fecha en que se creó el paquete.
  - o Ejemplo: "2025-01-15"
- 8. **Fecha estimada de entrega**: Fecha prevista para la entrega.
  - o Ejemplo: "2025-01-20"



## 2.1.2. Métodos de un Paquete

Los métodos son las acciones que se pueden realizar sobre el paquete. Estos pueden incluir:

#### 1. Inicializar un paquete:

o Método constructor (\_\_init\_\_) para crear un paquete con los atributos básicos.

#### 2. Actualizar estado:

o Cambiar el estado del paquete según el progreso en el proceso de envío.

#### 3. Calcular volumen:

o Calcula el volumen del paquete multiplicando largo × ancho × alto.

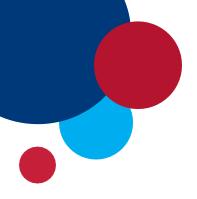
#### 4. Calcular costo de envío:

o Determina el costo del envío basado en peso, volumen o distancia al destino.

#### 5. Mostrar información del paquete:

o Muestra los detalles del paquete, como ID, descripción, destino, estado, etc.

#### 6. Validar dimensiones:



o Verifica que las dimensiones del paquete no excedan los límites permitidos.

#### 7. Estimar fecha de entrega:

o Calcula una fecha estimada de entrega basada en el destino y la fecha actual.

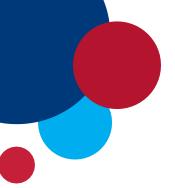


```
1 v class Paquete:
         def init (self, id paquete, descripcion, peso, largo, ancho, alto, destino):
              self.id_paquete = id_paquete
              self.descripcion = descripcion
 4
              self.peso = peso
 6
              self.largo = largo
              self.ancho = ancho
 7
              self.alto = alto
 8
              self.destino = destino
9
              self.estado = "Pendiente"
10
              self.fecha creacion = "2025-01-15"
11
              self.fecha entrega estimada = None
12
13
14 v
         def calcular volumen(self):
              """Calcula el volumen del paquete."""
15
              return self.largo * self.ancho * self.alto
17
18 V
         def calcular_costo_envio(self, tarifa_por_kg, tarifa_por_volumen):
              """Calcula el costo de envío basado en peso y volumen."""
19
              costo peso = self.peso * tarifa por kg
              volumen = self.calcular volumen()
21
              costo_volumen = volumen * tarifa_por_volumen
              return max(costo peso, costo volumen)
24
25 v
         def actualizar estado(self, nuevo estado):
              """Actualiza el estado del paquete."""
27
              self.estado = nuevo estado
```

```
28
           def mostrar_informacion(self):
29 v
                """Muestra los detalles del paquete."""
               return f"""
31
               ID: {self.id paquete}
               Descripción: {self.descripcion}
               Peso: {self.peso} kg
34
               \label{largo} \mbox{Dimensiones: } \{\mbox{self.largo}\}x\{\mbox{self.ancho}\}x\{\mbox{self.alto}\} \mbox{ cm}
               Destino: {self.destino}
               Estado: {self.estado}
               Fecha de creación: {self.fecha_creacion}
               Fecha estimada de entrega: {self.fecha_entrega_estimada or 'No disponible'}
40
41
```

```
# Crear un paquete
     paquete1 = Paquete("PKG001", "Ropa deportiva", 2.5, 30, 20, 10, "Asunción")
 2
 3
     # Calcular el costo de envío
 4
     costo_envio = paquete1.calcular_costo_envio(tarifa_por_kg=10, tarifa_por_volumen=0.0)
     print(f"Costo de envío: ${costo_envio:.2f}")
 7
     # Actualizar el estado
     paquete1.actualizar_estado("En tránsito")
10
     # Mostrar información del paquete
11
     print(paquete1.mostrar_informacion())
12
13
```

#### Salida del Programa



Costo de envío: \$25.00

ID: PKG001

Descripción: Ropa deportiva

Peso: 2.5 kg

Dimensiones: 30x20x10 cm

Destino: Asunción

Estado: En tránsito

Fecha de creación: 2025-01-15

Fecha estimada de entrega: No disponible



## 2.2 Propiedades

En Python, las **propiedades** son una forma de controlar el acceso y la modificación de los atributos de una clase utilizando **métodos especiales** llamados **getter**, **setter** y **deleter**. Esto permite encapsular la lógica asociada a la lectura, escritura o eliminación de un atributo, ofreciendo un control más fino sobre cómo se accede y se modifica.

# 2.2.1 Definición de Propiedades

Las propiedades se definen utilizando el decorador @property y sus complementos @nombre.setter y @nombre.deleter.

### 2.2.2 Estructura General

class ClaseEjemplo:

```
def __init__(self, valor):
```

self.\_atributo = valor # Atributo privado o protegido

# 11

```
@property
def atributo(self):
  """Getter: devuelve el valor del atributo."""
  return self._atributo
@atributo.setter
def atributo(self, nuevo_valor):
  """Setter: modifica el valor del atributo."""
  if nuevo_valor >= 0:
    self._atributo = nuevo_valor
  else:
    raise ValueError("El valor debe ser mayor o igual a 0.")
```

```
@atributo.deleter

def atributo(self):

"""Deleter: elimina el atributo."""

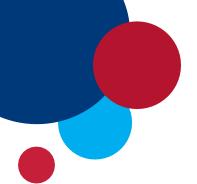
print("El atributo ha sido eliminado.")
```

# 2.2.3 Ejemplo Práctico: Clase con Propiedades

class Persona:

del self.\_atributo

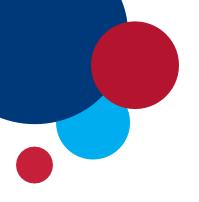
```
def __init__(self, nombre, edad):
    self._nombre = nombre # Atributo protegido
    self._edad = edad # Atributo protegido
```



# 13

```
@property
def nombre(self):
  """Getter para el atributo nombre."""
  return self._nombre
@nombre.setter
def nombre(self, nuevo_nombre):
  """Setter para el atributo nombre."""
  if len(nuevo_nombre) > 0:
    self._nombre = nuevo_nombre
  else:
    raise ValueError("El nombre no puede estar vacío.")
```





```
@property
def edad(self):
  """Getter para el atributo edad."""
  return self._edad
@edad.setter
def edad(self, nueva_edad):
  """Setter para el atributo edad."""
  if nueva_edad >= 0:
    self._edad = nueva_edad
  else:
    raise ValueError("La edad no puede ser negativa.")
```

```
@edad.deleter

def edad(self):
    """Deleter para el atributo edad."""
    print("El atributo 'edad' ha sido eliminado.")
    del self._edad
```

# 2.2.4 Uso del Ejemplo

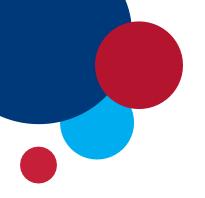
```
# Crear un objeto de la clase Persona
```

```
persona = Persona("Juan", 30)
```

# Acceder a los atributos usando los getters

print(persona.nombre) # Salida: Juan

print(persona.edad) # Salida: 30



```
# Modificar los atributos usando los setters
```

```
persona.nombre = "Ana"
```

persona.edad = 25

print(persona.nombre) # Salida: Ana

print(persona.edad) # Salida: 25

# Eliminar un atributo usando el deleter

del persona.edad # Salida: El atributo 'edad' ha sido eliminado.



### 2.2.5 Ventajas de las Propiedades

- 1. **Encapsulación**: Permite controlar el acceso a los atributos privados.
- 2. Validación: Puedes agregar lógica de validación al asignar valores a los atributos.
- 3. **Compatibilidad**: Ofrece una interfaz consistente para atributos, sin exponer detalles internos.
- 4. **Flexibilidad**: Facilita la adición de lógica sin cambiar cómo se accede al atributo desde fuera de la clase.

## 2.2.6 Propiedades vs Atributos Directos

Usar propiedades en lugar de atributos directos es útil cuando necesitas agregar lógica adicional al leer o modificar un valor. Por ejemplo, si necesitas validar que un valor sea positivo antes de asignarlo.

# Sin propiedades

persona.edad = -5 # Esto sería posible y podría generar errores.

# Con propiedades

# persona.edad = -5 # Esto lanzaría un ValueError gracias al setter.



### Referencias Bibliográficas

- Ceder, N. (2018). *The Quick Python Book* (3rd ed.). Manning Publications.

  Una guía práctica para aprender Python, con un enfoque claro en la programación orientada a objetos.
- Downey, A. B. (2015). *Think Python: How to Think Like a Computer Scientist* (2nd ed.). O'Reilly Media.
  - Proporciona una introducción detallada a Python, incluyendo los conceptos fundamentales de la programación orientada a objetos.
- Griffiths, D., & Griffiths, P. (2020). *Head First Python* (2nd ed.). O'Reilly Media. Una guía visual para aprender Python, que incluye ejemplos prácticos y explicaciones sobre el paradigma de programación orientado a objetos.
- Guttag, J. V. (2016). Introduction to Computation and Programming Using Python (2nd ed.). MIT Press.
  - Cubre los fundamentos de la programación con Python y una sólida introducción a la programación orientada a objetos.
- Lutz, M. (2013). Learning Python (5th ed.). O'Reilly Media.
   Un recurso integral para aprender Python, desde los fundamentos básicos hasta temas avanzados como la programación orientada a objetos.
- Python Software Foundation. (2023). The Python Tutorial. Recuperado de https://docs.python.org/3/tutorial/
  - Documentación oficial de Python, que incluye una sección sobre clases y objetos con ejemplos prácticos.
- Eckel, B. (2006). *Thinking in Java* (4th ed.). Prentice Hall.

  Aunque centrado en Java, este libro es ideal para entender los principios de la programación orientada a objetos, aplicables a otros lenguajes como Python.