1. **Programación orientada a objetos en Python**

Uno de los elementos más importantes de los lenguajes de programación es la utilización de clases para organizar programas en módulos y abstracciones de datos.

Las clases se pueden utilizar de diversas maneras. Pero en este artículo hablaremos de cómo utilizarlas en el contexto de la programación orientada a objetos. La clave para entender la programación orientada a objetos es pensar en objetos como agrupaciones de datos y los métodos que operan en dichos datos.

Por ejemplo, podemos representar a una persona con propiedades como nombre, edad, género, etc. y los comportamientos de dicha persona como caminar, cantar, comer, etc. De la misma manera podemos representar unos audífonos con propiedades como su marca, tamaño, color, etc. y sus comportamientos como reproducir música, pausar y avanzar a la siguiente canción.

Puesto de otra manera, la programación orientada a objetos nos permite modelar cosas reales y concretas del mundo y sus relaciones con otros objetos.

Las ideas detrás de la programación orientada a objetos tienen más de 50 años y han sido ampliamente aceptadas y practicadas en los últimos treinta. A mediados de la década de los setenta se comenzaron a escribir artículos académicos explicando los beneficios de esta aproximación a la programación. También durante esos años se comenzaron a escribir los primeros lenguajes de programación que incorporaban estas ideas (como Smalltalk y CLU). Pero no fue hasta la llegada de Java y C++ que la idea consiguió un número importante de seguidores.

Hasta ahora, en el curso previo de esta [serie](https://platzi.com/cursos/python-cs) hemos utilizado programación orientada a objetos de manera implícita. Cuando decimos “Los objetos son las principales cosas que un programa de Python manipula. Cada objeto tiene un tipo que define qué cosas puede realizar un programa con dicho objeto,” nos estamos refiriendo a las ideas principales de la programación orientada a objetos. Hemos utilizado los tipos lista y  
diccionario, entre muchos otros, así como los métodos asociados a dichos tipos.

Así como los creadores de un lenguaje de programación solo pueden diseñar una fracción muy pequeña de todas las funciones útiles (como abs, float, type, etc.), también pueden escribir una fracción muy pequeña de los tipos útiles (int, str, dict, list, etc.). Ya sabemos los mecanismos que nos permiten crear funciones, ahora veremos los mecanismos que nos permiten crear nuevos tipos (o clases).

**Clases en Python**

Las estructuras primitivas con las que hemos trabajado hasta ahora nos permiten definir cosas sencillas, como el costo de algo, el nombre de un usuario, las veces que debe correr un bucle, etc. Sin embargo, existen ocasiones cuando necesitamos definir estructuras más complejas, por ejemplo un hotel. Podríamos utilizar dos listas: una para definir los cuartos y una segunda para definir si el cuarto se encuentra ocupado o no.

cuartos\_de\_hotel = [101, 102, 103, 201, 202, 203]

cuarto\_ocupado = [**True**, **False**, **False**, **True**, **True**, **False**]

Sin embargo, este tipo de organización rápidamente se sale de control. ¿Qué tal que quisiéramos añadir más propiedades, cómo si el cuarto ya fue aseado o no? ¿Si el cuarto tiene cama doble o sencilla? Esto nos lleva a una falta fuerte de organización y es justamente el punto que justifica la existencia de clases.

Las clases nos permiten crear nuevos tipos que contiene información arbitraria sobre un objeto. En el caso del hotel, podríamos crear dos clases Hotel() y Cuarto() que nos permitiera dar seguimiento a las propiedades como número de cuartos, ocupación, aseo, tipo de habitación, etc.

Es importante resaltar que las clases solo proveen estructura. Son un molde con el cual podemos construir objetos específicos. La clase señala las propiedades que los hoteles que modelemos tendrán, pero no es ningún hotel específico. Para eso necesitamos las instancias.

**Instancias**

Mientras que las clases proveen la estructura, las instancias son los objetos reales que creamos en nuestro programa: un hotel llamado PlatziHotel o Hilton. Otra forma de pensarlo es que las clases son como un formulario y una vez que se llena cada copia del formulario tenemos las instancias que pertenecen a dicha clase. Cada copia puede tener datos distintos, al igual que cada instancia es distinta de las demás (aunque todas pertenecen a una misma clase).

Para definir una clase, simplemente utilizamos el *keyword* class. Por ejemplo:

**class** **Hotel**:

**pass**

Una vez que tenemos una clase llamada Hotel podemos generar una instancia llamando al constructor de la clase.

hotel = Hotel()

**Atributos de la instancia**

Todas las clases crean objetos y todos los objetos tienen atributos. Utilizamos el método especial \_\_init\_\_ para definir el estado inicial de nuestra instancia. Recibe como primer parámetro obligatorio self (que es simplemente una referencia a la instancia).

**class** **Hotel**:

**def** **\_\_init\_\_**(self, numero\_maximo\_de\_huespedes, lugares\_de\_estacionamiento):

self.numero\_maximo\_de\_huespedes = numero\_maximo\_de\_huespedes

self.lugares\_de\_estacionamiento = lugares\_de\_estacionamiento

self.huespedes = 0

hotel = Hotel(numero\_maximo\_de\_huespedes=50, lugares\_de\_estacionamiento=20)

print(hotel.lugares\_de\_estacionamiento) # 20

**Métodos de instancia**

Mientras que los atributos de la instancia describen lo que representa el objeto, los métodos de instancia nos indican qué podemos hacer con las instancias de dicha clase y normalmente operan en los mencionados atributos. Los métodos son equivalentes a funciones dentro de la definición de la clase, pero todos reciben self como primer argumento.

**class** **Hotel**:

...

**def** **anadir\_huespedes**(self, cantidad\_de\_huespedes):

self.huespedes += cantidad\_de\_huespedes

**def** **checkout**(self, cantidad\_de\_huespedes):

self.huespedes -= cantidad\_de\_huespedes

**def** **ocupacion\_total**(self):

**return** self.huespedes

hotel = Hotel(50, 20)

hotel.anadir\_huespedes(3)

hotel.checkout(1)

hotel.ocupacion\_total() # 2

Ahora que ya sabemos qué son las clases y cómo las podemos utilizar en nuestros programas, platícanos qué clases te serían útiles para modelar en uno de tus programas.

2.

## Tipos de datos abstractos y clases, Instancias

**Para los que se pregunten como se llego a ese resultado de forma matemática. aquí dejo mi aporte:**  
Primero empecemos con las definiciones:

* class es una clase
* def es una función, pero dentro de un class es un metodo, por lo tanto en nuestro ejemplo es un método.
* constructor es el primer método que se ejecuta
* instancia es lo que esta dentro de un método, después de self. Ejm.: (self, otra\_coordenada).

<code>

**class** **Coordenada**:

**def** **\_\_init\_\_**(self, x, y): #hasta aquí lo que esta después de self, solo son parametros.

self.x = x

self.y = y

# Aquí estamos inicializando las variables de instancia

luego escribimos el método que nos indica que hará el programa usando las instancias que se definieron en el constructor.

<code>

**def** **distancia**(self, otra\_coordenada): #el parametro otra\_coordenada es una instancia, que hara uso del molde en la primer clase de inicializacion, la cual usaremos despues, tomar atencion.

x\_diff = (self.x - otra\_coordenada.x)\*\*2

y\_diff = (self.y - otra\_coordenada.y)\*\*2

**return** (x\_diff +y\_diff)\*\*0.5

#estos solo son una representación matemática de lo que hará el programa (lo explicare después)

luego tenemos que permitir que el programa corra desde nuestro terminal con:

<code>

#mi caso yo prefiero usar comillas dobles, depende de ustedes.

**if** \_\_name\_\_ == "\_\_main\_\_":

coord\_1 = Coordenada(3, 30)

coord\_2 = Coordenada(4, 8)

#estas dos expresiones son instancias que usan el molde que es el primer método de inicializacion.

luego para ver el output que nos arroja el programa corremos print, este print hace uso de la instancia con las clases.

<code>

**print**(coord\_1.distancia(coord\_2))

#"coord\_1" hace uso **de** **la** primer instancia, mientras **que** coord\_2 al estar dentro del metodo distancia, ocupa el lugar **de** otra\_coordenada. (entender esta parte es muy importante)

lo que vimos hasta ahora es la parte del código, pero matemáticamente pasa lo siguiente:  
los parametros (x, y) en **init** son las ultimas instancias que definimos:  
(3, 30) = (x, y) ------------------------- primer instancia  
(4, 8) = (x, y) ------------------------- segunda instancia

ahora, recuerda que nuestro print fue:

print(coord\_1.distancia(coord\_2))

coord\_1 tomaria el lugar del primer metodo, es decir:  
self.x = 3  
self.y = 30  
luego como coord\_2 esta dentro del método distancia tomaría el lugar de la instancia “otra\_coordenada” pero lo mas interesante es que esta instancia hace uso del molde en el primer método:  
otra\_coordenada.x = 4  
otra\_coordenada.y = 8  
por tal motivo matemáticamente quedaría de la siguiente manera:  
x\_diff = (3 - 4)^2 = 1  
y\_diff = (30 - 8)^2 = 484

return (x\_diff + y\_diff)\*\*0.5 quedaría de la siguiente manera:  
(1 + 484)^(1/2) = 22.0227, su raiz cuadrada el resultado final.

Texto

Descripción generada automáticamente con confianza media

Texto

Descripción generada automáticamente

## Funciones: base de los decoradores6/25

**LECTURA**

El concepto de decorador en Python es algo que podríamos ubicar en un nivel “intermedio” en el manejo del lenguaje, por lo que es buena idea que tengas una base sólida del lenguaje, sobre todo en cuanto a [funciones](https://platzi.com/clases/1937-python/29635-aprendiendo-a-no-repetir-codigo-con-funciones/), al momento de profundizar e implementarlos.

Los decoradores son una forma sencilla de llamar [funciones de orden mayor](https://platzi.com/clases/1826-java-funcional/26224-funciones-de-orden-mayor/), es decir, funciones que toman otra función cómo parámetro y/o retornan otra función como resultado. De esta forma un decorador añade capacidades a una función sin modificarla.

Un ejemplo de esto son las llantas de un automóvil si les colocas cadenas para la nieve: aún puede andar y además extiende su funcionalidad para conducirse en otros terrenos.

## Recordando sobre funciones

Antes de abordar el tema de decoradores haremos un pequeño repaso por las funciones, las cuales retornan un valor ante la entrada de un argumento.

Analicemos este sencillo ejemplo donde una función que multiplica un número que se eleva a la tercer potencia:

def elevar\_cubo(numero):

return numero \* numero \* numero

Si damos como argumento el número 3, entonces tendremos como salida el número 27 al ejecutar la función:

>>> elevar\_cubo(3)

27

## Funciones cómo objetos de primera-clase

Otro concepto importante a tener en cuenta es que en Python las funciones son objetos de primera-clase, es decir, que pueden ser pasados y utilizados cómo argumentos al igual que cualquier otro objeto (strings, enteros, flotantes, listas, etc.).

Veamos un ejemplo donde definimos 3 diferentes funciones que utilizaremos de manera conjunta:

def presentarse(nombre):

return f"Me llamo {nombre}"

def estudiemos\_juntos(nombre):

return f"¡Hey {nombre}, aprendamos Python!"

def consume\_funciones(funcion\_entrante):

return funcion\_entrante("David")

Las primeras dos funciones son obvias en su resultado, donde nos mostrarán un mensaje con el valor de la variable nombre. La tercer función puede ser más compleja de predecir ya que toma una función cómo entrada y veamos que pasa cuando colocamos una función cómo atributo:

>>> consume\_funciones(presentarse)

'Me llamo David'

>>> consume\_funciones(estudiemos\_juntos)

'¡Hey David, aprendamos Python!'

Pongamos atención en cómo la función consume\_funciones() se escribe con paréntesis para ser ejecutada, mientras que la función presentarse y estudiemos\_juntos para solo hacer referencia a estas.

## Funciones anidadas

Al igual que los condicionales y bucles también puedes colocar funciones dentro de otra función.

Toma un minuto para analizar el siguiente código e inferir cual será el resultado de salida:

def funcion\_mayor():

print("Esta es una función mayor y su mensaje de salida.")

def librerias():

print("Algunas librerías de Python son: Scikit-learn, NumPy y TensorFlow.")

def frameworks():

print("Algunos frameworks de Python son: Django, Dash y Flask.")

frameworks()

librerias()

Si llamamos a la función funcion\_mayor tendremos la siguiente salida:

>>> funcion\_mayor()

Esta es una función mayor y su mensaje de salida.

Algunos frameworks de Python son: Django, Dash y Flask.

Algunas librerías de Python son: Scikit-learn, NumPy y TensorFlow.

Debemos considerar que las funciones anidadas dentro de funcion\_mayor no se ejecutan sino hasta que se llama esta primera, siendo muestra del [scope o alcance](https://platzi.com/clases/1764-python-cs/25241-scope-o-alcance/" \t "_blank) de las funciones y si las llamamos obtendremos un error.

En la siguiente lectura entraremos al concepto de decoradores, setters y getters , pues al entender mejor las funciones será más fácil asimilar su uso en la implementación del encapsulamiento.

----------------------

## Setters, getters y decorador property7/25

**LECTURA**

En este punto estamos comenzando a utilizar conceptos en Python que comienzan a ser más avanzados, por lo que es normal que puedan parecerte complejos o difíciles de asimilar, así que te animo a que los repases un par de veces.

Puedes tener la tranquilidad de que si bien, al inicio no los implementas en su totalidad, podrás seguir avanzando en el curso y poco a poco incorporarlos a tus proyectos donde lo más importante es que sepas que cuentas con estas herramientas.

## Entendiendo el concepto de decorador

Antes de comenzar me gustaría que analices el siguiente código:

def funcion\_decoradora(funcion):

def wrapper():

print("Este es el último mensaje...")

funcion()

print("Este es el primer mensaje ;)")

return wrapper

def zumbido():

print("Buzzzzzz")

zumbido = funcion\_decoradora(zumbido)

¿Qué pasará si llamamos a la función zumbido()? si logras determinar el resultado de salida o entenderlo con detalle, entonces podemos seguir adelante.

Lo que sucede es lo siguiente:

>>> zumbido()

Este es el último mensaje...

Buzzzzzz

Este es el primer mensaje ;)

Si no diste con el resultado no te preocupes, solo hay que analizarlo con detalle y el truco está en la línea zumbido = funcion\_decoradora(zumbido). Sucede que la función wrapper() recibió la la función zumbido() cómo parámetro y coloca su salida entre los otros dos prints.

Todo lo que sucede se conoce en programación como metaprogramación (metaprogramming), ya que una parte del programa trata de modificar a otra durante el tiempo de compilación. En tanto un decorador básicamente toma una función, le añade alguna funcionalidad y la retorna.

## Mejorando la sintaxis

Definitivamente la forma en que decoramos la función es complejo, pero afortunadamente Python lo tiene en cuenta y podemos utilizar decoradores con el símbolo @. Volviendo al mismo ejemplo de funcion\_decoradora(), podemos simplificarlo así:

@funcion\_decoradora

def zumbido():

print("Buzzzzzz")

En solo tres líneas de código tenemos el mismo resultado que escribir zumbido = funcion\_decoradora(zumbido).

## ¿Qué son getters y setters?

A diferencia de otros lenguajes de programación, en Python los getters y setters tienen el objetivo de asegurar el encapsulamiento de datos. Cómo habrás visto, si declaramos una variable privada en Python al colocar un guión bajo al inicio de esta (\_) y normalmente son utilizados para: añadir lógica de validación al momento de obtener y definir un valor y, para evitar el acceso directo al campo de una clase.

La realidad es que en Python no existen variables netamente privadas, pues aunque se declaren con un guión bajo podemos seguir accediendo a estas. En Programación Orientada a Objetos esto es peligroso, pues podemos alterar el método de alguna clase y tener efectos colaterales que afecten la lógica de nuestra aplicación.

### Clases sin getters y setters

Veamos un ejemplo con una clase que almacena un dato de distancia recorrida en millas (mi) y lo convierte a kilómetros (km):

class Millas:

def \_\_init\_\_(self, distancia = 0):

self.distancia = distancia

def convertir\_a\_kilometros(self):

return (self.distancia \* 1.609344)

Ahora creemos un objeto que haga referencia a un viaje:

**# Creamos un nuevo objeto**

avion = Millas()

**# Indicamos la distancia**

avion.distancia = 200

**# Obtenemos el atributo distancia**

>>> print(avion.distancia)

200

**# Obtenemos el método convertir\_a\_kilometros**

>>> print(avion.convertir\_a\_kilometros())

321.8688

### Utilizando getters y setters

Incluyamos un par de métodos para obtener la distancia y otro para que no acepte valores inferiores a cero, pues no tendría sentido que un vehículo recorra una distancia negativa. Estos son métodos getters y setters:

class Millas:

def \_\_init\_\_(self, distancia = 0):

self.distancia = distancia

def convertir\_a\_kilometros(self):

return (self.distancia \* 1.609344)

# Método getter

def obtener\_distancia(self):

return self.\_distancia

# Método setter

def definir\_distancia(self, valor):

if valor < 0:

raise ValueError("No es posible convertir distancias menores a 0.")

self.\_distancia = valor

El método getter obtendrá el valor de la distancia que y el método setter se encargará de añadir una restricción. También debemos notar cómo distancia fue reemplazado por \_distancia, denotando que es una variable privada.

Si probamos nuestro código funcionará, la desventaja es que cualquier aplicación que hayamos creado con una base similar deberá ser actualizado. Esto no es nada escalable si tenemos cientos o miles de líneas de código.

## Función property()

Esta función está incluida en Python, en particular crea y retorna la propiedad de un objeto. La propiedad de un objeto posee los métodos getter(), setter() y del().

En tanto la función tiene cuatro atributos: property(fget, fset, fdel, fdoc) :

* fget : trae el valor de un atributo.
* fset : define el valor de un atributo.
* fdel : elimina el valor de un atributo.
* fdoc : crea un docstring por atributo.

Veamos un ejemplo del mismo caso implementando la función property() :

class Millas:

def \_\_init\_\_(self):

self.\_distancia = 0

# Función para obtener el valor de \_distancia

def obtener\_distancia(self):

print("Llamada al método getter")

return self.\_distancia

# Función para definir el valor de \_distancia

def definir\_distancia(self, recorrido):

print("Llamada al método setter")

self.\_distancia = recorrido

# Función para eliminar el atributo \_distancia

def eliminar\_distancia(self):

del self.\_distancia

distancia = property(obtener\_distancia, definir\_distancia, eliminar\_distancia)

**# Creamos un nuevo objeto**

avion = Millas()

**# Indicamos la distancia**

avion.distancia = 200

**# Obtenemos su atributo distancia**

>>> print(avion.distancia)

Llamada al método getter

Llamada al método setter

200

Aunque en este ejemplo hay una sola llamada a print, tenemos tres líneas como salida pues esta llama a los primeros dos métodos. Por lo que la propiedad distancia es una propiedad de objeto que ayuda a mantener el acceso de forma privada.

## Decorador @property

Este decorador es uno de varios con los que ya cuenta Python, el cual nos permite utilizar getters y setters para hacer más fácil la implementación de la programación orientada a objetos en Python cambiando los métodos o atributos de las clases de forma que no modifiquemos el código.

Pero mejor veamos un ejemplo en acción:

class Millas:

def \_\_init\_\_(self):

self.\_distancia = 0

# Función para obtener el valor de \_distancia

# Usando el decorador property

@property

def obtener\_distancia(self):

print("Llamada al método getter")

return self.\_distancia

# Función para definir el valor de \_distancia

@obtener\_distancia.setter

def definir\_distancia(self, valor):

if valor < 0:

raise ValueError("No es posible convertir distancias menores a 0.")

print("Llamada al método setter")

self.\_distancia = valor

**# Creamos un nuevo objeto**

avion = Millas()

**# Indicamos la distancia**

avion.distancia = 200

**# Obtenemos su atributo distancia**

>>> print(avion.definir..distancia)

Llamada al método getter

Llamada al método setter

200

De esta manera usamos el decorador @property para utilizar getters y setters de una forma más prolija e incluimos una nueva funcionalidad a nuestro método definir\_distancia() , al mismo tiempo protegemos el acceso a nuestras variables privadas y cumplimos con el principio de encapsulación.

Más detalles acá: <https://pythones.net/decoradores-en-python-oop/>

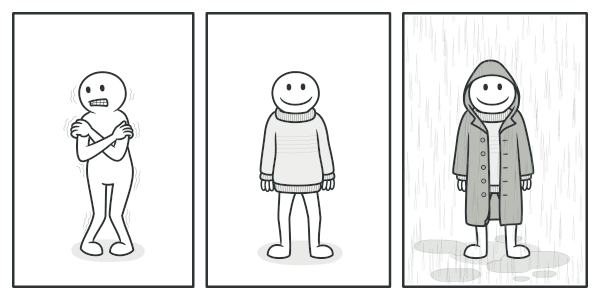
-------------------------------

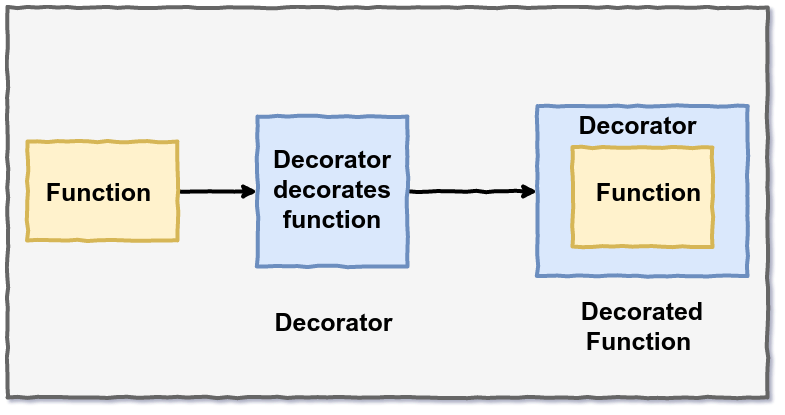
## Encapsulación, getters and setters

Texto

Descripción generada automáticamente

**Si todavía no te queda claro el concepto de decorators**

  
**Un Decorator**: Usar ropa es un ejemplo del uso de decoradores en la vida real. Cuando tienes frío, te envuelves en un suéter. Si todavía tienes frío con un suéter, puedes usar una chaqueta en la parte superior. Si está lloviendo, puedes ponerte un impermeable. Todas estas prendas “amplían” tu comportamiento básico, pero no son parte de ti, y pueden quitarse fácilmente cuando no las necesites.

  
**Un Decorator en código**: No es más que un objeto que agrega nuevas funcionalidades al objeto original.

## @property

**En Python**, **@property** es uno de estos **decorators**. Es una función integrada a Python y devuelve un objeto de tipo **property** con los metodos **Getters y Setters**.

## Todo esto y más en la fuente:

* Decorators --> <https://refactoring.guru/design-patterns/decorator>
* Python @property decorator --> <https://www.programiz.com/python-programming/property>

-----------------------------------

## Herencia

Diagrama

Descripción generada automáticamente

Un aspecto importantísimo de la Herencia es la posibilidad de reutilizar código. La reutilización es uno de los pilares de la POO, de manera que evitemos reinventar la rueda cada vez. Si tenemos un comportamiento que es común entre una serie de objetos de la misma categoría, este comportamiento debe enviarse a un clase superior que permita compartirlo con sus clases hijas. Esto facilita la mantenibilidad del código haciéndolo más estable.

-----------------------------------------------

## Polimorfismo

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente