La pregunta en realidad no es específica de Kivy sino de Python y del paradigma de la POO (Programación orientada a objetos) en general.

La respuesta corta sería que aunque el método **\_\_init\_\_**es a veces llamado "constructor" **en realidad es solo un inicializador de la instancia**, se usa generalmente para ***inicializar los atributos del objeto*** que creamos y se ejecuta de forma automática nada más instanciar la clase. En tu caso, como dentro de él llamamos al método showtext el texto ya está cargado en el label cuando nuestra app se inicia.

**super** se usa en este caso para llamar al inicializador de la clase padre BoxLayout. En este caso es equivalente a BoxLayout.\_\_init\_\_(self, \*args, \*\*kwargs).

Si nos ponemos estrictos, el término 'constructor' en Python puede ser algo ambiguo. Cuando creamos un objeto se implican dos métodos especiales, lo primero que se llama es el método **\_\_new\_\_** y posteriormente se llama al método **\_\_init\_\_:**

* **\_\_new\_\_**: básicamente lo que hace es retornar una instancia válida de la clase. Solo **construye un objeto de su clase y la retorna**, por lo que éste debería ser considerado el verdadero "constructor" de la clase. Recibe una referencia a la clase como primer argumento (cls).

Sobrescribir este método no es tan común como en el caso de \_\_init\_\_, generalmente, su uso es personalizar la instanciación de la clase, por ejemplo es una de las formas de crear un singleton:

class Singleton:

\_instance = None

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not cls.\_instance:

cls.\_instance = super().\_\_new\_\_(cls, \*args, \*\*kwargs)

return cls.\_instance

>>> a = Singleton()

>>> b = Singleton()

>>> a

<\_\_main\_\_.Singleton object at 0x7f79ba4022e8>

>>> b

<\_\_main\_\_.Singleton object at 0x7f79ba4022e8>

Si sobrescribimos el método debemos obtener la nueva instancia llamando al \_\_new\_\_ del padre (que proviene de la clase object de la que deriva toda clase de *nuevo estilo*), por ejemplo:

instance = super().\_\_new\_\_(cls, \*args, \*\*kwargs)

* **\_\_init\_\_**: si \_\_new\_\_ retorna una instancia de su clase (lo normal) seguidamente se ejecuta implícitamente el método \_\_init\_\_ al que le pasa la instancia recien creada como primer argumento (self por convención). Si \_\_new\_\_ no retorna una instancia de la clase no es llamado y hay que hacerlo de forma explícita. Algunas de sus características son:
  + No puede ser llamado nuevamente.
  + No puede retorna nada.
  + Puede recibir parámetros que se utilizan normalmente para inicializar atributos de instancia.
  + Es opcional.

En muchos lenguajes ambos métodos están unidos en uno y es considerado como constructor. Te puedes encontrar peleas dialécticas al respecto por ahí, hay una tendencia importante a llamar constructor al \_\_init\_\_ aunque no sea estrictamente correcto.

El objetivo fundamental de \_\_init\_\_ es inicializar los atributos del objeto que creamos:

class Círculo:

pi = 3.1415

def \_\_init\_\_(self, radio: float):

self.radio = radio

def obtener\_área(self) -> float:

print(self.pi \* self.radio \*\* 2)

>>> c = Círculo(4)

>>> c.obtener\_área()

50.264

El \_\_init\_\_ en este caso inicializa el atributo de instancia radio, propiedad de todo objeto crea el Círculo y que lo diferencia de otros círculos.

Podemos crear un atributo fuera del \_\_init\_\_ o de cualquier método como puedes haber observado, este es el caso de pi en el ejemplo anterior. Hay una diferencia sustancial en ambos casos, el atributo pi es un atributo que pertenece a la clase y compartido por todas las instancias de la clase que hagamos. radio en cambio es creado por \_\_init\_\_ para la instancia, pertenece al objeto en exclusiva inicializándola y dándole unas propiedades diferenciadoras. Para más información la siguiente pregunta puede ser de ayuda:

[Diferencia entre atributos de instancia y atributos de clase](https://es.stackoverflow.com/q/132561/15089)

Ya para terminar esto, hablar de [super](https://docs.python.org/3/library/functions.html" \l "super) ya que es usado comúnmente en el inicializador de la clase, cómo en el ejemplo que muestras. Esto se debe a que al definir nuestro propio inicializador sobrescribimos el \_\_init\_\_ de la clase padre. Para que nuestra clase herede todas las características de su padre implementadas en su inicializador (el cual hemos sobrescrito) es necesario por tanto llamarlo de forma explícita. De esto se encarga precisamente super(MyWidget, self).\_\_init\_\_().

*En general permite****hacer referencia explícita a la clase padre****, por lo que permite delegar la llamada de un método a la clase (o clases) padre.*

class A():

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

print("Soy el \_\_init\_\_ de la clase A")

class B(A):

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

super(B, self).\_\_init\_\_()

print ("Soy el \_\_init\_\_ de la clase B")

obj = B()

Salida:

Soy el \_\_init\_\_ de la clase A  
Soy el \_\_init\_\_ de la clase B

En casos de herencia múltiple es dónde super tiene su potencial verdadero ya que va a buscar el método entre las clases padre en un orden determinada (*Method Resolution Order*), pero esto ya está muy alejado de la pregunta.

Nota: en Python 3 no es necesario pasar las referencias a la clase y a la instancia a super: super().\_\_init\_\_()