Creado por:

Isabel Maniega

Clases y Funciones

Nos permiten ordenar el código, como por ejemplo código repetido, etc

Ejemplo 1: 1 Clase - 2 funciones

In [1]:
```
class Clase1:

    def funcion1():
        print("Estamos en: Clase1 - Funcion1")

    def funcion2():
        print("Estamos en: Clase1 - Funcion2")
```

In [2]:
```
# funcion1() --> no funciona, no encuentra la funcion
```

In [3]:
```
# se define primero la clase + función
Clase1.funcion1()
```

```
Estamos en: Clase1 - Funcion1
```

In [4]:
```
Clase1.funcion2()
```

```
Estamos en: Clase1 - Funcion2
```

Ejemplo 1

In [5]:
```
class Clase1:

    def imprimir(y):
        print("Estamos en: ")
        print("Clase Python - funcion imprimir, argumento y: ", y)

    def funcion_suma():
        x = 1
        z = 3
        y = x + z

        print(f"La suma de {x} más {z} es {y}")
```

In [7]:
```
# con argumento
Clase1.imprimir(6)
```

```
Estamos en:
Clase Python - funcion imprimir, argumento y:  6
```

In [8]:
```
# sin argumento
Clase1.funcion_suma()
```

```
La suma de 1 más 3 es 4
```

Ejemplo 2

In [9]:
```
class Clase1:

    def funcion1():
        print("Estamos en: Clase1 - Funcion1")

    def funcion2():
        print("Estamos en: Clase1 - Funcion2")
        print("\n")
        funcion1() # No accede a la función se necesita declarar la clase
```

In [10]:
```
Clase1.funcion1()
```

```
Estamos en: Clase1 - Funcion1
```

In [11]:
```
Clase1.funcion2()
```

```
Estamos en: Clase1 - Funcion2
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last
t)
Cell In[11], line 1
----> 1 Clase1.funcion2()

Cell In[9], line 9, in Clase1.funcion2()
      7 print("Estamos en: Clase1 - Funcion2")
      8 print("\n")
----> 9 funcion1()

NameError: name 'funcion1' is not defined
```

In [12]:
```
class Clase1:

    def funcion1():
        print("Estamos en: Clase1 - Funcion1")

    def funcion2():
        print("Estamos en: Clase1 - Funcion2")
        print("\n")
        Clase1.funcion1()
```

In [13]:
```
Clase1.funcion2()
```

```
Estamos en: Clase1 - Funcion2


Estamos en: Clase1 - Funcion1
```

Ejemplo 3

In [14]:
```
class Clase1:

    def imprimir(y):
        print("Estamos en: ")
        print("Clase Python - funcion imprimir, argumento y: ", y)
```

```
    def funcion_suma():
        x = 1
        z = 3
        y = x + z

        Clase1.imprimir(y)
```

In [15]:
```
Clase1.funcion_suma()
```

```
Estamos en:
Clase Python - funcion imprimir, argumento y:  4
```

In [16]:
```
Clase1.imprimir(10)
```

```
Estamos en:
Clase Python - funcion imprimir, argumento y:  10
```

2 Clases con la misma función

- Una misma función puede estar en varias clases diferentes

Dado que tenemos que llamar previamente a la clase...

entonces:

- Sólo se ejecutará la funcion dentro de la clase

In [17]:
```
class Clase1:

    def funcion1():
        print("Estamos en: Clase1 - Funcion1")

class Clase2:

    def funcion1():
        print("Estamos en: Clase2 - Funcion1")
```

In [18]:
```
Clase1.funcion1()
```

```
Estamos en: Clase1 - Funcion1
```

In [19]:
```
Clase2.funcion1()
```

```
Estamos en: Clase2 - Funcion1
```

Ejemplo

In [20]:
```
class Frutas:
    def suma(x, y):
        suma = x + y
        print("la suma de frutas que tenemos es: ", suma)

class Verduras:
    def suma(x, y):
        suma = x + y
        print("la suma de verduras que tenemos es: ", suma)
```

In [21]:
```
Frutas.suma(3, 2)
```

```
la suma de frutas que tenemos es:  5
```

In [22]:
```
Verduras.suma(6, 2)
```

```
la suma de verduras que tenemos es:  8
```

1 clase que recibe otra clase

In [23]:
```
class Clase1:
    def funcion1():
        print("Estamos en: Clase1 - Funcion1")

class Clase2(Clase1):
    def funcion2():
        print("Estamos en: Clase2 - Funcion2")

    # def funcion1():
        # print("Estamos en: Clase1 - Funcion1")
```

In [24]:
```
Clase1.funcion1()
```

```
Estamos en: Clase1 - Funcion1
```

In [25]:
```
Clase2.funcion2()
```

```
Estamos en: Clase2 - Funcion2
```

In [26]:
```
# heredado con la clase --> HERENCIA
Clase2.funcion1()
```

```
Estamos en: Clase1 - Funcion1
```

Ejemplo práctico

In [27]:
```
class Frutas:
    def manzanas():
        suma_manzanas = 6
        return suma_manzanas

class Alimentos(Frutas):
    def peras():
        suma_peras = 7
        return suma_peras
    #def manzanas():
    #    suma_manzanas = 6
    #    return suma_manzanas
```

In [28]:
```
Frutas.manzanas()
```

Out[28]: 6

In [29]:
```
Alimentos.peras()
```

Out[29]: 7

In [30]:
```
Alimentos.manzanas()
```

Out[30]: 6

También podemos "llamar" variables

In [1]:
```
class Clase1:
    x = 5

class Clase2:
    def funcion():
        print("Estamos en la Clase2")
        print("\n")
        print("Ahora llamamos a la variable de la Clase1")
        print("Valor: ", Clase1.x)
        # print("Valor: ", x) # Necesario añadir la clase, sino error de
```

In [2]:
```
Clase2.funcion()
```

```
Estamos en la Clase2


Ahora llamamos a la variable de la Clase1
Valor:  5
```

In [3]:
```
class Clase1:
    x = 5

class Clase2:
    def funcion():
        x = 6
        print("Estamos en la Clase2")
        print("\n")
        print("Ahora llamamos a la variable de la Clase1")
        print("Valor: ", Clase1.x)
        print("Valor: ", x) # Necesario añadir la clase, sino error de va
```

In [4]:
```
Clase2.funcion()
```

```
Estamos en la Clase2


Ahora llamamos a la variable de la Clase1
Valor:  5
Valor:  6
```

In [5]:
```
class Clase1:
    x = 6
    def funcion1(x):
        return x
```

In [6]:
```
Clase1.funcion1(Clase1.x)
```

Out[6]: 6

In [14]:
```
class Clase1:
    y = 6
```

```
    def funcion1(y):
        return y
```

In [15]:
```
Clase1.funcion1(y)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 Clase1.funcion1(y)

NameError: name 'y' is not defined
```

In [9]:
```
x = 6
def funcion1(x):
    return x
```

In [10]:
```
funcion1(x)
```

Out[10]: 6

Decoradores

Cierres/Closures

Cierres es una técnica que permite almacenar valores a pesar de que el contexto en el que se crearon ya no existe.

In [16]:
```
def outer(par):
    loc = par


var = 1
outer(var)

print(var)
print(loc)
```

```
1
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[16], line 9
      6 outer(var)
      8 print(var)
----> 9 print(loc)

NameError: name 'loc' is not defined
```

Las dos últimas líneas provocarán una excepción NameError - ni par ni loc son accesibles fuera de la función. Ambas variables existen cuando y solo cuando la función exterior() esta siendo ejecutada.

In [17]:
```
def outer(par):
    loc = par

    def inner():
        return loc
    return inner


var = 1
fun = outer(var)
print(fun())
```

1

Hay un elemento completamente nuevo - una función (llamada inner) dentro de otra función (llamada outer).

¿Cómo funciona? Como cualquier otra función excepto por el hecho de que inner() solo se puede invocar desde dentro de outer(). Podemos decir que inner() es una herramienta privada de outer(), ninguna otra parte del código la puede acceder.

Observa cuidadosamente:

La función inner() devuelve el valor de la variable accesible dentro de su alcance, ya que interior() puede utilizar cualquiera de las entidades a disposición de outer(). La función outer() devuelve la función inner() por si misma; mejor dicho, devuelve una copia de la función inner() al momento de la invocación de la función outer(); la función congelada contiene su entorno completo, incluido el estado de todas las variables locales, lo que también significa que el valor de loc se retiene con éxito, aunque outer() ya ha dejado de existir.

La función devuelta durante la invocación de outer() es un cierre.

Un cierre se debe invocar exactamente de la misma manera en que se ha declarado.

In [18]:
```
def outer(par):
    loc = par

    def inner():
        return loc
    return inner # La función inner() no tenía parámetros, por lo que tuv


var = 1
fun = outer(var)
print(fun())
```

1

In [19]:
```
def make_closure(par):
    loc = par
    print('valor de loc: ', loc)

    def power(p):
        print('valor de p: ', p)
        return p ** loc
```

```
    return power


fsqr = make_closure(2)
fcub = make_closure(3)

for i in range(5):
    print(i, fsqr(i), fcub(i))
```

```
valor de loc:  2
valor de loc:  3
valor de p:  0
valor de p:  0
0 0 0
valor de p:  1
valor de p:  1
1 1 1
valor de p:  2
valor de p:  2
2 4 8
valor de p:  3
valor de p:  3
3 9 27
valor de p:  4
valor de p:  4
4 16 64
```

Es totalmente posible declarar un cierre equipado con un número arbitrario de parámetros, por ejemplo, al igual que la función power().

Esto significa que el cierre no solo utiliza el ambiente congelado, sino que también puede modificar su comportamiento utilizando valores tomados del exterior.

Este ejemplo muestra una circunstancia más interesante: puedes crear tantos cierres como quieras usando el mismo código. Esto se hace con una función llamada make_closure(). Nota:

- El primer cierre obtenido de make_closure() define una herramienta que eleva al cuadrado su argumento.
- El segundo está diseñado para elevar el argumento al cubo.

Decoradores

Los decoradores permiten reducir las líneas de código duplicadas, hacer que nuestro código sea legible, fácil de testear, fácil de mantener.

Lo primero que decir que una función se puede asignar como una variable, como vemos en el siguiente ejemplo:

In [20]:
```
def hola():
    print('Hola soy una función')

def super_fun(funcion):
    funcion()
```

```
funcion = hola  # Asignamos la función a una variable!

super_fun(funcion)
```

```
Hola soy una función
```

Un decorador no es más que una función la cual toma como input una función y a su vez retorna otra función.

Al momento de implementar un decorador estaremos trabajando, con por lo menos, 3 funciones. El input, el output y la función principal. Para que nos quede más en claro a mi me gusta nombrar a las funciones como: a, b y c.

Donde 'a' recibe como parámetro 'b' para dar como salida a 'c'. Esta es una pequeña "formula" la cual me gusta mucho mencionar.

a(b) -> c

Veamos un ejemplo de como crear un decorador en Python.

In [21]:
```
def funcion_a(funcion_b):
    def funcion_c():
        print('Antes de la ejecución de la función a decorar')
        funcion_b()
        print('Después de la ejecución de la función a decorar')

    return funcion_c
```

Ya tenemos el decorador creado, ahora lo que nos hace falta es decorar una función. Al nosotros utilizar la palabra decorar estamos indicando que queremos modificar el comportamiento de una función ya existente, pero sin tener que modificar su código.

Para decorar una función basta con colocar, en su parte superior de dicha función, el decorador con el prefijo @.

In [22]:
```
@funcion_a
def hola():
    print('Hola mundo!!')

hola()
```

```
Antes de la ejecución de la función a decorar
Hola mundo!!
Después de la ejecución de la función a decorar
```

Ahora usaremos el ejemplo usada en los cierres para realizar un ejemplo más complejo de decorador, en este ejemplo vamos a crear un listado de números elevados al cuadrado o al cubo cada uno de ellos:

- Se declara la función make_closure que recoge las variables para elevar al cuadrado o al cubo cada uno de los valores, recoge los valores de elevado y el valor a elevar.
- la función function que retorna los valores con los que tiene que trabajar la función make_closure.

Observemos el ejemplo:

In [23]:
```
num = 5

def make_closure(func):
    def power(*args, **kwargs):
        loc = args[0]
        p = args[1]
        # print('valor de loc: ', loc)
        # print('valor de p: ', p)
        return p ** loc
    return power

@make_closure
def function(par, p):
    return par, p

range_sqr = [function(2, i) for i in range(num)]

print('listas de números al cuadrado: ', range_sqr)
print('\n')

range_cub = [function(3, i) for i in range(num)]

print('lista de numeros al cubo: ', range_cub)
```

```
listas de números al cuadrado:  [0, 1, 4, 9, 16]


lista de numeros al cubo:  [0, 1, 8, 27, 64]
```

In [24]:
```
#  Usando map:

num = 5

def make_closure(func):
    def power(*args, **kwargs):
        loc = args[0]
        p = args[1]
        # print('valor de loc: ', loc)
        # print('valor de p: ', p)
        return p ** loc
    return power


@make_closure
def function(par, p):
    return par, p

def range_sqr(i):
    return function(2, i)

range_list = range(num)
values_list = list(map(range_sqr, range_list))

print('listas de números al cuadrado: ', values_list)
```

```
listas de números al cuadrado:  [0, 1, 4, 9, 16]
```

Creado por:

Isabel Maniega