Tramo III - POO –

clase 1:



Las funciones o expresiones lambda sirven para crear funciones anónimas.

Una función anónima, como su nombre indica es una función sin nombre. En Python podemos ejecutar una función sin definirla con def. De hecho son similares pero con una diferencia fundamental: El contenido de una función lambda debe ser una única expresión en lugar de un bloque de acciones. Y es que más allá del sentido de función que tenemos, con su nombre y sus acciones internas, una función en su sentido más trivial significa realizar algo sobre algo. Por tanto podríamos decir que, mientras las funciones anónimas lambda sirven para realizar funciones simples, las funciones definidas con def sirven para manejar tareas más extensas. Si deconstruimos una función sencilla, podemos llegar a una función lambda.

Por ejemplo la siguiente función para doblar un valor:

def duplicar(num):

    resultado = num\*2

    return resultado

print(duplicar(2)) # resultado: 4

Vamos a simplificar el código un poco:

def duplicar(num):

    return num\*2

print(duplicar(2))

Todavía más, podemos escribirlo todo en una sola línea:

def duplicar(num): return num\*2

print(duplicar(2))

Esta notación simple es la que una función lambda intenta replicar.

Vamos a convertir la función en una función anónima:

lambda num: num\*2

Aquí tenemos una función anónima con una entrada que recibe num, y una salida que devuelve num \* 2. Lo único que necesitamos hacer para utilizarla es guardarla en una variable y utilizarla tal como haríamos con una función normal:

duplicar = lambda num: num\*2

print(duplicar(2))

Gracias a la flexibilidad de Python podemos implementar infinitas funciones simples.

Por ejemplo comprobar si un número es impar:

impar = lambda num: num%2 != 0

print(impar(5)) # resultado: True

Darle la vuelta a una cadena utilizando slicing:

revertir = lambda cadena: cadena[::-1]

print(revertir("Hola")) # resultado: aloH

Incluso podemos enviar varios valores, por ejemplo para sumar dos números:

sumar = lambda x,y: x+y

print(sumar(5,2)) # resultado: 7

cuadrado = lambda n: n\*n

print(cuadrado(2)) # resultado: 4

resto = lambda x,y : x%y

print(resto(5,2)) # resultado: 1

Podemos realizar cualquier cosa que se nos ocurra, siempre que lo podamos definir en una sola expresión.

def ejecutar(funcion,valor):

    return funcion(valor)

print("CUADRADO  : ",ejecutar(lambda x: x\*x,    2))

print("CUBO 3    : ",ejecutar(lambda x: x\*x\*x  ,3))

print("esPar el 5: ",ejecutar(lambda x: x%2==0 ,5))

print("esPar el 6: ",ejecutar(lambda x: x%2==0 ,6))

def operar(funcion,x,y):

    return funcion(x,y)

print("sumar 2+3 : ",operar(lambda x,y: x+y,2,3))

print("multr 2\*3 : ",operar(lambda x,y: x\*y,2,3))

print("restr 2-3 : ",operar(lambda x,y: x-y,2,3))

print("divir 2/3 : ",operar(lambda x,y: x/y,2,3))

print(f"divir 2/3 : {operar(lambda x,y: x/y,2,3):.2f}")



Tal como su nombre indica filter significa filtrar.

A partir de una lista o iterador y una función condicional, es capaz de devolver una nueva colección con los elementos filtrados que cumplan la condición.

Por ejemplo:

Supongamos que tenemos una lista varios números y queremos filtrarla, quedándonos únicamente con los múltiples de 5...

def es\_multiplo(numero):

    return numero%5 == 0

numeros = [2, 5, 10, 23, 50, 33]

fil = filter(es\_multiplo, numeros)

print(fil)

print(type(fil))

print(str(fil))

print(list(fil))

<filter object at 0x000001FDD09BB940>

<class 'filter'>

<filter object at 0x000001FDD09BB940>

[5, 10, 50]

Si ejecutamos el filtro obtenemos un objeto de tipo filtro, pero podemos transformarlo en una lista fácilmente haciendo un cast (conversión):

numeros = [2, 5, 10, 23, 50, 33]

lista = list(filter(es\_multiplo, numeros) )

print(lista)

Por tanto cuando utilizamos la función filter() tenemos que enviar una función condicional. Podemos no definir la función usando una función anónima lambda:

lista\_numeros = [2, 5, 10, 23, 50, 33]

lista = list( filter(lambda numero: numero%5 == 0, lista\_numeros) )

print(lista)

print(list(filter(lambda numero: numero%5 == 0, lista\_numeros)))

def filtrar\_divisibles(lista:list,valor:int)->list:

    nueva = []

    for numero in lista:

        if numero%valor == 0:

            nueva.append(numero)

    return nueva

print(filtrar\_divisibles(lista\_numeros,5))

En los tres casos el print, retorna una lista: [5, 10, 50], o sea el mismo resultado, lo único que en la tercera opción se resuelve en 7 líneas de código y en segunda opción, con solo una línea.

Otro ejemplo:

def esPar(n):

    return n%2 == 0

lista\_numeros = [1,2,-3,10,-1,-17,4,33,-2]

print("NUMEROS: ",lista\_numeros)

negativos = list( filter(lambda x: x < 0, lista\_numeros) )

print("NEGATIVOS: ",negativos)

pares = tuple( filter(esPar,lista\_numeros) )

print("PARES: ",pares)

pares = list( filter(esPar,lista\_numeros) )

print("PARES: ",pares)

NUMEROS: [1, 2, -3, 10, -1, -17, 4, 33, -2]

NEGATIVOS: [-3, -1, -17, -2]

PARES: (2, 10, 4, -2)

PARES: [2, 10, 4, -2]



class Alumno:

    def \_\_init\_\_(self, nombre, nota):

        self.\_\_nombre = nombre

        self.\_\_nota = nota

    def \_\_str\_\_(self):

        return "Alumno: {} Nota: {} ".format(self.\_\_nombre, self.\_\_nota)

    def getnota(self):

        return self.\_\_nota

a = Alumno("Raulito",7)

c = Alumno("Fernanda",10)

print(str(a))

alumnos = [

    Alumno("Victoria", 10),

    Alumno("Manu",7),

    Alumno("Sergio", 5),

    Alumno("Maria", 6),

    Alumno("Brian", 4)

]

aprobados = filter(lambda x: x.getnota() >= 6, alumnos)

for a in aprobados:

    print(a)

Alumno: Raulito Nota: 7

Alumno: Victoria Nota: 10

Alumno: Manu Nota: 7

Alumno: María Nota: 6

class Producto(object):

    def \_\_init\_\_(self,codigo,grupo,nombre,precio):

        self.\_\_codigo=codigo

        self.\_\_grupo=grupo

        self.\_\_nombre=nombre

        self.\_\_precio=precio

    def \_\_repr\_\_(self):

        return f' tipo: {type(self)} Atributos  ==> {self.\_\_codigo},{self.\_\_nombre},{self.\_\_grupo},{self.\_\_precio}'

    def \_\_str\_\_(self):

        return f"{self.\_\_codigo} - {self.\_\_nombre} - {self.\_\_grupo} - {self.\_\_precio}"

    def get\_grupo(self):

        return self.\_\_grupo

lp=[

        Producto(1,1,"Arvejas",32),

        Producto(2,1,"Lentejas",29),

        Producto(3,2,"Leche",59)

]

print("\nArticulos:", lp)

almacen = list(filter(lambda x: x.get\_grupo() == 1,lp))

print("\nAlmacen:\n")

for a in almacen:

    print(a)



Esta función trabaja de una forma muy similar a filter(), con la diferencia que en lugar de aplicar una condición a un elemento de una lista o secuencia, aplica una función sobre todos los elementos y como resultado se devuelve un iterable de tipo map:

def doblar(numero):

    return numero\*2

numeros = [2, 5, 10, 23, 50, 33]

m = map(doblar, numeros)

lista = list(m)

print(numeros)

print(m)

print(type(m))

print(lista)

print(numeros)

[2, 5, 10, 23, 50, 33]

<map object at 0x000001E31C08B880>

<class 'map'>

[4, 10, 20, 46, 100, 66]

[2, 5, 10, 23, 50, 33]

Y podemos simplificarlo con una función lambda para substituir la llamada de una función definida:

print(numeros)

print(list( map(lambda x: x\*2, numeros)))

print(numeros)

numeros = list(( map(lambda x: x\*2, numeros)))

print(numeros)

[2, 5, 10, 23, 50, 33]

[4, 10, 20, 46, 100, 66]

[2, 5, 10, 23, 50, 33]

[4, 10, 20, 46, 100, 66]

La función map() se utiliza mucho junto a expresiones lambda ya que permite ahorrarnos el esfuerzo de crear bucles for.

Además se puede utilizar sobre más de un iterable con la condición que tenga la misma longitud.

Por ejemplo si queremos multiplicar los números de dos listas:

a = [1, 2, 3, 4, 5]

b = [6, 7, 8, 9, 10]

lista = list( map(lambda x,y : x\*y, a,b) )

print(lista)

c = [11, 12, 13, 14, 15]

lista = list( map(lambda x,y,z : x\*y\*z, a,b,c) )

print(lista)

[6, 14, 24, 36, 50]

[66, 168, 312, 504, 750]

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

def cuadrado(n):

    return n\*n

def selec\_menores(n):

    if n<6:

        return n\*2

    return n

numeros = (2,3,4,5,6,7,8,9)

x = map(lambda x: x\*\*2,numeros)

print(type(x))

cuadrados1 = tuple(x)

print("cuadrados1: ",cuadrados1)

cuadrados2 = tuple(map(cuadrado,numeros))

print("cuadrados2: ",cuadrados2)

menores1 = tuple( map (lambda x: x>6 and x<8    ,numeros))

print("menores1: ",menores1)

menores2 = tuple(map(selec\_menores,numeros))

print("menores2: ",menores2)

print("TUPLA ORIGINAL: ",numeros)

cuadrados1: (4, 9, 16, 25, 36, 49, 64, 81)

cuadrados2: (4, 9, 16, 25, 36, 49, 64, 81)

menores1: (False, False, False, False, False, True, False, False)

menores2: (4, 6, 8, 10, 6, 7, 8, 9)

TUPLA ORIGINAL: (2, 3, 4, 5, 6, 7, 8, 9)



Evidentemente, siempre que la utilicemos correctamente podemos mapear una serie de objetos sin ningún problema:

class Alumno:

    def \_\_init\_\_(self, nombre, nota):

        self.\_\_nombre = nombre

        self.\_\_nota = nota

    def \_\_str\_\_(self):

        return "Alumno:{} Nota:{} ".format(self.\_\_nombre, self.\_\_nota)

    def get\_nota(self):

        return self.\_\_nota

    def set\_nota(self,nota):

        self.\_\_nota = nota

alumnos = [

    Alumno("Victoria", 10),

    Alumno("Manu",7),

    Alumno("Sergio", 5),

    Alumno("Maria", 6),

    Alumno("Brian", 4)

]

def incrementar(p:Alumno)->Alumno:

    if p.get\_nota() < 10:

        p.set\_nota(p.get\_nota()+1)

    return p

lista = list(map(incrementar, alumnos))

for a in lista:

    print(a)

Aplica una función a una colección y retorna solo un valor

HAY QUE IMPORTARLA DEL PAQUETE DE FUNCIONES DE PYTHON

from functools import reduce

sumar = lambda x,y : x+y

def restar(x,y):

    return x-y

def mayor(x,y):

    if x > y:

        return x

    return y

def maximo(x,y):

    return max(x,y)

numeros = [1,2,3,4,5,66,7,8,9,10]

z = reduce(lambda x,y: max(x,y),numeros)

print("z: ",z)

producto = reduce(lambda x,y: x\*y,numeros)

print(numeros," PRODUCTO: ",producto)

suma = reduce(sumar,numeros)

print(numeros," SUMA: ",suma)

resta = reduce(restar,numeros)

print(numeros," RESTA: ",resta)

print(numeros," El mayor: ",reduce(mayor,numeros))

z: 66

[1, 2, 3, 4, 5, 66, 7, 8, 9, 10] PRODUCTO: 39916800

[1, 2, 3, 4, 5, 66, 7, 8, 9, 10] SUMA: 115

[1, 2, 3, 4, 5, 66, 7, 8, 9, 10] RESTA: -113

[1, 2, 3, 4, 5, 66, 7, 8, 9, 10] El mayor: 66



listaTuplas = [("3", "a", 4), ("1", "z", 1), ("8", "n", 9),("5", "i", 6), ("3", "v", 4)]

print("lista: ", listaTuplas)

listaTuplas\_ord = sorted(listaTuplas, key=lambda x: x[1])

print("ordenada: ", listaTuplas\_ord)

print("lista: ", listaTuplas)

lista: [('3', 'a', 4), ('1', 'z', 1), ('8', 'n', 9), ('5', 'i', 6), ('3', 'v', 4)]

ordenada: [('3', 'a', 4), ('5', 'i', 6), ('8', 'n', 9), ('3', 'v', 4), ('1', 'z', 1)]

lista: [('3', 'a', 4), ('1', 'z', 1), ('8', 'n', 9), ('5', 'i', 6), ('3', 'v', 4)]



class Perro(object):

    def \_\_init\_\_(self, nombre="perro", raza="mixto", sexo='X', edad=0):

        self.nombre = nombre

        self.raza = raza

        self.sexo = sexo

        self.edad = edad

    def get\_nombre(self):

        return nombre

    def get\_raza(self):

        return raza

    def get\_sexo(self):

        return sexo

    def get\_edad(self):

        return edad

    def \_\_str\_\_(self):

        return "{:s},{:s},{:s},{:d}".format(self.nombre, self.raza, self.sexo, self.edad)

    def \_\_repr\_\_(self):

        return "{:s},{:s},{:s},{:d}".format(self.nombre, self.raza, self.sexo, self.edad)

    def \_\_gt\_\_(self, perro):

        return self.edad > perro.edad

listaPerrors = [Perro("colita", "calle", 'F', 4), Perro("pepe", "doberman", 'M'), Perro()]

print("LISTA DE PERROS: ", listaPerrors)

print("ORDEN NATURAL: ", sorted(listaPerrors))

print("ORDEN NATURAL INVERTIDO: ", sorted(listaPerrors, reverse=True))

print("ORDEN NOMBRE + EDAD: ", sorted(listaPerrors, key = lambda x: x.nombre + str(x.edad)))

print("ORDEN NOMBRE INVERTIDO: ", sorted(listaPerrors, key=lambda x: x.nombre, reverse=True))

LISTA DE PERROS: [colita,calle,F,4, pepe,doberman,M,0, perro,mixto,X,0]

ORDEN NATURAL: [pepe,doberman,M,0, perro,mixto,X,0, colita,calle,F,4]

ORDEN NATURAL INVERTIDO: [colita,calle,F,4, pepe,doberman,M,0, perro,mixto,X,0]

ORDEN NOMBRE + EDAD: [colita,calle,F,4, pepe,doberman,M,0, perro,mixto,X,0]

ORDEN NOMBRE INVERTIDO: [perro,mixto,X,0, pepe,doberman,M,0, colita,calle,F,4]

POO programación orientada a objetos

Cuando asignamos nombre a los objetos, la primera letra va en mayúscula, para definir un objeto, nosotros, vamos a definir su clase (la idea de clase es que es un molde), un molde con el que vamos a construir los objetos, los objetos construidos con ese molde, se llaman instancias.

Cada vez que definimos un objeto mediante una clase, tenemos que ver, que es lo que el objeto tiene (atributos) y que es lo que el objeto hace (métodos).

Hay que tener en cuenta en por ejemplo, si hacemos, una clase gato (class Gato), no vamos a moldear un gato de verdad, solo vamos a definir algunos atributos que nos interesen para el programa, para definir lo que tiene, vamos a definir un método especial (def \_\_init\_\_(self) -> None: ) al que llamamos: Método Constructor.

Es el método que se va a ejecutar cuando instanciamos un objeto de esa clase, el método constructor es también llamado método init, todas las clases tienen init.

El parámetro self es la referencia al objeto actual (o sea una referencia a si mismo)

Para que se muestre un objeto con sus atributos, tengo que definir en método \_\_str\_\_(self) -> str:

Los objetos tienen métodos privados y métodos públicos, los métodos privados no se ven, ni se pueden acceder fuera del objeto.

La teoría del objeto dice: *“Cuando voy a crear una clase que va a modelar un objeto (instancia), lo que hago es declarar a los atributos privados y los métodos públicos”*

Para Python los métodos privado los escribe con dos guiones bajo antes de la palabra que representa el método y dos guiones bajo después de la palabra, ej método privado init: \_\_init\_\_ , cuando un programador Python ve eso significa que no lo va a usar, salvo dentro de la clase.

class Gato:

    # METODO CONSTRUCTOR

    def \_\_init\_\_(self,nombre='',edad=0,color='') -> None:

        # ATRIBUTOS DE INSTANCIA

        self.nombre = nombre

        self.edad = edad

        self.color = color

    def \_\_str\_\_(self) -> str:

        return f'Nombre: {self.nombre}, color: {self.color}, tiene: {self.edad} Años'

    def hacer\_ruido(self)-> str:

        return f'{str(self)} ==> Miauuu'

        #return f'{self.nombre} Miauuuuu'

def main():

    g1 = Gato()

    g1.nombre = "Michi"

    g1.edad = 3

    g1.color = 'Negro'

    g2 = Gato(5,"blanco") #"Garfield"

    print(g1)

    print(str(g2))

    print(g2.hacer\_ruido())

main()

Nombre: Michi, color: Negro, tiene: 3 Años

Nombre: 5, color: , tiene: blanco Años

Nombre: 5, color: , tiene: blanco Años ==> Miauuu

clase 2:

Cuando decía: "..que map, no se podía mostrar ",( ahora que vimos POO y el método str), map no tiene escrito en su código un método str, por eso el objeto no se sabe mostrar, pero ¿porque no le escribieron a map para que se muestre?, no se lo escribieron porque no se lo puede escribir, básicamente porque los objetos que map contiene, realmente no los contiene.

def es\_multiplo(numero):

    return numero%5 == 0

numeros = [2, 5, 10, 23, 50, 33]

fil = filter(es\_multiplo, numeros)

print("fil:",fil)

print("str:(fil)",str(fil))

print(type(fil))

print("lista",list(fil))

fil: <filter object at 0x0000020E9274B9A0>

str:(fil) <filter object at 0x0000020E9274B9A0>

<class 'filter'>

lista [5, 10, 50]

# vamos a demostrar que el objeto fil es iterable

fil = filter(es\_multiplo, numeros)

print("Primera vez")

for e in fil:

    print(e)

print(type(fil))

print("lista",list(fil))

print("Segunda vez")

for e in fil:

    print(e)

print(type(fil))

print("lista",list(fil))

Primera vez

5

10

50

<class 'filter'>

lista []

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

Segunda vez

<class 'filter'>

lista []

como puede observarse, luego de iterar la lista la primera vez, la lista, queda vacia, por eso no se muestra la segunda vez.

Hay que imaginar, si por ejemplo uno tuviera una lista de 5 millones de elementos, le aplico un filtro y obtengo una lista de 2 millones de elementos, tendría un consumo muy grande de memoria al mantener la lista y el filtro simultáneamente, entonces, el objeto filter, lo que hace es manejar un **generador**, y un generador son objetos potenciales, o sea, los objetos no están hasta que los necesito, ahora una vez que lo recorro con un iterable todos los objetos, ya no existe mas.

print("Un filtro nuevo")

for e in filter(lambda x : x<10, numeros):

    print(e)

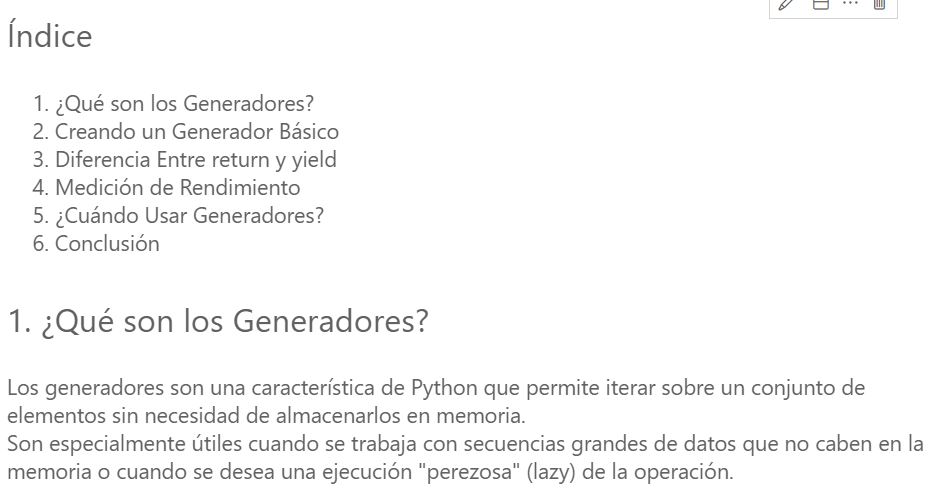
Un filtro nuevo

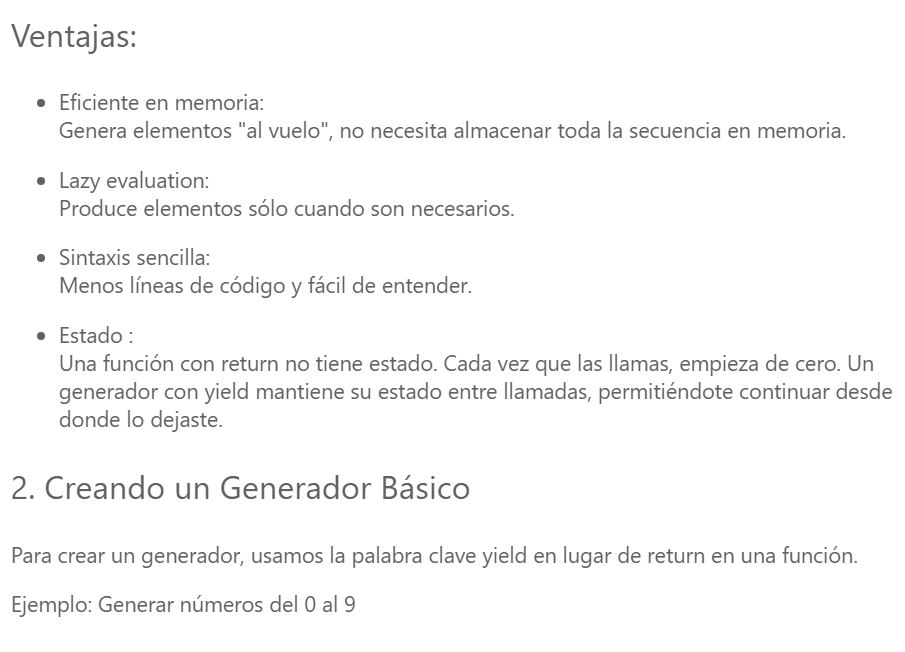
2

5

En este otro ejemplo iteramos directamente el objeto filter, ( que internamente usa un generador), o sea no guarda lo que filtro.







def generar\_numeros\_otro():

    lista = []

    num = 0

    while num < 10:

        lista.append(num)

        num += 1

    return lista

# Usando la lista

for num in generar\_numeros\_otro():

    print(num,end=' ')

print(generar\_numeros\_otro())

0 1 2 3 4 5 6 7 8 9

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

En este ejemplo, la función generar\_numeros\_otro(), genera los números del 0 al nueve y los almacena en una lista, que luego retorna, ahora esa función se puede iterar al iterar sobre la lista que la misma función creo.

def generar\_numeros():

    num = 0

    while num < 10:

        yield num

        num += 1

# Usando el generador

for num in generar\_numeros():

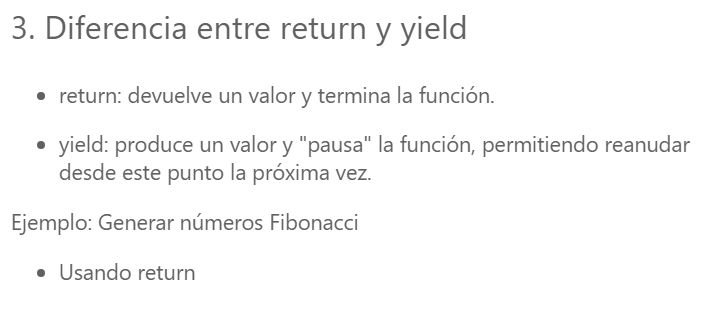
    print(num,end=' ')

print(generar\_numeros())

0 1 2 3 4 5 6 7 8 9

<generator object generar\_numeros at 0x000002DBB2A04AC0>

En este segundo ejemplo, usando el generador (yield, en lugar de return) la función generar\_numeros(),crea los números, se puede iterar, pero los números no se almacenan en ningún lado, por ende es mucho más rápida de iterar frente a colecciones de datos muy grandes.



def fibonacci\_return(n):

    numeros = []

    a, b = 0, 1

    for \_ in range(n):

        numeros.append(a)

        a, b = b, a + b

    return numeros

for x in fibonacci\_return(10):

    print(x,end=' ')

0 1 1 2 3 5 8 13 21 34



def fibonacci\_yield(n):

    a, b = 0, 1

    for \_ in range(n):

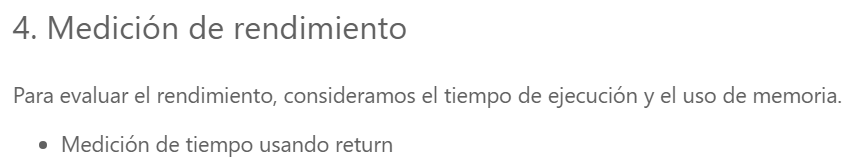
        yield a

        a, b = b, a + b

for x in  fibonacci\_yield(10):

    print(x,end=' ')

0 1 1 2 3 5 8 13 21 34



import time

import sys

def fibonacci\_return(n):

    numeros = []

    a, b = 0, 1

    for \_ in range(n):

        numeros.append(a)

        a, b = b, a + b

    return numeros

# Medir memoria y tiempo con return

inicio = time.time()

fib\_return = fibonacci\_return(100000)  # 100,000

for num in fib\_return:

    pass

fin = time.time()

print(f"Tiempo usando return: {fin - inicio} segundos")

print(f"Memoria usando return: {sys.getsizeof(fib\_return)} bytes")

Tiempo usando return: 0.47000765800476074 segundos

Memoria usando return: 800984 bytes



import time

import sys

def fibonacci\_yield(n):

    a, b = 0, 1

    for \_ in range(n):

        yield a

        a, b = b, a + b

# Medir memoria y tiempo con yield

inicio = time.time()

fib\_yield = fibonacci\_yield(100000)  # 100,000

for num in fib\_yield:

    pass

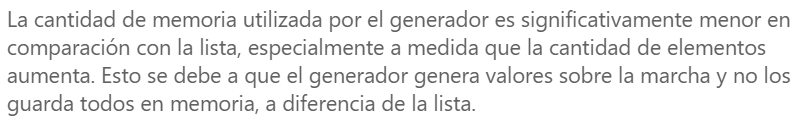
fin = time.time()

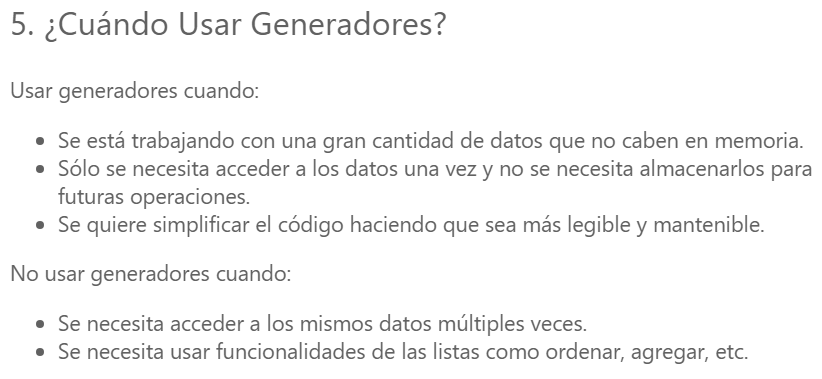
print(f"Tiempo usando yield: {fin - inicio} segundos")

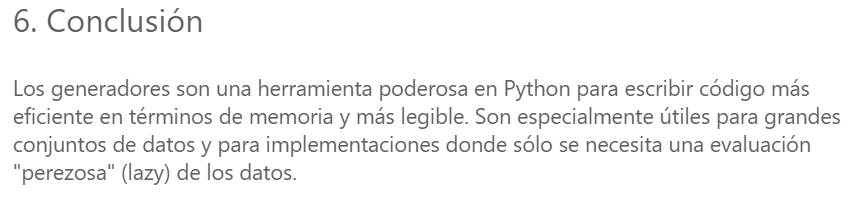
print(f"Memoria usando yield: {sys.getsizeof(fib\_yield)} bytes")

Tiempo usando yield: 0.17026591300964355 segundos

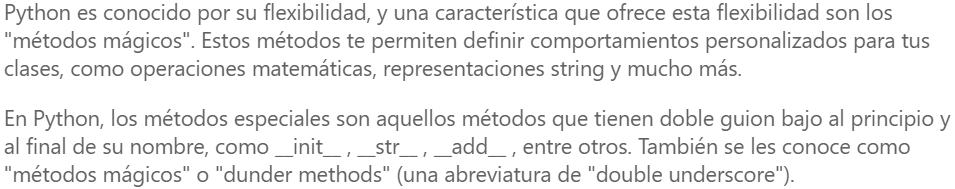
Memoria usando yield: 232 bytes

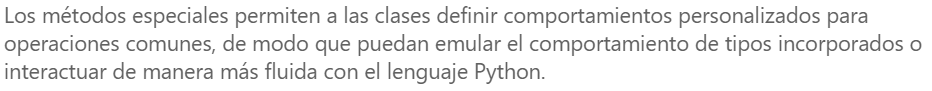


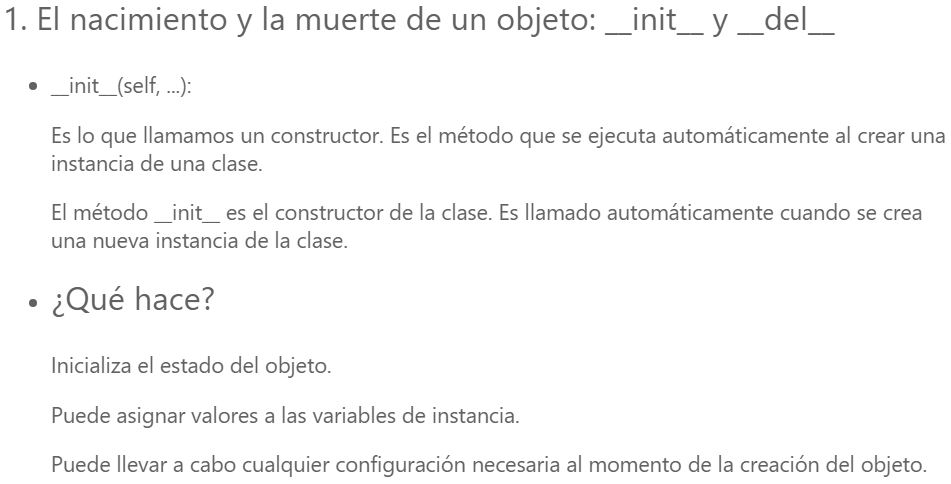


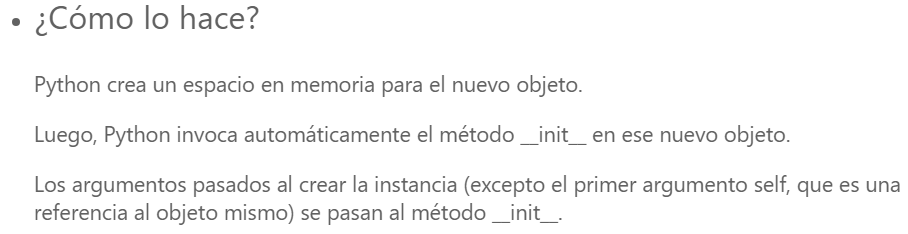












class Persona:

# Aquí, al crear una nueva Persona, le damos un nombre y una edad que se almacenan en el objeto.

    def \_\_init\_\_(self, nombre, edad):

        print("¡Se ha creado el objeto!")

        self.nombre = nombre

        self.edad = edad

p = Persona("Ana", 25)

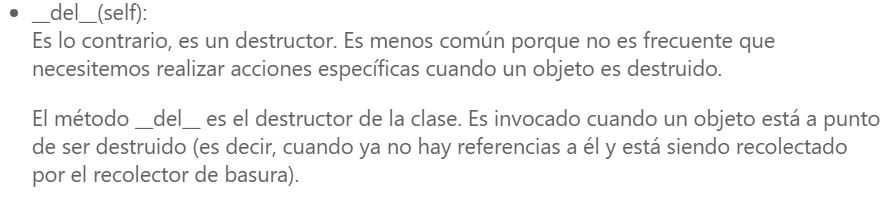
print(p.nombre)

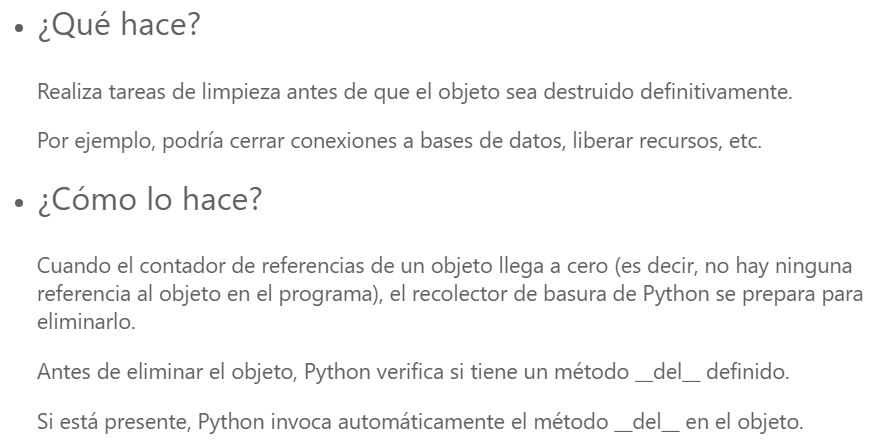
print(p.edad)

¡Se ha creado el objeto!

Ana

25





class EjemploDel:

    def \_\_init\_\_(self):

        print("¡Se ha creado el objeto!")

    def \_\_del\_\_(self):

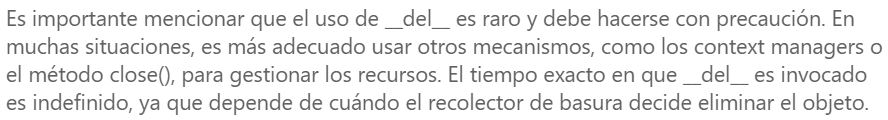
        print("¡El objeto está siendo destruido!")

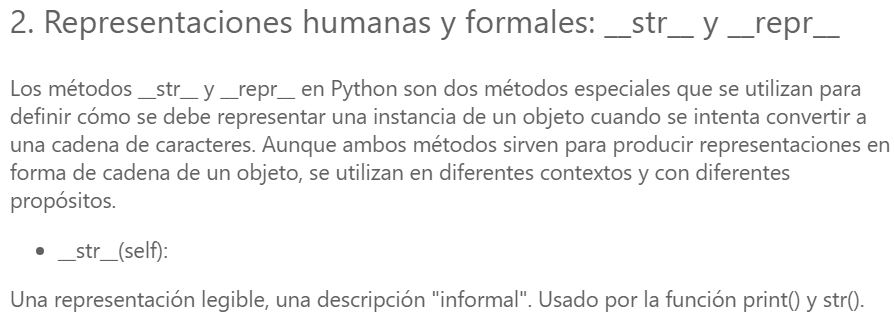
obj = EjemploDel()

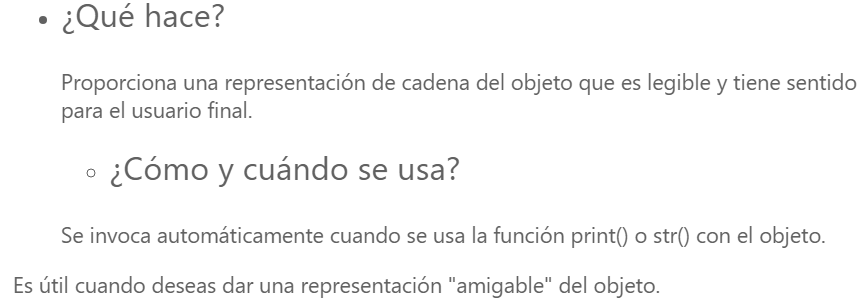
del obj  # Objeto siendo destruido

¡Se ha creado el objeto!

¡El objeto está siendo destruido!







class Persona:

    def \_\_init\_\_(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad

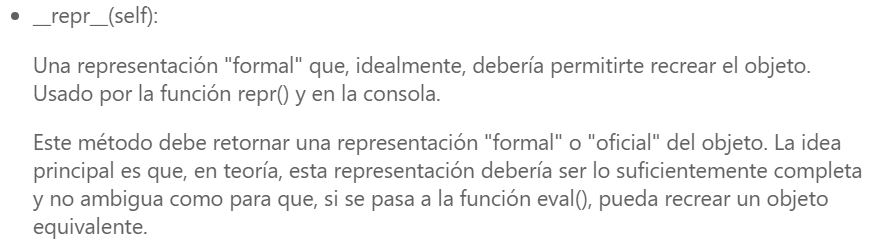
    def \_\_str\_\_(self):

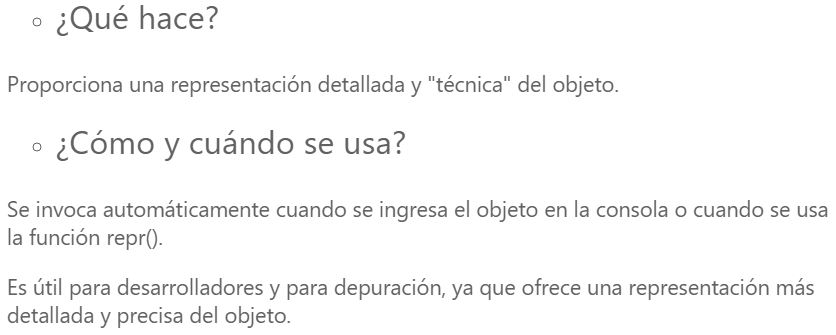
        return f"{self.nombre}, {self.edad} años"

persona = Persona("Juan", 25)

print(persona)  # Salida: Juan, 25 años

Juan, 25 años





class Persona:

    def \_\_init\_\_(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad

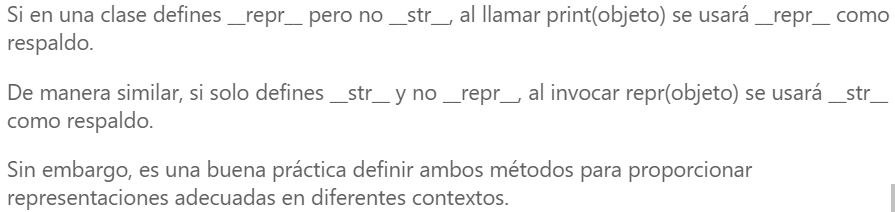
    def \_\_repr\_\_(self):

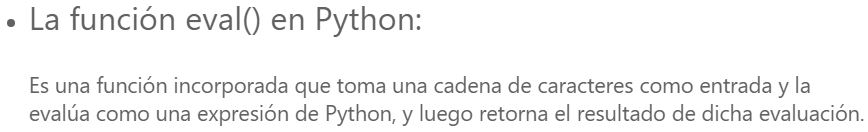
        return f"Persona('{self.nombre}', {self.edad})"

persona = Persona("Juan", 25)

print(repr(persona))  # Salida: Persona('Juan', 25)

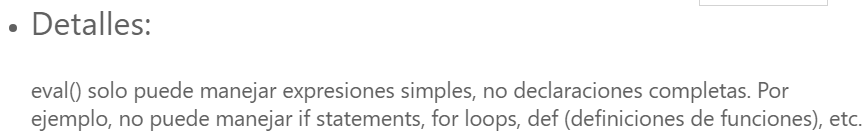
Persona('Juan', 25)

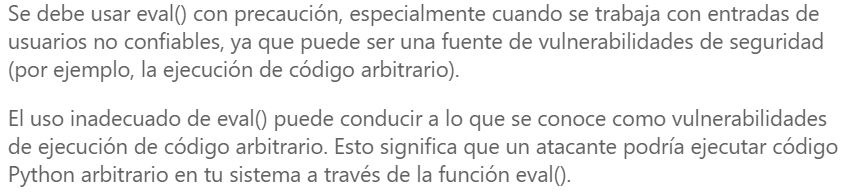




resultado = eval("3 + 4")

print(resultado)  # Salida: 7





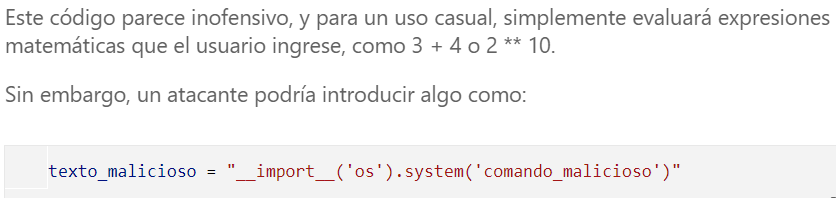


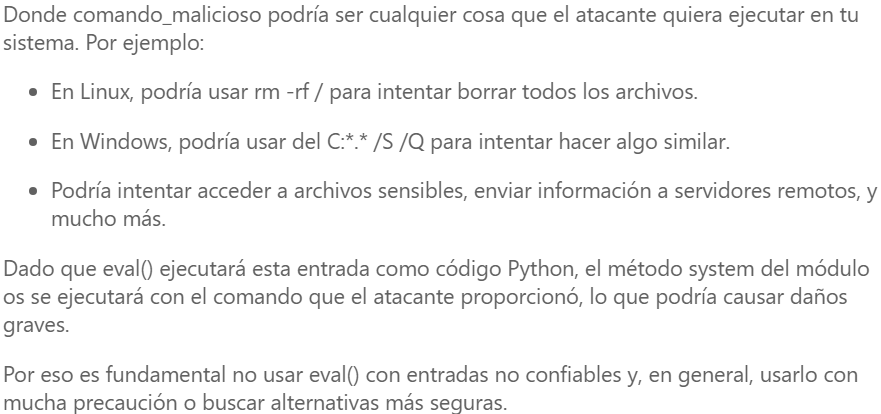
entrada\_usuario = input("Introduce una expresión para evaluar: ")

print(f'{entrada\_usuario}: {eval(entrada\_usuario)}')

Introduce una expresión para evaluar: 5\*6+35

5\*6+35: 65





class Persona:

    def \_\_init\_\_(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad

    def \_\_repr\_\_(self):

        return f"Persona('{self.nombre}', {self.edad})"

    def \_\_str\_\_(self):

        return f"{self.nombre}, {self.edad} años"

persona1 = Persona("Juan", 25)

print('str(persona1):',str(persona1))

print('repr(persona1):',repr(persona1))

persona2 = eval(repr(persona1))

print('str(persona2):',str(persona2))

print('repr(persona2):',repr(persona2))

str(persona1): Juan, 25 años

repr(persona1): Persona('Juan', 25)

str(persona2): Juan, 25 años

repr(persona2): Persona('Juan', 25)



class Fraccion:

    def \_\_init\_\_(self, numerador, denominador):

        if denominador == 0:

            raise ValueError("El denominador no puede ser cero.")

        self.numerador = numerador

        self.denominador = denominador

        # Simplificamos la fracción al crearla

        self.\_\_simplificar()

    def \_\_simplificar(self):

        """Método privado para simplificar la fracción."""

        mcd = self.\_\_mcd(self.numerador, self.denominador)

        self.numerador //= mcd

        self.denominador //= mcd

    def \_\_mcd(self, a, b):

        """Método privado para calcular el máximo común divisor."""

        while b:

            a, b = b, a % b

        return a

    def \_\_repr\_\_(self):

        """Representación formal del objeto."""

        return f"Fraccion({self.numerador}, {self.denominador})"

    def \_\_str\_\_(self):

        """Representación amigable del objeto."""

        return f"{self.numerador}/{self.denominador}"

    def \_\_add\_\_(self, other):

        """Suma de dos fracciones."""

        if not isinstance(other, Fraccion):

            raise TypeError(f"No se puede sumar Fraccion con

{type(other).\_\_name\_\_}")

        num = self.numerador \* other.denominador + other.numerador \*

self.denominador

        den = self.denominador \* other.denominador

        return Fraccion(num, den)

    def \_\_eq\_\_(self, other):

        """Determina si dos fracciones son iguales."""

        if not isinstance(other, Fraccion):

            raise TypeError(f"No se puede comparar Fraccion con

{type(other).\_\_name\_\_}")

        return self.numerador \* other.denominador == other.numerador \*

self.denominador

    def \_\_ne\_\_(self, other):

        """Determina si dos fracciones no son iguales."""

        if not isinstance(other, Fraccion):

            raise TypeError(f"No se puede comparar Fraccion con

{type(other).\_\_name\_\_}")

        return not self.\_\_eq\_\_(other)

    def \_\_lt\_\_(self, other):

        """Determina si una fracción es menor que otra."""

        if not isinstance(other, Fraccion):

            raise TypeError(f"No se puede comparar Fraccion con

{type(other).\_\_name\_\_}")

        return self.numerador \* other.denominador < other.numerador \*

self.denominador

    def \_\_le\_\_(self, other):

        """Determina si una fracción es menor o igual que otra."""

        if not isinstance(other, Fraccion):

            raise TypeError(f"No se puede comparar Fraccion con

{type(other).\_\_name\_\_}")

        return self.numerador \* other.denominador <= other.numerador \*

self.denominador

    def \_\_gt\_\_(self, other):

        """Determina si una fracción es mayor que otra."""

        if not isinstance(other, Fraccion):

            raise TypeError(f"No se puede comparar Fraccion con

{type(other).\_\_name\_\_}")

        return self.numerador \* other.denominador > other.numerador \*

self.denominador

def \_\_ge\_\_(self, other):

        """Determina si una fracción es mayor o igual que otra."""

        if not isinstance(other, Fraccion):

            raise TypeError(f"No se puede comparar Fraccion con

{type(other).\_\_name\_\_}")

        return self.numerador \* other.denominador >= other.numerador \*

self.denominador

# Ejemplo de uso:

f1 = Fraccion(3, 26)

print(f1)

f2 = Fraccion(1, 2)

print(f2)

print(f1 == f2)

f3 = f1 + f2

print(f3)

f4 = Fraccion(1,8)

print(f4)

3/26

1/2

False

8/13

1/8

"""

Implementar la clase Caramelera, que recibe en su constructor la cantidad de caramelos que puede contener, y tiene el siguiente comportamiento:

>>> c = Caramelera(20)            >>> c.sacar\_caramelos(50)

>>> c.poner\_caramelos(10)        Traceback (most recent call last):

>>> c.sacar\_caramelos(4)         ...

>>> c.caramelos()                ValueError: No quedan tantos caramelos!

6                                >>> c.poner\_caramelos(50)

>>> print(c)                     Traceback (most recent call last):

Caramelera con 6/20 caramelos    ...

                                    ValueError: No queda lugar para tantos caramelos

"""

class Caramelera:

    def \_\_init\_\_(self,capacidad:int) -> None:

        if not capacidad:

            raise ValueError("No se puede crear una Caramelera sin

cantidad inicial.")

        self.\_\_capacidad = capacidad

        self.\_\_cantidad = 0

    def poner\_caramelos(self,cantidad:int)->None:

        if not cantidad or cantidad < 0 :

            raise ValueError("No se puede agregar!.")

        if self.\_\_capacidad < self.\_\_cantidad+cantidad:

            raise  ValueError("No entran!.")

        self.\_\_cantidad += cantidad

    def sacar\_caramelos(self,cantidad:int)->None:

        if not cantidad or cantidad < 0 :

            raise ValueError("No se puede sacar!.")

        if cantidad >  self.\_\_cantidad:

            raise  ValueError("No hay!.")

        self.\_\_cantidad -= cantidad

    def \_\_len\_\_(self)->int:

        return self.\_\_cantidad

    def \_\_str\_\_(self) -> str:

        return f"Caramelera con {self.\_\_cantidad}/{self.\_\_capacidad} caramelos."

def main():

    c = Caramelera(25)

    print(c)

    c.poner\_caramelos(5)

    print(c)

    try:

        c.sacar\_caramelos(30)

    except ValueError as e: # creo una variable e con el texto de la

exception

        print(f"No se pudo realizar la operacion ==> {e}")

    else:

        print("se pudo sacar!!")

    if len(c) > 30:

       c.sacar\_caramelos(30)

       print("se pudo sacar!!")

    else:

        print("No hay")

    print(c)

main()

Caramelera con 0/25 caramelos.

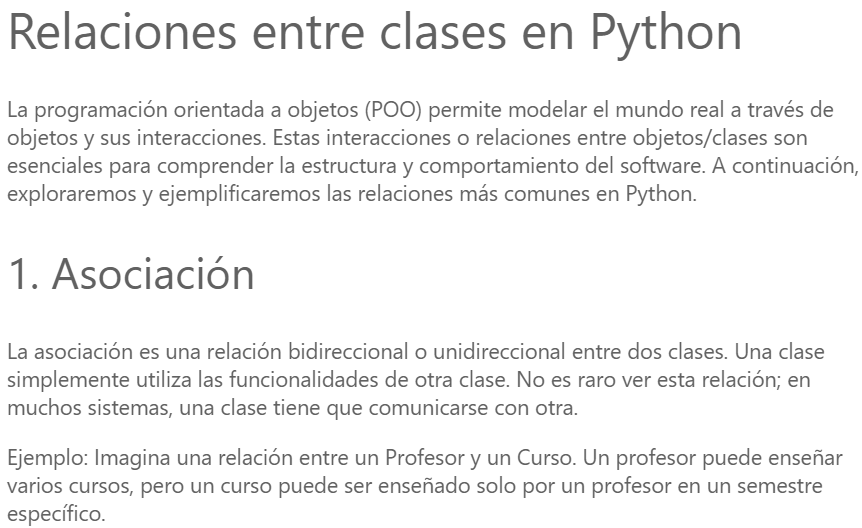
Caramelera con 5/25 caramelos.

No se pudo realizar la operacion ==> No hay!.

No hay

Caramelera con 5/25 caramelos.

clase 3:



class Curso:

    def \_\_init\_\_(self, nombre):

        self.nombre = nombre

class Profesor:

    def \_\_init\_\_(self, nombre):

        self.nombre = nombre

        self.cursos = []

    def asignar\_curso(self, curso):

        self.cursos.append(curso)

profesor = Profesor("Juan Perez")

curso1 = Curso("Matemáticas")

curso2 = Curso("Historia")

profesor.asignar\_curso(curso1)

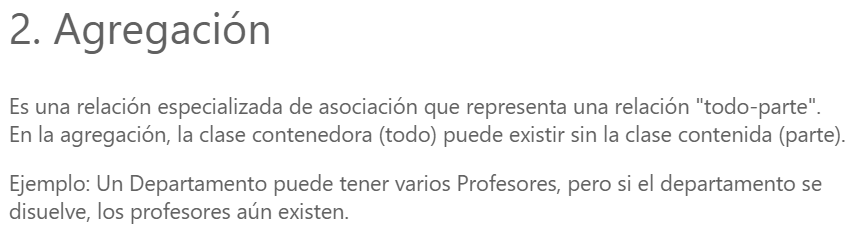
profesor.asignar\_curso(curso2)

profesor.asignar\_curso(Curso("Programación 1"))

profesor.asignar\_curso(Curso("Programación 2"))

print(f"{profesor.nombre} enseña: {', '.join( [curso.nombre for curso in profesor.cursos] )}.")

Juan Perez enseña: Matemáticas, Historia, Programación 1, Programación 2.



class Profesor:

    def \_\_init\_\_(self, nombre):

        self.nombre = nombre

class Departamento:

    def \_\_init\_\_(self, nombre):

        self.nombre = nombre

        self.profesores = []

    def agregar\_profesor(self, profesor):

        self.profesores.append(profesor)

historia = Departamento("Historia")

profesor1 = Profesor("Alicia Pepe")

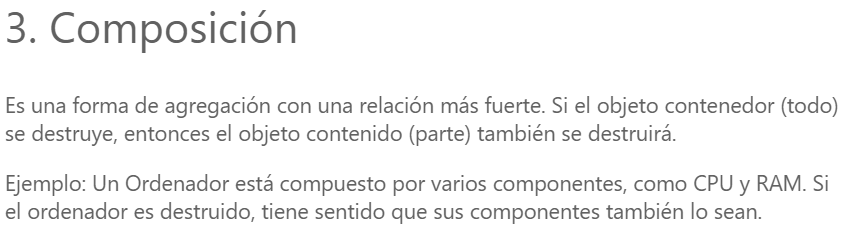
profesor2 = Profesor("Camila Sarasa")

historia.agregar\_profesor(profesor1)

historia.agregar\_profesor(profesor2)

print(f"{historia.nombre} Profesores: {', '.join([profesor.nombre for profesor in historia.profesores])}")

Historia Profesores: Alicia Pepe, Camila Sarasa



class CPU:

    pass

class RAM:

    pass

class Ordenador:

    def \_\_init\_\_(self):

        self.cpu = CPU()

        self.ram = RAM()

    def \_\_del\_\_(self):

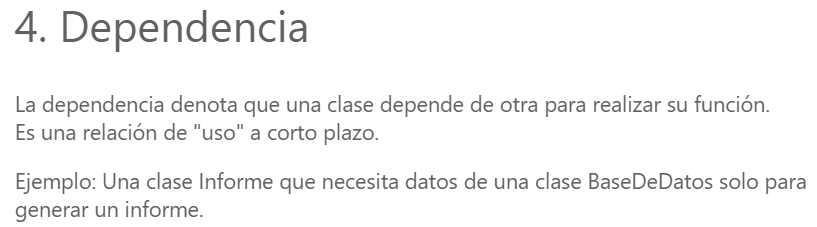
        print("El ordenador ha sido destruido junto con sus componentes.")

# Uso

pc = Ordenador()

del pc  # Esto destruirá el Ordenador y sus componentes

El ordenador ha sido destruido junto con sus componentes.



class BaseDeDatos:

    @staticmethod

    def obtener\_datos():

        return "datos del año 2023"

class Informe:

    @staticmethod

    def generar\_reporte():

        datos = BaseDeDatos.obtener\_datos()

        print(f"Generando informe con {datos}.")

# Uso

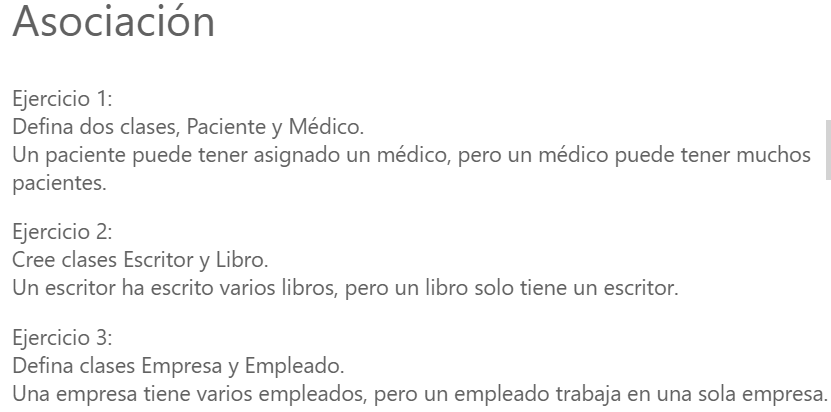
inf = Informe()

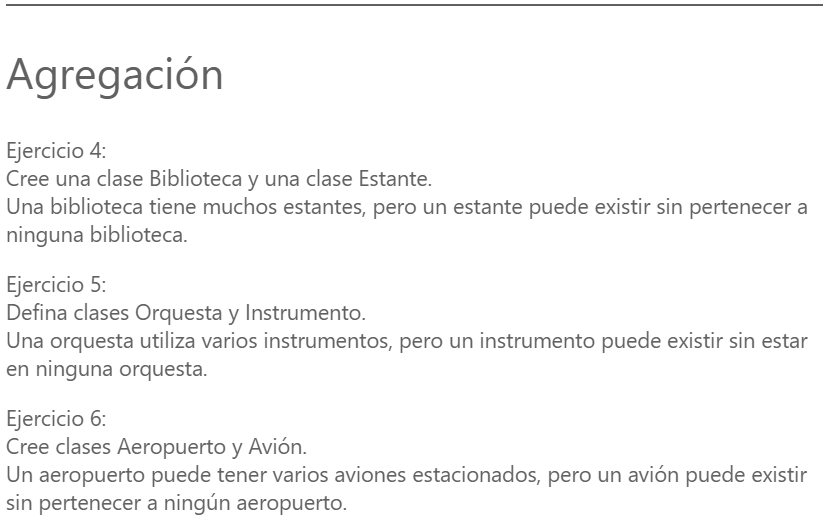
inf.generar\_reporte()

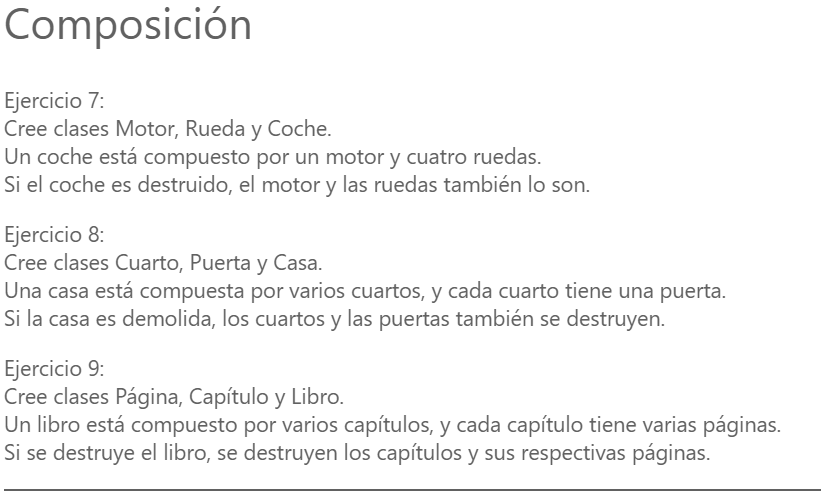
Informe.generar\_reporte()

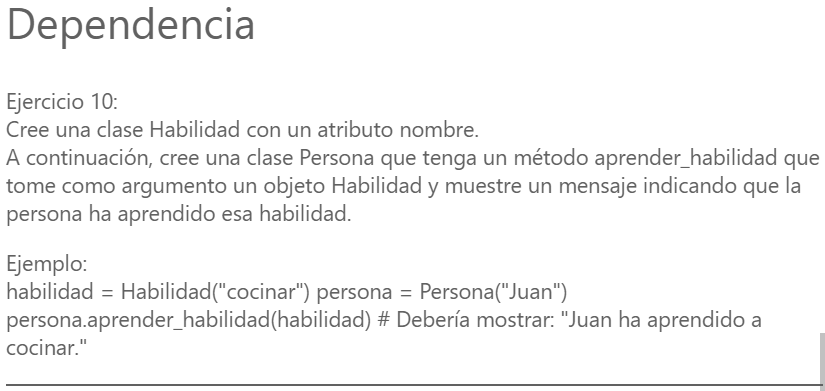
Generando informe con datos del año 2023.

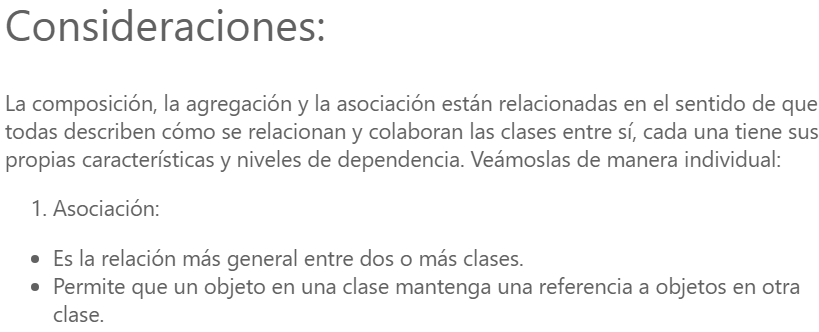
Generando informe con datos del año 2023.

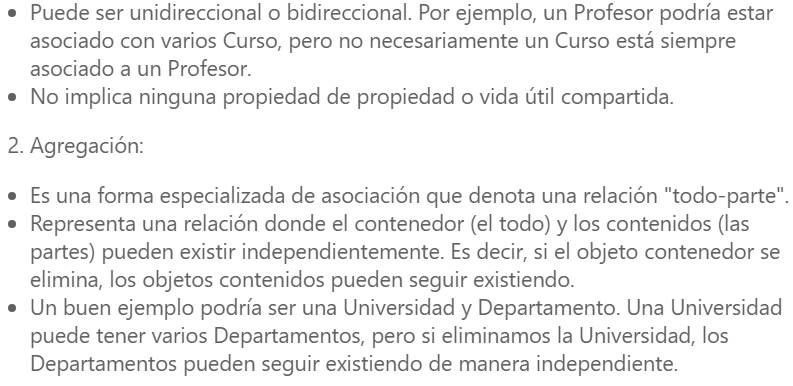


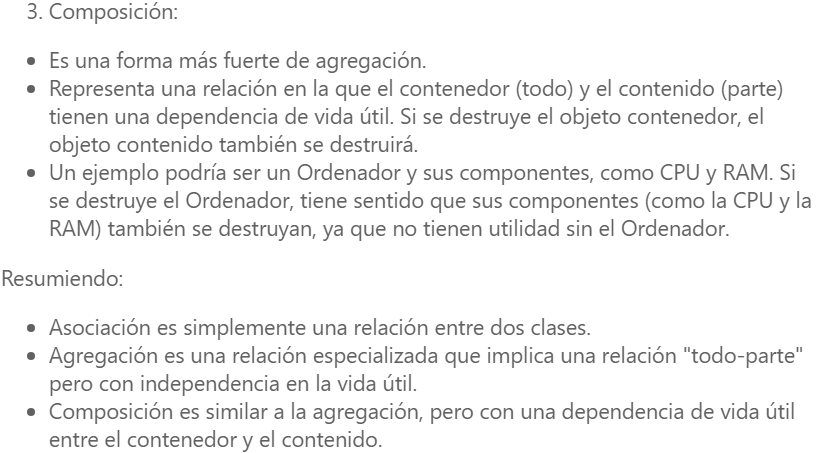


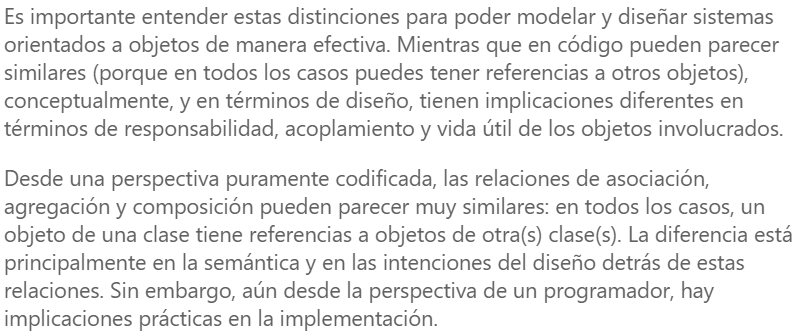


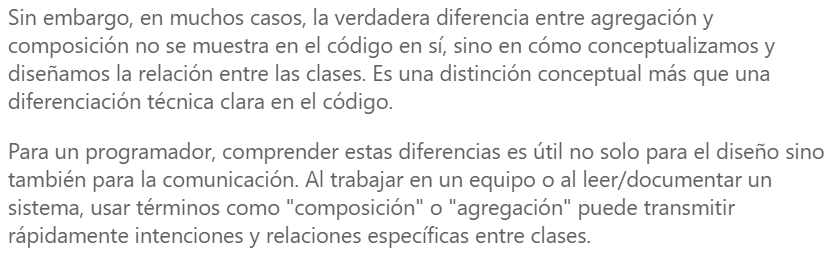












"""

Se pide implementar la clase Boleteria, que recibe en su constructor un evento y la cantidad de localidades para el mismo; de modo tal que cumpla el siguiente comportamiento:

>>> b = Boleteria("Rush",500)              >>> b.localidades\_agotadas()

>>> print(b)                               False

Evento: Rush - Localidades vendidas: 0 de 500     >>> b.vender\_localidades(100)

>>> b.vender\_localidades(400)              >>> b.localidades\_agotadas()

>>> b.vender\_localidades(200)              True

Traceback (most recent call last):         >>> print(b)

...                                     Evento: Rush - Localidades vendidas: 500 de 500

ValueError: No hay localidades suficientes

"""

class Boleteria:

    def \_\_init\_\_(self,nombre\_evento:str,cantidad\_entradas:int) -> None:

        self.\_\_nombre\_evento:str=nombre\_evento

        self.\_\_cantidad\_entradas:int = cantidad\_entradas

        self.\_\_vendidas:int = 0

    def getnombre\_evento(self)->str:

        return self.\_\_nombre\_evento

    def localidades\_agotadas(self)->bool:

        return self.\_\_cantidad\_entradas == self.\_\_vendidas

    def hay\_lugar(self,cantidad\_entradas)->bool:

        disponible = self.\_\_cantidad\_entradas - self.\_\_vendidas

        return cantidad\_entradas <= disponible

    def vender\_entradas(self,cantidad\_comprada:int)->None:

        if not self.hay\_lugar(cantidad\_comprada):

            raise ValueError('No hay localidades suficientes')

        self.\_\_vendidas+=cantidad\_comprada

    def \_\_str\_\_(self) -> str:

        return f"Evento: {self.\_\_nombre\_evento} - Localidades vendidas:

{self.\_\_vendidas} de {self.\_\_cantidad\_entradas}"

class EmpresaEntradas:

    def \_\_init\_\_(self,nombre:str) -> None:

        self.\_\_nombre:str = nombre

        self.\_\_boleterias:list[Boleteria] = []

    def agregar\_boleteria(self,boleteria:Boleteria)->None:

        if boleteria is None or not isinstance(boleteria,Boleteria):

            raise ValueError("No se puede agregar algo que no sea una

boleteria")

        self.\_\_boleterias.append(boleteria)

    def vender\_entradas(self,nombre\_evento:str,cantidad\_entradas:int)->None:

        evento\_encontrado = self.\_\_buscar\_evento(nombre\_evento)

        if evento\_encontrado is None:

            print("No existe el evento")

        elif evento\_encontrado.localidades\_agotadas():

            print("Localidades agotadas!!!")

        elif not evento\_encontrado.hay\_lugar(cantidad\_entradas):

            print("No hay lugar")

        else:

            evento\_encontrado.vender\_entradas(cantidad\_entradas)

            print("Localidades vendidas OK!!")

    def \_\_buscar\_evento(self,nombre\_evento:str)->Boleteria:

        for boleteria in self.\_\_boleterias:

            if boleteria.getnombre\_evento() == nombre\_evento:

                return boleteria

        return None

def main():

    ee = EmpresaEntradas("Ticketon")

    b1 = Boleteria("Palito Ortega",1000)

    b2 = Boleteria("Juan Ramon",1000)

    b3 = Boleteria("Juan y Juan",200)

    ee.agregar\_boleteria(b1)

    ee.agregar\_boleteria(b2)

    ee.agregar\_boleteria(b3)

    ee.vender\_entradas("Palito Ortega",20)

    ee.vender\_entradas("Palito Ortega",2000)

    ee.vender\_entradas("Juan Ramona",20)

Localidades vendidas OK!!

No hay lugar

No existe el evento