Licenciatura en Ciencia de Datos

Algoritmos II

Paradigma funcional: ¿Qué es?

-El paradigma funcional se basa en el concepto de **funciones matemáticas**

Concepto de función: es un tipo especial de **relación** en la que **cada elemento** del **dominio** está **asociado** con **exactamente un elemento de la imagen**

-Se centra en el uso de **expresiones**, la **evaluación de funciones** y en la **composición** de estas para resolver problemas.

-Es un **PARADIGMA DECLARATIVO** (se diseñan **soluciones** a través de la **aplicación de funciones**, definiendo **QUÉ se debe calcular en lugar de CÓMO** realizarlo a través de instrucciones)

Conceptos del paradigma funcional

-Inmutabilidad

-Funciones puras

-Transparencia referencial

-Evaluación perezosa

-Recursión

Concepto de estado y evaluación de funciones

-En el **POO**, el **ESTADO** de un **OBJETO** se determina por los **VALORES DE SUS ATRIBUTOS** en un momento dado, que **pueden modificarse.**

```
contador: int = 0
def incrementar_contador():
    global contador
    contador += 1
incrementar_contador()
print(contador)      # Salida: 1
```

-Se producen **EFECTOS SECUNDARIOS** (**cambios de estado**: pueden incluir la modificación de variables globales, de estructuras de datos mutables, la interacción con dispositivos de I/O , la manipulación de archivos, bases de datos, etc)

-En el **PARADIGMA FUNCIONAL** se reemplaza la noción de **ESTADO** por la **EVALUACIÓN DE FUNCIONES**, que generan **NUEVOS OBJETOS** a partir de otros **RECIBIDOS.**

- En Python todo **parámetro** de funciones **se pasa por VALOR**, lo que reduce la posibilidad de modificar variables.

Ciudadanos de primera clase

-Las **funciones como ciudadanos de primera clase** está relacionado con el concepto de **funciones de orden superior** de las matemáticas.

Una **función de ORDEN SUPERIOR** cumple **al menos una** de estas **condiciones**:

- **Recibe** una o más **funciones** como **argumento**
- **Devuelve** una **función**

-En un lenguaje que admite funciones como ciudadanos de primera clase, **LAS FUNCIONES SON TRATADAS COMO CUALQUIER OTRO TIPO DE DATO**: pueden ser **asignadas a variables**, **pasadas como argumentos** a otras funciones, **retornadas** como resultados de funciones y **almacenadas en estructuras de datos**

Ciudadanos de primera clase (cont)

```python
from typing import Callable

# Definimos una función que toma otra función como argumento y la aplica a un valor
def aplicar_funcion(func: Callable[[int], int], valor: int) -> int:
    return func(valor)

# Definimos una función que vamos a usar como argumento
def cuadrado(x: int) -> int:
    return x * x

# Podemos asignar la función 'cuadrado' a una variable
mi_funcion: Callable[[int], int] = cuadrado

# Ahora podemos pasar la función 'cuadrado' (o 'mi_funcion') como argumento a
'aplicar_funcion'
resultado1: int = aplicar_funcion(cuadrado, 5)
resultado2: int = aplicar_funcion(mi_funcion, 5)
print(resultado1)  # Salida: 25
print(resultado2)  # Salida: 25
```

Ejercicio

Ejercicio: Función de orden superior

1)Implementar una función llamada wrapper que reciba por parámetro a otra función f sin argumentos, la ejecute e imprima en pantalla el mensaje de ejecución: "Ejecutada f()".

2)Extender la función wrapper de forma que pueda aceptar cualquier función con argumentos variables y se puedan pasar también desde la función wrapper para que se invoquen en f. Por ejemplo, si f acepta 3 argumentos, éstos deberían también pasarse a wrapper para que se invoque f(arg1, arg2, arg3) dentro.

TIP: Ver el type hint Callable.

TIP 2: Ver pasaje de argumentos con *args y **kwargs.

Composición de funciones

-El **RESULTADO de una función se pasa como ARGUMENTO a OTRA**. **Python** lo facilita porque trata a las **funciones** como **ciudadanos de primera clase**

-La idea es evaluar la **APLICACIÓN SUCESIVA DE FUNCIONES** a partir de un **argumento inicial**: **h = g o f => h(x) = g(f(x))**

```
from typing import Callable

# Definimos una función que toma dos funciones como argumentos y retorna su composición
def componer(f: Callable[[int], int], g: Callable[[int], int]) -> Callable[[int], int]:
    def funcion_compuesta(x: int) -> int:
        return f(g(x))
    return funcion_compuesta
# Definimos algunas funciones simples para usar en la composición
def sumar_dos(x: int) -> int:
    return x + 2
def multiplicar_por_tres(x: int) -> int:
    return x * 3
# Componemos las funciones
# Primero se multiplica por 3, luego se suma 2
composicion = componer(sumar_dos, multiplicar_por_tres)
# Probamos la composición de funciones
resultado = composicion(4) # = sumar_dos(multiplicar_por_tres(4))
print(resultado)  # Salida esperada: 14, porque (4 * 3) + 2 = 14
```

Inmutabilidad

-**INCAPACIDAD** de un **objeto** para **CAMBIAR SU ESTADO después** de su **CREACIÓN**.

-Una vez que se **creó un objeto inmutable**, sus **ATRIBUTOS no** pueden ser **MODIFICADOS**.

Transitividad

- En Python, **los ATRIBUTOS de un objeto pueden ser REFERENCIAS a otros objetos**. Si un objeto contiene referencias a otros y son **MUTABLES**, el **objeto original puede MUTAR.**

-**MUTABILIDAD TRANSITIVA:** Si el **objeto B es mutable**, cualquier **cambio** en B a través de cualquier referencia **afectará** el **estado al objeto A**.

-**INMUTABILIDAD TRANSITIVA:** para que **el objeto A sea inmutable**, todos los **objetos** a los que A **hace referencia** (como B) **también** deben ser **inmutables**.

-**Tipos nativos de Python inmutables**: int, float, complex, bytes, str, tuple

- Tipos nativos de Python mutables: list, dict, set

Clases inmutables: ocultando atributos

El atributo de clase especial **__slots__** en Python permite **optimizar el uso de memoria** al **crear instancias** de una clase. Especifica un **conjunto FIJO de atributos** para las instancias

```
class Punto:
    __slots__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y
# Ahora, un objeto de la clase Punto solo puede tener los atributos x e y.
p = Punto(1, 2)
print(p.x, p.y)  # Salida: 1 2
# Intentar agregar un nuevo atributo dará un error
p.z = 3  # AttributeError: 'Punto' object has no attribute 'z' (si no hubiéramos definido slots funcionaría)
```

Clases inmutables: Properties

- Convertir los **atributos** en **PROPIEDADES DE SÓLO LECTURA**; **no definir los SETTERS**

```
class MiClaseInmutable:
    def __init__(self, valor_inicial):
        self._valor = valor_inicial
    @property
    def valor(self):
        return self._valor

objeto_inmutable = MiClaseInmutable(20)
objeto_inmutable.valor                      # 20
objeto_inmutable.valor = 10                 # AttributeError: property 'valor' of
'MiClaseInmutable' object has no setter
objeto_inmutable._valor = 10                # Modifica el valor
objeto_inmutable.valor                      # 10
```

Clases inmutables: Métodos __setattr__ y __delattr__

-Cuando realizamos la **ASIGNACIÓN de un ATRIBUTO,** internamente se invoca el método **__setattr__ ;** recibe como argumentos: el **objeto**, el **nombre del atributo** y el **valor** a asignarle**.**

Si lo SOBREESCRIBIMOS en nuestra CLASE INMUTABLE podríamos evitar cualquier tipo de asignación en los atributos DE LA CLASE.

-El método **__delattr__** es similar; recibe como **argumento** el **nombre del ATRIBUTO** para **ELIMINARLO de un objeto**. **SOBREESCRIBIRLO** sirve para **EVITAR que se eliminen** atributos**.**

-Hay que combinarlos con **__slots__**

Clases inmutables: Métodos __setattr__ y __delattr__

```
class Inmutable:
    __slots__ = ('x', 'y')
    def __init__(self, x, y):
        super().__setattr__('x', x)#Asignamos los valores iniciales utilizando super().__setattr__
        super().__setattr__('y', y) # para evitar la llamada a nuestro propio __setattr__.
    def __setattr__(self, nombre, valor):
        raise AttributeError(f"No se puede asignar el atributo '{nombre}' en una instancia de
Inmutable")
    def __delattr__(self, name):
        raise AttributeError(f"No se puede eliminar el atributo '{nombre}' en una instancia de
Inmutable")
    def __str__(self):
        return f"Inmutable(x={self.x}, y={self.y})"
# Ejemplo de uso
punto = Inmutable(1, 2)
print(punto)  # Salida: Inmutable(x=1, y=2)
punto.x = 3  # Salida: AttributeError: No se puede asignar el atributo 'x' en una instancia de
Inmutable
del punto.y  # Salida: AttributeError: No se puede eliminar el atributo 'y' en una instancia de
Inmutable
```

Clases inmutables: Named Tuples

-Las **tuplas** son un **tipo de dato inmutable** en Python

-En Python tenemos **namedtuple**. Permite **crear un OBJETO SUBCLASE DE TUPLE** => también **son inmutables**

-**Ventaja de namedtuple**:**inmutabilidad** + **proporciona NOMBRES a los ATRIBUTOS en lugar de ÍNDICES** =>código más **legible** y **fácil de mantener**

```
from collections import namedtuple

Punto = namedtuple('Punto', ['x', 'y']) # Definimos una namedtuple llamada 'Punto' con los atributos 'x' y 'y'
p = Punto(1, 2) # Creamos una instancia de 'Punto'
# Accedemos a los atributos por nombre
print(p.x)  # Salida: 1
print(p.y)  # Salida: 2
 p.x = 3  # Esto lanzará un error: AttributeError: can't set attribute
```

Clases inmutables: dataclasses

-Implementa automáticamente **métodos especiales.** Se **define** la estructura de la clase con **variables de clase con type hints,** y **@dataclass** genera los **atributos de instancia** implementando **__init__(), __repr__() y __eq__()**

```
from dataclasses import dataclass

@dataclass
class Persona:
    nombre: str
    apellido: str
    edad: int

    def es_adulta(self):
        return edad >= 18
```

=> genera:

-

Clases inmutables: dataclasses (cont)

```
class Persona:
    def __init__(self, nombre: str, apellido: str, edad: int):
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad

    def __repr__(self):
        return f"Persona(nombre={self.nombre}, apellido={self.apellido}, edad={self.edad})"

    def __eq__(self, otro):
        if isinstance(otro, Persona) and self.nombre == otro.nombre and
self.apellido == otro.apellido and self.edad == otro.edad)

    # Otras funcionalidades adicionales como __hash__ si la clase es inmutable,
    # comparaciones (si se usan parámetros de ordenación), etc.
```

Clases inmutables: dataclasses (cont)

- **@dataclass** tiene un **parámetro llamado frozen (booleano)** que permite **convertir a los atributos de instancia de solo lectura**.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Persona:
    nombre: str
    apellido: str
    edad: int
    def es_adulta(self):
        return edad >= 18

p = Persona("Julia", "Martinez", 22)
print(p)          # Persona(nombre='Julia', apellido='Martinez', edad=22)
p.edad = 20       # FrozenInstanceError: cannot assign to field 'edad'
```

Ejercicio

Implementar una versión de un conjunto de elementos de cualquier tipo que sea inmutable. Podemos apoyarnos en la `tuple` de Python. El conjunto se crea con una cantidad de elementos variables y luego ya no puede modificarse.

Funciones puras

Una **función pura** cumple con **DOS CONDICIONES**:

1) Dados los **mismos parámetros de entrada**, siempre d**evuelve el mismo valor = TRANSPARENCIA REFERENCIAL**
2) **No** debe producir **EFECTOS SECUNDARIOS:** incluir interacciones con el entorno del programa y **modificar el estado fuera del ámbito local de la función**

```
def suma(x: int, y: int) -> int:
    return x + y


nro: int = suma(10, 6) * 2
nro: int = 16 * 2           # Reemplazamos suma(10, 6) por su valor evaluado 16
```

La operación **suma** es una **función pura** porque cumple **ambas condiciones**

Tipos de efectos secundarios

-Modificación de Variables Globales

-Modificación de Argumentos

-Operaciones de Entrada/Salida (I/O): Interacciones con el mundo exterior, como lectura o escritura en archivos, envío de correos electrónicos, acceso a bases de datos, etc.

-**Impresiones en Consola o Registro de Eventos:** son operaciones de salida.

Interacciones de Red: Las operaciones de red, como solicitudes a una API

-Llamadas a Funciones con Efectos Secundarios: Si una función llama a otra función impura, **automáticamente se convierte en una función impura**.Si ambas producen efectos colaterales, **se pueden acumular y propagarse.**

-Generación de Números Aleatorios

Ejercicio

Proponer ejemplos de funciones impuras para cada tipo de efecto secundario mencionado y cómo se podrían convertir, si es posible, a versiones de funciones pura

Modificación de Variables Globales

```
contador = 0

def
incrementar_contador():
    global contador
    contador += 1
    return contador
```

```
def incrementar_contador_puro(contador):
    return contador + 1
```

Modificación de Argumentos

```
def agregar_elemento(lista):
    lista.append(4)
    return lista

mi_lista = [1, 2, 3]
agregar_elemento(mi_lista)
```

```
def agregar_elemento_puro(lista):
    nueva_lista = lista + [4]
    return nueva_lista

mi_lista = [1, 2, 3]
nueva_lista =
agregar_elemento_puro(mi_lista)
```

Creamos una nueva lista en lugar de modificar la existente.

Operaciones de Entrada/Salida (I/O)

```
def leer_archivo():
    with open('datos.txt', 'r') as file:
        contenido = file.read()
    return contenido
```

Las operaciones de I/O son **inherentemente impuras y no se pueden convertir directamente a funciones puras**. Sin embargo, **se puede minimizar su impacto** limitando su uso a una pequeña parte del código y mantener el resto del código puro.

Impresiones en Consola o Registro de Eventos

```python
def imprimir_mensaje():
    print("Hola, mundo!")
```

```python
def obtener_mensaje():
    return "Hola, mundo!"

# Uso
mensaje = obtener_mensaje()
print(mensaje)
```

La función pura **devuelve el mensaje** en lugar de imprimirlo directamente.

Interacciones de Red

```
import requests

def obtener_datos():
    response = requests.get('https://api.example.com/datos')
    return response.json()
```

Al igual que las operaciones de I/O, las interacciones de red son **impuras**. Se puede **encapsular** esta funcionalidad en una pequeña parte del código.

Llamadas a Funciones con Efectos Secundarios

```
def funcion_impura():
    print("Esto es impuro")

def otra_funcion():
    funcion_impura()
```

```
def funcion_pura():
    return "Esto es puro"

def otra_funcion_pura():
    mensaje = funcion_pura()
    return mensaje

# Uso
mensaje = otra_funcion_pura()
print(mensaje)
```

La segunda función puede **devolver un valor** que luego se puede **imprimir fuera de la función.**

Generación de Números Aleatorios

```python
import random

def generar_numero():
    return random.randint(1, 10)
```

```python
def generar_numero_puro(seed):
    random.seed(seed)
    return random.randint(1, 10)

# Uso
seed = 12345
numero = generar_numero_puro(seed)
```

Pasamos una semilla a la función para que el **resultado sea determinista.**

Estrategias de evaluación

-En el cálculo lambda **computamos** una expresión a través de su **REESCRITURA** aplicando **reglas de CONVERSIÓN Y REDUCCIÓN** hasta llegar a **EXPRESIONES IRREDUCIBLES= FORMA NORMAL O FORMA CANÓNICA**

-Dos **estrategias**:

1)**Orden aplicativo:** Se reducen primero las **expresiones reducibles** más **internas**. Ejemplo: **cuadrado(4 + 2) -> cuadrado(6) -> 6 * 6 -> 36**

2) **Orden normal:** Se reducen primero las expresiones reducibles más **externas.**

Ejemplo: **cuadrado(4 + 2) -> (4 + 2) * (4 + 2) -> 6 * 6 -> 36**

Evaluación estricta

-Similar al **ORDEN APLICATIVO**

-Se relaciona con el concepto de **evaluación impaciente o eager:** debemos **EVALUAR TODAS LAS EXPRESIONES INTERNAS** antes de avanzar con la externa, **aún si NO FUERAN NECESARIAS para calcular el valor**.

-**EN PYTHON CASI TODO SE EVALÚA ESTRICTAMENTE,** salvo algunas excepciones (como **and, or**)

Evaluación estricta (cont)

```
def imprimir_valor(valor):
    print(f"Valor: {valor}")
def doble(x):
    return x * x
# En evaluación estricta, 5 se evalúa antes de pasar a la función doble.
resultado = doble(5)
# Ahora llamamos a imprimir_valor con el resultado de la función doble.
# Nuevamente, en evaluación estricta, resultado ya está evaluado antes
# de pasar a imprimir_valor.
imprimir_valor(resultado)
# Si tuviéramos una expresión más compleja, también se evaluaría primero.
complejo = doble(3 + 2)  # 3 + 2 se evalúa primero (5), luego se pasa a doble.
imprimir_valor(complejo)
# Primero se evalúa doble(2), luego se evalúa el resultado como argumento de otra
llamada a doble.
resultado_anidado = doble(doble(2))  # doble(2) -> 4, luego doble(4) -> 8
imprimir_valor(resultado_anidado)
# Salidas esperadas:# Valor: 25 # Valor: 25 # Valor: 16
```

Evaluación no estricta

-Similar al **ORDEN NORMAL**, pero **no necesariamente** requiere **evaluar antes todas** las externas

-Podría **devolver un resultado** aún si **no se evaluaron todos sus argumentos** (porque **no se necesitó**).

-**ES LO QUE HACE EFICIENTE A LA PROGRAMACIÓN FUNCIONAL YA QUE PERMITE IGNORAR EXPRESIONES COSTOSAS O QUE PUDIERAN GENERAR ALGÚN ERROR**

Evaluación de cortocircuito

Se aplica a **expresiones booleanas.**

-Se implementa en Python y **permite evitar la evaluación de un segundo término dependiendo del valor del primero.**

<expresion_1> ***and*** *<expresion_2>* : Si **<expresion_1> es False**, <expresion_2> no se evalúa.

<expresion_1> ***or*** *<expresion_2>:* Si **<expresion_1> es True,** <expresion_2> no se evalúa.

```
def esDivisor(nro: int, divisor: int) -> bool:
    return (divisor > 0) and (nro % divisor == 0)

esDivisor(10, 0)     # False
```

Evaluación perezosa

-**Estrategia** que establece que **la EVALUACIÓN DE UNA EXPRESIÓN** puede **DILATARSE** hasta que sea **necesario su valor**.

-Es fundamental en la programación funcional: **PERMITE TRABAJAR CON ESTRUCTURAS "INFINITAS".**

-En Python: se hace a través de **GENERADORES** implementados mediante **funciones generadoras** que retornan un **iterador perezoso** mediante la palabra reservada **yield en vez de return**.

Evaluación perezosa (cont)

-Un **iterador** en Python es un **objeto** que **implementa** los métodos especiales **__iter()__ y __next()__** que nos **permite iterar** sobre una **colección de datos ITERABLE**.

-Al invocar la **función generadora**, el iterador puede ser almacenado en una **variable**. Cuando se invoca **next(<iterador>)** (implementado en el método __next()__) se ejecutan las **instrucciones** de la **función generadora** hasta el **yield**, se **suspende la ejecución de la función y se devuelve el valor actual** dado por la expresión del yield en ese instante. En **cada invocación del next() continúa la ejecución de la función** y se **vuelve a suspender** como el caso previo **o termina** si ya no quedan instrucciones.

-un **generador** sólo puede **consumirse una única vez**. Al finalizar, **no se puede volver atrás.**

Evaluación perezosa (cont)

```
from collections.abc import Iterator

def genera_saludo() -> Iterator[str]:
    yield "Hola"
    yield "Buenas"
    yield "Buen día"

iterador_saludos = genera_saludo()
print(next(iterador_saludos))    # Hola
print(next(iterador_saludos))    # Buenas
print(next(iterador_saludos))    # Buen día
print(next(iterador_saludos))    # Error StopIteration
```

Ejercicio

Implementar una función generadora que permita producir todos los números primos uno a uno.

Nota: Un número es primo si no es divisible por ningún número entre 2 y su raíz cuadrada.

Transformación de funciones: Currificación

-**Conversión** de una **función con n argumentos** en **n funciones con un único argumento**. Se **devuelve una función con aplicación parcial de un argumento**: **f(x, y, z) -> f(x)(y)(z)**

- **f(x) devuelve** una **función nueva** con el **argumento x** y **espera** como **argumento** a **y.** Esta **última** devuelve una **nueva función con y aplicado** y **devuelve otra función** que **espera** como argumento a **z** y **devuelve el valor final** de **f(x, y ,z)**.

Currificación

```
def suma(x, y):
    return x + y
def suma_curry(x):# Función currificada de suma
    def suma_x(y):  # Define una función interna que tomará el segundo argumento
        return x + y # Devuelve la suma de x (del ámbito exterior) e y (del
ámbito interior)
    return suma_x   # Devuelve la función interna
# Llama a la función suma con argumentos 1 y 3 y lo imprime
print(suma(1, 3))  # Esperado: 4
# Llama a la función currificada suma_curry con argumento 1.Esto devuelve la
función suma_x que se llama con el segundo argumento 3 y lo imprime
print(suma_curry(1)(3)) # Esperado: 4
```

functools.partial

-En Python tenemos la función **partial** que nos permite realizar la **vinculación de la aplicación parcial** a otra función.

```
from functools import partial
def producto(x: int, y: int) -> int:
      return x * y
producto_10 = partial(producto, 10) # Usa `partial` para crear una nueva función `producto_10` que recibe la función producto y setea x en 10
print(producto_10(2)) # Llama a la nueva función `producto_10` con el segundo argumento `y` igual a 2.Salida esperada: 20
# Esto equivale a producto(10,2), que devuelve `10 * 2` = 20
```

pymonad.tools.curry

-Podemos usar la **función decoradora @curry()** de la currificación. Le debemos indicar la **cantidad de argumentos con la cual se currifica**.

```
from pymonad.tools import curry
@curry(2)  # Aplica el decorador `curry` a la función `producto`, especificando que `producto` tomará 2 argumentos
def producto(x: int, y: int) -> int:
    return x * y
producto_10 = producto(10) # Llama a la función `producto` con el primer argumento `x` igual a 10. Debido al decorador `curry`, esto devuelve una nueva función que espera el segundo argumento `y`
print(producto_10(2)) # Llama a la función `producto_10` con el segundo argumento `y` igual a 2.  Esto completa la aplicación de `producto` con ambos argumentos, devolviendo `10 * 2
# Salida esperada: 20
```

Ejercicio

A lo largo de nuestro programa es posible que necesitemos almacenar información de interés en el log de ejecución. A efectos prácticos, nuestro destino de log será la consola, por lo que podemos utilizar simplemente un print() para registrar un mensaje de log.

Implementar una función log currificada que permita registrar un mensaje de log y el tipo, que puede ser error, alerta o información.

Composición con decoradores

-Una composición es la **aplicación de una función** sobre el **resultado** de otra **función evaluada**.

-Un decorador puede cumplirlo con esa definición ya que realiza:

mi_funcion = decorador(mi_funcion)

-**Útil** para definir cierto **comportamiento común** aplicable a varias funciones

Composición con decoradores (cont)

```
from collections.abc import Callable
from functools import wraps  # Para mantener los metadatos originales de la función decorada.
def trim(f: Callable[[str], str]) -> Callable[[str], str]:
    # Define un decorador que recibe una función 'f' que toma una cadena y devuelve una cadena
    @wraps(f)  #asegura rque el decorador preserve ciertos atributos de la función original, como su
nombre, docstring, y anotaciones, cuando ésta es envuelta por otro decorador.
    def wrapper(texto: str) -> str:
        return f(texto).strip()
    return wrapper
@trim  # Aplica el decorador 'trim' a la función 'transforma_texto'.
def transforma_texto(texto: str) -> str:
    return texto.replace('.', ' ')
transforma_texto('  esto es una prueba. ')  # 'esto es una prueba'
# Como 'transforma_texto' está decorada con 'trim', primero se reemplazan los puntos por espacios,
 y luego se eliminan los espacios en blanco al principio y al final del resultado.
```

Ejercicio

Se pide implementar una función decoradora acepta_no_valor que permita adaptar una función con un único parámetro de cualquier tipo no nulo de forma que devuelva la evaluación de esa función si el argumento recibido no es None. De lo contrario, debe devolver None.

TIP: Se puede usar el hint de tipo de retorno de la decoradora como: Callable[[T | None], R | None]. Ver Generics.

Funciones lambda

-Son una forma rápida de definir **funciones anónimas** que se usan para realizar operaciones simples, cuando se necesita una función para una **operación breve** y de **una sola línea**.

-Sintaxis: `lambda argumentos: expresión`

donde:

- **argumentos**: **parámetros** , separados por **comas**.
- **expresión**: se **evalúa** y se **devuelve como resultado**.

Ejemplo:

```
suma = lambda x, y: x + y
print(suma(2, 3))  # Salida: 5
```

Funciones lambda: Ejemplos

```
sin_parametros = lambda: 'Hola, mundo'
print(sin_parametros())  # Salida: Hola, mundo

doble = lambda x: x * 2
print(doble(5))  # Salida: 10

suma = lambda x, y: x + y
print(suma(3, 4))  # Salida: 7

potencia = lambda x, y=2: x ** y
print(potencia(3))  # Salida: 9 (3^2)
print(potencia(3, 3))  # Salida: 27 (3^3)
```

Iteraciones e iterables

-En la **programación imperativa** trabajamos con **bucles**

-En el **paradigma funcional** no tenemos estas estructuras => modelamos esta lógica a través de **funciones puras** y **recursión** (próximo tema).

-**Funciones** que podemos utilizar para trabajar con **objetos iterables** desde un **enfoque funcional** en Python:

- **Mapeos**: Construyen una **nueva colección** a partir de la **original** con la **misma cantidad de elementos** pero **aplicando cierta transformación**,
- **Filtrado**: Construyen una **nueva colección** a partir de la original pero con **una cantidad reducida de elementos**, **ignorando** aquellos que **no cumplen cierto criterio**.
- **Reducciones**: Producen **un valor** a partir de los **elementos de una colección**, por ejemplo sum(), max(), len().

map

-Recibe una **función** (orden superior) y **al menos un objeto iterable**, y **devuelve** un **iterador perezoso** (yield) que **entrega (a demanda) el resultado de aplicar esa función a cada elemento del iterable**.

-Si se pasaran **más de un iterable**, entonces la **función** debe **aceptar tantos argumentos como iterables.**

Sintaxis: `map(function, iterable, *iterables)`

-**Reemplaza** el comportamiento de un **bucle for-each**

map (cont)

```
xs: list[int] = [1, 2, 3, 4]
ys: list[int] = []
operacion = lambda x: x * x
for x in xs:
    ys.append(operacion(x))
cuadrados: map = map(operacion, xs) # <map at 0x1beb3187940> -> el objeto
retornado por map es un iterador perezoso: un objeto map
list(cuadrados)      # [1, 4, 9, 16]
```

-El objeto retornado es un **iterador perezoso** por lo cual aplica la función a toda la colección xs cuando sea necesario. **Recién cuando construimos una lista** a partir de un iterador **con el constructor list(), se itera sobre cada elemento**.

Ejercicio

A través del uso del map, dada una lista de cadenas generar una nueva lista que devuelva la cantidad que tiene de cierta letra (pasada como argumento) cada elemento.

Por ejemplo, si queremos contar la letra 'a' en ['casa', 'hogar', 'espacio', 'cuento'] deberíamos obtener [2, 1, 1, 0].

filter

-El filtrado de una **colección** consiste en aplicarle una **función booleana** para generar una **nueva colección** que **contiene sólo aquellos elementos** de la original donde **al aplicarles** la función retorna **Verdadero.** También es una función de **orden superior.**

-También devuelve un **iterador perezoso**

-Sintaxis: `filter(function, iterable)`

```
def es_par(n: int) -> bool:
    return n % 2 == 0
xs = [1, 2, 3, 4, 5, 6]
ys = []
for x in xs:
    if es_par(x):
        ys.append(x)
filter(es_par, xs)  # <filter at 0x1d2af1aed70>
list(filter(es_par, xs))    # [2, 4, 6]
```

reduce

-Representa el concepto funcional denominado **fold**, donde **se produce un valor** a partir de la **aplicación de una función acumuladora/combinadora/reductora** sobre una **estructura iterable**

La **idea** de esta operación se resume en estos **pasos**:

1. Obtener un **valor inicial** que será valor **acumulado/reducido**.
2. Si el **iterable no** tiene **elementos por iterar**, **devolver valor acumulado**, sino **avanzar al paso siguiente**.
3. **Aplicar una función reductora sobre el valor acumulado y el elemento actual** del iterable.
4. **Repetir 1** usando el **retorno de 2** como **nuevo valor acumulado**.

reduce (cont)

-Sintaxis: `reduce(funcion, secuencia)`

Ejemplo:

```
from functools import reduce
numeros = [1, 2, 3, 4, 5]
suma = reduce(lambda x, y: x + y, numeros) # Aplicar reduce para sumar todos los números en la lista
print(suma)  # Output: 15
```

reduce: Uso con valor inicial

-Es útil en casos donde la **secuencia podría estar vacía** o para **definir un valor inicial específico**.

-Sintaxis: `reduce(funcion, secuencia, valor inicial)`

Ejemplo:

```
from functools import reduce
numeros = [1, 2, 3, 4, 5]
suma_con_inicial = reduce(lambda x, y: x + y, numeros, 0)
print(suma_con_inicial)  # Output: 15
```

Ejercicio

Definir utilizando reduce una operación que dada una lista de cadenas devuelva un diccionario donde las claves sean cada elemento de la lista y los valores sean la cantidad de apariciones que tiene ese elemento en la lista.

Ejemplo: contar(['a', 'b', 'c', 'a', 'a', 'c', 'b', 'd', 'c', 'a', 'e']) -> {'a': 4, 'b': 2, 'c': 3, 'd': 1, 'e': 1}.