Tarea 1
OOP, notación O y decoradores

Profesores: Ignacio Bugueño, Pablo Martín **Auxiliar:** Esteban Muñoz
Ayudantes: Agustín González, Ignacio Romero, Diego Torreblanca

Fecha de entrega: 11 de Septiembre de 2024

1. Instrucciones

La siguiente tarea consta de una sección de preguntas teóricas y una sección de programación. Ambas secciones tributan a los contenidos vistos durante las clases anteriores, además de la introducción de un concepto existente en el lenguaje de programación Python. El objetivo es que adquieran nociones sobre conceptos básicos de programación, y tengan la capacidad de utilizarlos en su código. También se espera que conozcan un concepto que cuenta con su propia sintaxis en Python, y sepan cómo utilizarlo apropiadamente en su código.

La entrega será por U-Cursos a más tardar el **11 de Septiembre de 2024**, de modo que tienen 2 semanas para esta tarea. El formato de entrega es un link del repositorio de GitHub en donde almacenen su tarea. **Por favor mantengan ese repositorio en privado durante el plazo de la tarea para evitar la posibilidad de copias**. Dicho repositorio puede contener un único archivo `.ipynb` con la respuesta de ambas secciones. Sin embargo, si lo desea, puede realizar su tarea con una estructura de proyecto y usando archivos `.py` en lugar de `.ipynb`, en cuyo caso se le entregará una bonificación. Si elige esta última opción, su repositorio debe tener un archivo `README.md` con las respuestas a las preguntas teóricas, y con la explicación de cómo obtener los resultados.

La segunda parte de la sección práctica contiene varios pasos que sirven para introducir un nuevo concepto. Si usted ya sabe sobre decoradores, puede ir directamente al último paso y programar lo que se pide. La puntuación en esta parte tiene relación con el paso al que logra llegar, de modo que si no logra llegar al último paso recibirá un puntaje de todos modos, siempre y cuando entregue lo que se le pide.

2. Preguntas teóricas

- ¿Qué es un paradigma de programación?
- ¿En qué se basa la programación orientada a objetos?
- ¿Cuál es la diferencia entre recursividad e iteración, y cómo se relaciona esto con la notación big O?
- explicar la diferencia de rendimiento entre $O(1)$ y $O(n)$
- ¿Cómo se calcula el orden en un programa que funciona por etapas?
- ¿Cómo se puede determinar la complejidad temporal de un algoritmo recursivo?

3. Caminos en una PCB

Diseñar una PCB no es tarea fácil, pues existe una cantidad enorme de posibles soluciones para fabricar un mismo circuito. A modo de obtener una noción sobre este último punto, imagine este problema: usted quiere trazar un camino entre un punto A y un punto B en una PCB. A modo de facilitar el problema, se asumirán las siguientes simplificaciones:

- La PCB se divide en un conjunto de celdas, que forman una grilla de tamaño $N \cdot M$
- El punto A se encuentra en la esquina superior izquierda de esta grilla, mientras que el punto B se ubica en la esquina inferior derecha
- No puede moverse diagonalmente a través de la grilla, solo horizontal y verticalmente
- Solo puede acercarse al punto B. No puede ir a una celda que se encuentre más lejos o devolverse en su camino

Programe una clase que sea capaz de calcular la cantidad posible de caminos entre A y B, en al menos 2 formas distintas.

Existe una enorme posibilidad de que la eficiencia de sus soluciones sea diferente. A partir de ahora se le entregarán una serie de pasos que le permitirán extraer información sobre estas soluciones sin tener que modificar el código anterior, ni tener que crear nuevas funciones o métodos por cada solución.

1. Primero, puede crear un nuevo método por cada solución programada, que entregue el tiempo de ejecución de dicha solución para un input determinado.
2. ¿Qué pasaría si tuviese 6 soluciones distintas? ¿Tendría que programar 6 nuevas funciones o métodos para extraer la información deseada? Sin dudas esto sería un problema, pues repetiría muchas veces su código de forma innecesaria. Para evitar este escenario, puede aprovechar que las funciones en Python aceptan como argumento no sólo tipos básicos de datos, sino que también (entre otras cosas) aceptan funciones

```python
def funcion_que_recibe_otra_funcion(func, a, b):
    print("Llamando una función dentro de otra función")
    resultado = func(a, b)
    print("Haciendo algo después del resultado")
    return resultado
```

Código 1: Ejemplo de una función que recibe otra función

Como resultado, se obtiene lo mismo que entrega la función `func`, pero además se ha extendido su funcionalidad para hacer algo más. Sabiendo lo anterior, ahora puede programar un único método que recibe una función, y puede calcular el tiempo de ejecución de dicha función.

3. Lo anterior es útil en varios casos, pero tiene el siguiente inconveniente: suponga que tiene un proyecto de cientos de archivos, y en muchos de estos archivos tiene decenas de llamados a la función `func`. Usted se vería en la obligación de cambiar todas esas llamadas de `func(a, b)` por `funcion_que_recibe_otra_funcion(func, a, b)`, lo que empeora mucho la legibilidad de su código, se le podría olvidar cambiar alguno, y podría tener otros problemas que tienen relación con la calidad de su código.

   Para lidiar con estos problemas puede plantear la siguiente solución: usted tiene una función a la que le desea agregar funcionalidad. En lugar de crear una nueva función que entregue el mismo output, usted puede crear una función que *renueve* a `func` dotándola de nuevo funcionamiento. Para ello, en vez de retornar el mismo valor de `func`, puede retornar una nueva versión de `func` con el comportamiento extra que usted quiere entregarle.

```python
def funcionalidad_especifica(func):
    def nueva_func(a, b):
        print("Ejecutando código antes de func")
        resultado = func(a, b)
        print("Ejecutando código despues de func")
        return resultado

    return nueva_func
```

Código 2: Ejemplo de decorador en Python

A esto último se le llama *decorador*, y le permite hacer lo siguiente

```python
func = funcionalidad_especifica(func)
```

Por lo que ahora puede agregarle cualquier tipo de funcionalidad extra a `func` y a cualquier otra función, sin tener que modificar su contenido. Además, en Python existe una sintaxis específica para los decoradores

```python
@funcionalidad_especifica
def func(a, b):
resultado = a + b
return resultado
```

Código 3: Función decorada

4. Programe un decorador que permita almacenar de alguna forma el tiempo de ejecución de una función, y utilícelo para decorar un nuevo método en su clase. Este nuevo método debe ser capaz de cambiar la forma en que calcula la respuesta entre las distintas soluciones que programó en un principio.

Finalmente, escriba un programa capaz de: generar gráficos con el tiempo de ejecución de sus soluciones, o guardar estos gráficos en formato `svg` si decide entregar un proyecto de python en lugar de un notebook. Puede entregar uno o varios gráficos que muestren el tiempo de ejecución de todas sus soluciones para varios inputs distintos. El o los gráficos deben contener un título, leyenda, y tener cada eje rotulado.