1. **Contexto técnico**

Introducción a la Sobrescritura de Métodos y la Clase Comparator en Python

En el fascinante campo de las estructuras de datos, uno de los aspectos clave es la capacidad de comparar objetos. Python permite realizar esta tarea de manera eficiente a través de la sobrescritura de métodos especiales y la creación de clases personalizadas. Esto último es una técnica común en la programación Orientada a Objetos.

Sobrescritura de Métodos Especiales

En Python, los métodos especiales como __eq__, __gt__ y __lt__ son utilizados para definir comportamientos personalizados al comparar objetos. La sobrescritura de estos métodos permite que nuestras propias clases puedan ser comparadas utilizando los operadores estándar de igualdad y comparación.

- __eq__: Este método define el comportamiento del operador de igualdad ==. Permite especificar cuándo dos objetos se consideran iguales.
- __gt__: Este método define el comportamiento del operador "mayor que" >. Permite especificar cuándo un objeto se considera mayor que otro.
- __lt__: Este método define el comportamiento del operador "menor que" <. Permite especificar cuándo un objeto se considera menor que otro.

Ejemplo de Sobrescritura de Métodos

Consideremos una clase Persona donde deseamos comparar objetos basándonos en su edad:

```python
class Persona:
def __init__(self, nombre, edad):
self.nombre = nombre
self.edad = edad

def __eq__(self, other):
return self.edad == other.edad

def __gt__(self, other):
return self.edad > other.edad

def __lt__(self, other):
return self.edad < other.edad
```

Creación de una Clase Comparator

En la segunda parte de esta área de estudio, nos enfocaremos en la creación de una clase Comparator. Esta clase se encargará de comparar objetos y retornar un valor indicativo del resultado de la comparación: -1 si el objeto A es menor que B, 0 si son iguales, y 1 si B es menor que A.

Implementación de la Clase Comparator

A continuación, presentamos un ejemplo de cómo implementar la clase Comparator:

```python
class Comparator:
def compare(self, other):
  #primero TODAS las condiciones que podrían hacer que consideremos self < other
  if  self.propiedad1 <  other.propiedad1 or (self.propiedad2 == other.propiedad2 and ....< ) or  ...:
    return -1
  # ahora todas las condiciones que podrían hacer que consideremos other < self
  elif self.propiedad1 >  other.propiedad1 or (self.propiedad2 == other.propiedad2 and .... >) or  ...:
    return 1
  else:  # si no, son iguales.
    return 0
return 0
```

Con esta clase, podemos comparar objetos de manera flexible y eficiente, permitiendo integraciones sencillas en algoritmos de ordenación y otras aplicaciones que requieren comparación de objetos.

Ejemplo de Uso de la Clase Comparator

Consideremos el siguiente uso práctico de la clase Comparator con objetos de la clase Persona:

```python
persona1 = Persona("Alice", 30)
persona2 = Persona("Bob", 25)
comparator = Comparator()
resultado = comparator.compare(persona1, persona2)
print(resultado)  # Salida esperada: 1, ya que persona1 es mayor que persona2
```

En resumen, la sobrescritura de métodos especiales y la creación de una clase Comparator permite a los desarrolladores de Python manejar comparaciones de objetos de manera eficiente y personalizada, adaptando el comportamiento de los operadores estándar a las necesidades específicas de sus programas.

2. Requerimientos específicos

Primera Parte

Baje los archivos de NFL play by play en el repositorio https://github.com/ryurko/nflscrapR-data/tree/master/legacy_data/season_play_by_play.
Luego haga una clase llamada punt_play en python.
Dicha clase tendrá por atributos: id del juego (mismo del Excel), un string con las abreviaturas de los equipos en formato "abreviatura del equipo visita" @ "abreviatura del equipo casa", total de yardas ganadas (columna V), y cuarto (qtr) en que ocurrió la jugada.

Deberá colocar todos los archivos EN UNA CARPETA LLAMADA "\data\primeraprogramada" o "c:\data\primeraprogramda" según use MAC o PC. SI SU PROGRAMA NO BUSCA LOS ARCHIVOS AHÍ, INMEDIATAMENTE TENDRÁ COMO NOTA MAXIMA UN 75.

Haga una clase que se llame lector_data que recibe el nombre del archivo y retorna una lista de las jugadas que hayan sido PUNTS (contienen en la descripción "punts") y NO hubo fumble en ellas (o sea no contienen la palabra Fumble).

Ahora, en la clase punt_play, debe sobre escribir los métodos __eq__, __lt__ y __gt__ y los que correspondan (si los hay... investigue), para que puedan funcionar los operadores >,<,==, <=, >= PARA COMPARAR OBJETOS DE ESA CLASE. No puede acceder directamente a las propiedades. **Dicha comparación deberá hacerse por la cantidad de yardas obtenida en la jugada (si abre el .csv con Excel, verá que es la columna en la posición "v").**

Ahora, tome o busque código en python de ordenamiento por burbuja, ordenamiento por inserción, merge sort (SIN RECURSIVIDAD Y CON RECURSIVIDAD... o sea 2) y Quicksort (igual con y sin recursividad) y modifíquelo para aplicarlo a la lista retornada por la clase "lector_data".

Al modificar dicho código para comparar objetos NO podrá acceder a las propiedades de las jugadas.

Su programa debe leer los archivos, y arrojar:

1. Un archivo con las jugadas punt ordenadas de mayor a menor para cada algoritmo. (nombre del archivo "XXXXX-resultado.csv" donde XXXXX debe ser algo que indique que es la primera parte y el nombre del algoritmo. Obviamente el contenido será igual. El archivo de salida debe contener la versión abreviada para salida de nuestra clase para guardar jugadas en memoria, NO LA LINEA DE TEXTO ORIGINAL.
2. Debe imprimir en pantalla el tiempo de inicio, duración y tiempo final de la ejecución de cada algoritmo.
3. MODIFIQUE LOS ALGORITMOS PARA CONTAR CUANTAS COMPARACIONES E INTERCAMBIOS DE POSICIÓN HACE. Debe incluir esas estadísticas.

Estructuras de Datos
I Cuatrimestre 2025
Profesor Mario Agüero.

Segunda Parte

Ahora debe hacer OTRA implementación de los mismos algoritmos mencionados en la 1era parte, pero en lugar de comparar por la distancia ganada, debe ordenar las jugadas siguiendo los siguientes criterios:

1. Por fecha, de la mas anterior a la mas reciente (columna Date)
2. Por cuarto en que ocurrió la jugada. (columna qtr)
3. Distancia recorrida
4. En caso de que 2 jugadas tengan la misma distancia debe considerar la anterioridad en el tiempo (columna time)

LAS CONDICIONES DADAS IMPLICAN QUE DEBE MODIFICAR LA IMPLEMENTACIÓN de la clase hecha la primera parte.

La diferencia en la implementación de esta parte es que en lugar de usar los operadores ==, <, >, <=, >= ... para comparar usará una clase nueva llamada play_comparator. Dicha clase tendrá un método llamado compare() que recibirá 2 jugadas, llamelas A y B.

- Si A es “menor” que B, retorna -1
- Si B es menor que A, retorna 1
- Si son iguales (rarísimo), retorna 0.

Los archivos de salida deben cumplir las especificaciones en la parte 1 pero indicando que son los de la 2da parte.
Debe imprimir en pantalla los tiempos igual que en la parte 1.
Para esta OMITA el conteo de comparaciones e intercambios, solo indique los tiempos que se duró en sacar los archivos. No debe obtener estadísticas.

NOTA TECNICA:

- Recuerden que no pueden usar nada que requiera un pip install.. SOLO LA BIBLIOTECA ESTÁNDAR DE PYTHON. El uso de cualquier biblioteca como que requiera ello, pandas o pynum o lo que sea, inmediatamente TIENE CERO EN LA TAREA.
- DEBE ENTREGAR ARCHIVOS .PY
- DEBEN USAR VISUAL STUCIO CODE. CUALQUIER TAREA EN FORMATO “JUPYTER” u COLAB TIENE UN CERO INAPELABLE.
- DEBE USAR ORIETACIÓN A OBJETOS. NO HACERLO HACE QUE LA NOTA MAXIMA INMEDIAATAMENTE SEA 50.
- LA ENTREGA VIA GITHUB ES OPCIONAL.

FECHA DE ENTREGA: SABADO 15 DE FEBRERO DEL 2025 ANTES DE LA MEDIA NOCHE.