Tecnología de la Programación **Extraordinaria de 2024**
Duración: 3h:15m Evaluación Final.

Las soluciones tienen que estar **razonadas**. **Comenta** el código. **Indica los tipos de datos** de todas variables, atributos, argumentos de los métodos y el del valor de retorno. Se asume que **todos los atributos privados tienen propiedades** get/set (no los escribas). Para responder a una cuestión posterior **no debes reescribir el código** de cuestiones anteriores: solo debes indicar la clase de la cuestión anterior y añadir los métodos/atributos necesarios para la cuestión posterior.

ENTREGA EL ENUNCIADO JUNTO CON TUS SOLUCIONES

Apellidos y Nombre: .....

Ejercicio 1 (5 puntos = 2+2+1 puntos) + (1 Extra, si llega al menos a un 6)
Se considera la clase `Grafo` que codifica grafos NO-dirigidos con vértices enteros utilizando una lista de adyacencia: una lista que de vértices y cada uno tiene asociado una lista de sus vértices adyacentes.
• La lista de vértices está formados por nodos de la clase `NodoGrafo` que contiene los atributos `.dato`, un entero que almacena el valor del vértice, `.siguiente`, que almacena `None` o una referencia al siguiente nodo de tipo `NodoGrafo`, y `.adyacentes`, que referencia a una lista que almacena los vértices adyacentes al vértice `.dato`. Esta lista de adyacentes está formada por nodos de la clase `NodoLista` que contiene los atributos `.dato`, que almacena el valor del vértice adyacente, `.siguiente`, que almacena `None` o una referencia al siguiente nodo de tipo `NodoLista`. En consecuencia se tiene la clase `ListaVertices`, que usa una lista simplemente enlazada con nodos de tipo `NodoGrafo`, y la clase `Lista`, que usa una lista simplemente enlazada con nodos de tipo `NodoLista`. Las listas NO tienen cabecera. 1 *Escribe las clases indicadas en el párrafo anterior, utilizando la máxima abstracción que te permita Python (herencia), establece los atributos de cada clase indicando sus tipos de datos y construye los métodos de inicialización.*
• Los vértices, que son enteros, se guardan de forma creciente tanto en la lista de vértices como en cada una de las listas de adyacentes; es decir: `nodo_previo` $\prec$ `nodo_siguiente` sii `nodo_previo.dato` $<$ `nodo_siguiente.dato`.
2 Para una lista enlazada, *construye el método* `.append()` *que recibe como entrada un entero (un vértice), lo añade a la lista y retorna la referencia del nodo que lo almacena.* Ten en cuenta que el vértice ya puede existir.
• La clase `Grafo` dispone del método `.add_arista()` que recibe dos vértices `n` y `m`, y construye los arcos `(n, m)` y `(m, n)` del grafo, añadiendo, si fuera necesario los vértices faltantes. 3 *Construye dicho método.*
• Extra. 4 *Construye algún iterador que participe para mostrar la lista de adyacencia..*

Ejercicio 2 (5 puntos = 1+1+1+1+1) + (1 Extra, si llega al menos a un 6)
• Son **visitantes** (de una feria), los **menores** de edad y los **adultos**. De todo visitante se conoce su edad (no negativa) **y a partir de ella** se determinará si tiene derecho a un descuento o no. El menor de edad tiene menos de 18 años y el adulto tiene 18 años o más. Si un menor de edad tiene 8 años o menos (que llamaremos infantil) o si un adulto tiene 65 años o más (que llamaramos mayor) tendrán derecho a un descuento.
1 *Implementa las clases con los atributos y métodos mencionados comprobando el invariante de la representación en el constructor he indicando, usando la notación de Python, sus modificadores de visibilidad y si tendría una propiedad o método set y/o get en forma de comentario.* P.e. `self._salero: Set[Sal] # get, set` representa un atributo protegido y su valor es un conjunto cuyos elementos son granos de sal y tiene sentido hacer un get y set sobre dicho atributo.
`self.__notas: Dict[DNI, int] # get` es un diccionario privado y solo se puede consultar el valor entero asocido a un DNI.
• Una **cola de visitantes**, `ColaVisitantes`, es una clase hija del TDA Cola y cuyos elementos son objetos de tipo visitante. Dispone de un método que permite guardar los visitantes de la cola en un fichero de texto y que genera un **error propio** si los datos no pueden ser guardados. **2** *¿Cómo se construirá dicho método teniendo en cuenta que podrá generar un error propio?* • `ColaVisitantes` dispone del método factoría `def cola_aleatoria(n: int)->ColaVisitantes` que construye una cola de $2n$-visitantes aleatorios, n de cada clase, y que incluye el código de la izquierda. **3** *Complete el código dado para que se genere una cola de 10 visitantes de cada clase, usando las clases de 1, y añade el método factoría a la clase* `ColaVisitantes` *del apartado 2. ¿Cómo se invoca al método?* **4** *¿Cómo se invocará al método* `.guardar()` *para guardar la cola de visitantes que acabas de crear de tal forma que el programa no pare si se generara un error?*

```
for clase in [MenorEdad,
↪ Adulto]:
    ...
    edad=randint(0, 100)
    visitante = clase(edad)
    ...
```

• Tanto el recito ferial (solo existirá uno) como cada atracción (hay varias) tienen un aforo (son valores sin relación entre ellos). Tanto en el recinto como en las atracciones, los visitantes **entran** de uno en uno (e.d. hay un. método para entrar). El recinto y las atracciones conocen a todos los que entran en sus instalaciones. Cuando se alcanza un aforo no podrán entrar más visitantes (generando el correspondiente error propio). **5** *¿Cómo hemos de modelar el recinto y las atracciones? Sé explícito en su representación UML.* No se pide código.
• Además, en el caso de una atracción, solo pueden entrar los visitantes para los que está destinada la atracción, que puede ser para público infantil ($\leq$ 8 años), adulto ($\geq$ 18 años) o familiar (todas las edades). **6** *Dada una cola de visitantes ¿cómo es el método* `.entrar()` *en una atracción? ¿Cómo se invoca al método?* Genera y controla los errores propios.

Ejercicio 1

```python
from abc import ABC
from typing import Any, Union

class Nodo(ABC):
    def __init__(self, dato: Any):
        self.dato = dato

class NodoLista(Nodo):
    # Difiere de Nodo en que tienen un siguiente.
    def __init__(self, dato: Any):
        super().__init__(dato)
        self.siguiente: Union['Nodo', None] = None

class NodoGrafo(NodoLista):
    # Difiere de NodoLista en que tienen la lista de adyacentes de .dato
    def __init__(self, dato: Any):
        super().__init__(dato)
        self.adyacentes: Lista = Lista()
```

```python
class Lista:
    # Lista simplemente enlazada que ordena los nodos de forma creciente.
    # La lista no tiene nodo cabecera.

    # Esta lista trabaja con nodos de tipo NodoLista.
    # Permite añadir los nodos del grafo en la lista de adyacencia.

    NodoClass = NodoLista.

    def __init__(self):
        self.primero: Union['Lista.NodoClass', None] = None

    def append(self, dato: Any) -> Any: #'Lista.NodoClass':
        # Retorna el nodo de la lista donde se almacenó el vértice.

        if not self.primero:
            self.primero = self.NodoClass(dato)
            return self.primero

        if dato < self.primero.dato:
            nuevo_nodo = self.NodoClass(dato)
            nuevo_nodo.siguiente = self.primero
            self.primero = nuevo_nodo
            return nuevo_nodo

        actual = self.primero
        while actual.siguiente and actual.siguiente.dato <= dato:
            actual = actual.siguiente

        if actual.dato == dato:
            return actual

        nuevo_nodo = self.NodoClass(dato)
        nuevo_nodo.siguiente = actual.siguiente
        actual.siguiente = nuevo_nodo
        return nuevo_nodo

class ListaVertices(Lista):
    # Esta lista trabaja con nodos de tipo NodoGrafo (vertice + sus adyacentes).
    # Permite añadir los adyacentes de un nodo en la lista de adyacencias.
    NodoClass = NodoGrafo.
```

```python
class Grafo:
    def __init__(self):
        # Lista de adyacencias.
        self.lista_adyacencias = ListaVertices()

    def append(self, dato: Any) -> NodoGrafo:
        return self.lista_adyacencias.append(dato)

    def add_arista(self, n: int, m: int) -> None:
        # Añade n y m como vértices.
        nodo_n: NodoGrafo = self.append(n)
        nodo_m: NodoGrafo = self.append(m)

        # Añade m como adyacente de n, y n como adyacente de m.
        nodo_n.adyacentes.append(m)
        nodo_m.adyacentes.append(n)
```

Ejercicio 2

```
class ColaVisitantes(Cola[Visitante]):
    ...
    def guardar(self, nombre_fichero: str):
        try:
            with open(nombre_fichero, 'w') as f:
                cola: Cola[Visitante] = Cola() # Auxiliar para guardar los elementos de la cola
                while not self.is_empty():
                    visitante = self.pop()
                    cola.queue(visitante)
                    f.write(f"{visitante.edad}\n")
                while not cola.is_empty(): # Devolvemos los elementos a la cola original
                    self.queue(cola.pop())
        except FileNotFoundError as e:
            raise GuardarError(f"No se ha podido guardar la cola de visitantes en el fichero {nombre_fichero}") from e


cola_visitantes = ColaVisitantes.cola_aleatoria(10)
try:
    cola_visitantes.guardar("visitantes.txt")
except GuardarError as e:
    print(f"Error al guardar la cola de visitantes en el fichero {e}")
```

```
class Atraccion(IEntrar):
    ...
    def entrar(self, visitante: Visitante):
        if self.contador_personas < self.aforo_maximo:
            if self.tipo == TipoAtraccion.INFANTIL and not (isinstance(visitante, MenorEdad) and visitante.es_infantil()):
                raise AtraccionInfantilError("Este visitante no puede entrar en una atracción infantil")
            elif self.tipo == TipoAtraccion.ADULTO and not isinstance(visitante, Adulto):
                raise AtraccionAdultoError("Este visitante no puede entrar en una atracción de adultos")
            self.visitantes.add(visitante)
            self.contador_personas += 1
        else:
            raise AtraccionLlenaError("La atracción está llena")
```