EJERCICIOS PYTHON - CLASES

1. *TIMER.PY*

Tu tarea es implementar una clase capaz de contar segundos.

Tu clase se llamará Timer. Su constructor acepta tres argumentos que representan horas (un valor del rango [0..23]), minutos (del rango [0. .59]) y segundos (del rango [0..59]). Cero es el valor predeterminado para todos los parámetros anteriores. ***No es necesario realizar ninguna comprobación de validación.***

La clase en sí debería proporcionar las siguientes facilidades:

- Los objetos de la clase deben ser "imprimibles", es decir, deben poder convertirse implícitamente en cadenas de la siguiente forma: "hh:mm:ss", con ceros a la izquierda agregados cuando cualquiera de los valores es menor que 10.
- La clase tendrá dos métodos sin parámetros llamados next_second() y previous_second(), que incrementan el tiempo almacenado dentro de los objetos en +1/-1 segundos respectivamente.

Sugerencias:

- Las propiedades del objeto deben ser privadas.

- Escribe una función separada (¡no un método!) para formatear la cadena con el tiempo rellenando con ceros si es necesario.

```
class Timer:
   def __init__( ??? ):
      # Escribir código aquí

   def __str__(self):
      # Escribir código aquí para obtener hh:mm:ss

   def next_second(self):
      # Escribir código aquí

   def prev_second(self):
      # Escribir código aquí


timer = Timer(23, 59, 59)
print(timer)
timer.next_second()
print(timer)
timer.prev_second()
print(timer)
```

Ejecuta tu código y comprueba si el resultado es:

23:59:59
00:00:00
23:59:59

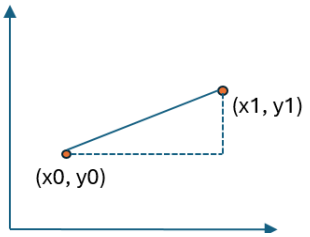
NOTA:

Cuando Python necesita presentar como una cadena alguna clase u objeto (en este caso cuando se ejecuta la sentencia print(timer), invoca a un método del objeto llamado __str__() y emplea la cadena que

devuelve. Por defecto, el método __str__() devuelve una cadena muy fea y poco informativa. Definiendo nuestro propio método __str__() haremos que la cadena a imprimir tenga otro aspecto más entendible.

2. POINT.PY

Tu tarea es escribir una clase en Python, que almacene las coordenadas cartesianas de un punto (**x** e **y**) como números flotantes y pueda evaluar la distancia entre dos puntos.



Puedes calcular la distancia entre los dos puntos empleando la función hypot() disponible en el módulo math que ya has empleado en ejercicios anteriores.

Indicaciones:

- Llama Point a la clase.
- Su constructor acepta dos argumentos (x e y), que por defecto serán cero.
- Todas las propiedades deben ser privadas.
- La clase contiene dos métodos sin parámetros llamados ***getx()*** y ***gety()***, que devuelven cada una de las dos coordenadas (las coordenadas se almacenan de forma privada, por lo que *no se puede acceder* a ellas directamente desde el objeto).
- La clase proporciona un método llamado ***distance_from_xy(x,y)***, que calcula y devuelve la distancia entre el punto almacenado dentro del objeto y el otro punto dado en un par de números flotantes.
- La clase proporciona un método llamado ***distance_from_point(point)***, que calcula la distancia (como el método anterior), pero la ubicación del otro punto se da como otro objeto de clase Point.

```
import math

class Point:
    def __init__(¿?):
        # Escribir código aquí

    def getx(¿?):
        # Escribir código aquí

    def gety(¿?):
        # Escribir código aquí

    def distance_from_xy(self, x, y):
        # Escribir código aquí

    def distance_from_point(self, point):
        # Escribir código aquí

point1 = Point(0, 0)
point2 = Point(1, 1)
print(point1.distance_from_point(point2))
print(point2.distance_from_xy(2, 0))
```

Ejecuta tu código y comprueba si el resultado es:

1.4142135623730951

1.4142135623730951

3. QUEUE.PY

Una *cola (queue)* es un modelo de datos caracterizado por el término **FIFO: primero en entrar, primero en salir**. Tu tarea es implementar la clase Queue con dos operaciones básicas:

- put(elemento): coloca elemento al final de la cola.
- get(): toma un elemento del principio de la cola y lo devuelve como resultado (la cola no puede estar vacía para realizarlo correctamente).

Indicaciones:

- Emplea una lista para el almacenamiento (como en el ejercicio de la pila).
- put() debe agregar elementos al principio de la lista, mientras que get() debe eliminar los elementos del final de la lista.
- Cuando get() intente operar en una lista vacía, se lanzará una excepción personalizada del tipo QueueError. Recuerda las indicaciones del ejercicio PaqueteNotas

```
raise ValueError("texto a mostrar")
```

genera una excepción del tipo ValueError que puede ser recogida en un try-except ValueError para su tratamiento.

```
except ValueError as e:
  print(f"Error: {e}")
```

- En este caso, necesitaremos crear la clase QueueError de la siguiente forma:

```
class QueueError(IndexError):
  pass
```

```
class QueueError (IndexError):
#  La clase QueueError es una subclase de IndexError
   pass


class Queue:
   def __init__(?):
      # Escribe el código aquí.

   def put(?):
      # Escribe el código aquí.

   def get(?):
      # Escribe el código aquí.


cola = Queue()
# Agrega y elimina elementos a la  cola
# Recoge el tratamiento de posibles excepciones
# que se pueden generar si la cola está vacía
```

4. SUPERQUEUE.PY

Tu tarea es crear una nueva clase SuperQueue que amplíe las capacidades de la clase Queue. Debe incorporar un método sin parámetros que devuelva True si la cola está vacía y False en caso contrario.

```
    # Código del ejercicio anterior.

class SuperQueue(Queue):
    # Escribe código nuevo aquí.

# Añade las sentencias para probar el correcto funcionamiento del método
```

5. TRIANGLE.PY

Vas a crear una clase Triangle que creará un triángulo definiendo sus tres vértices utilizando la clase Point del ejercicio Point.py

Indicaciones:

- El constructor de la clase triangle acepta tres argumentos - todos ellos son objetos de la clase Point.
- Los puntos se almacenan dentro del objeto triangle como una lista privada.
- La clase proporciona un método sin parámetros llamado perimeter(), que calcula el perímetro del triángulo descrito por los tres puntos (o sea, la suma de las longitudes de sus tres lados)

```
import math

class Point:
    # Código del ejercicio anterior.

class Triangle:
    def __init__(???):
        # Escribir código aquí

    def perimeter(??):
        # Escribir código aquí


triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
print(triangle.perimeter())
```

Ejecuta tu código y comprueba si el resultado es:

3.414213562373095

6. POLYGON.PY

Vas a transformar la clase Triangle en una clase Polygon, que sirva para trabajar con polígonos de cualquier número de lados, no solo triángulos.

Indicaciones:

- El constructor de la clase Polygon acepta cualquier cantidad de vértices, todos ellos objetos de la clase Point. Para ello podemos usar un parámetro de tipo *args, que permite recibir un número indeterminado de argumentos.

def function(nombre,*args):

Cuando un método declara un argumento de tipo ***args**, en la llamada al método cualquier argumento posicional adicional que se pase después de los parámetros definidos se captura en una tupla de nombre ***args**.

function("Elena", 4, 8,9)

El método function creará la tupla = (4,8,9)

Recuerda que la función list(args) crea una lista a partir de los elementos recibidos en la tupla.

- No realizaremos una validación de los puntos. Se asumirá que son puntos válidos y aparecen en el orden correcto para generar el polígono
- Los puntos se almacenan dentro del objeto polygon como una lista privada.
- La clase proporciona un método sin parámetros llamado perimeter(), que calcula el perímetro del polígono sumando las distancias entre los vértices consecutivos y cerrando el polígono al conectar el último vértice con el primero.

```
import math

class Point:
  # Código del ejercicio anterior.

class Polygon:
  def __init__(self, *vertices):
    # Escribir código aquí

  def perimeter(??):
    # Escribir código aquí


poligono = Polygon(Point(0, 0), Point(1, 0), Point(0, 1))
print(poligono.perimeter())
poligono = Polygon(Point(0, 0), Point(1, 0), Point(1,1), Point(0, 1))
print(poligono.perimeter())
poligono = Polygon(Point(0, 0), Point(2, 0), Point(2,1), Point(0, 1))
print(poligono.perimeter())
poligono = Polygon(Point(0, 0), Point(2, 0), Point(1,1), Point(0, 1))
print(poligono.perimeter())
```

Ejecuta tu código y comprueba si el resultado es:

3.414213562373095
4.0
6.0
5.414213562373095

7. ADDSTACK.PY

Vas a transformar la clase Stack para que disponga de un contador cuyo valor será la suma de todos los valores de la pila.

Tu tarea es diseñar la clase AddStack, subclase de la clase Stack para añadir esta funcionalidad. Incorporará además un método getSum() que devuelve el valor del contador.

Indicaciones:

- El constructor de la clase AddStack inicializará el contador con la suma de los elementos a cero.
- Los métodos push y pop de la clase AddStack deben actualizar el valor del contador.

```
class Stack:
   # Código del ejercicio anterior.


class AddStack(?):
   def __init__(?):
      # Escribir código aquí

   def get_sum(?):
      # Escribir código aquí

   def push(?):
      # Escribir código aquí

   def pop(?):
      # Escribir código aquí


pila = AddStack()
# Añade las sentencias para probar el correcto funcionamiento del método
```