Taller POO

1.Seleccion múltiple:

Dada la clase:

```
class A:
        x = 1
        _y = 2
        __z = 3
a = A()
```

¿Cuáles de los siguientes nombres existen como atributos accesibles directamente desde
a?
A) a.x
B) a._y
C) a.__z
D) a._A__z

RTA: A) a.x

2.Salida del programa:

```
class A:
        def __init__(self):
                self.__secret = 42

a = A()
print(hasattr(a, '__secret'), hasattr(a, '_A__secret'))
```

¿Qué imprime?

RTA= Imprime un false y un true, pues los atributos con protección se nombrarán automáticamente como “_nombreclase__nombreatributo”

3.Verdadero o falso:

a) El prefijo _ impide el acceso desde fuera de la clase.

RTA: Falso, lo que hace este es indicar que el atributo es protegido, mas no impide el acceso desde fuera de la clase.

b) El prefijo __ hace imposible acceder al atributo.

RTA: Falso, lo que hace ese prefijo es que solo deja acceder al atributo desde la misma clase en el que fue creado (a través del name mangling), mas no hace imposible acceder a este atributo.

c) El name mangling depende del nombre de la clase.

RTA: Verdadero, pues lo que hace este mecanismo es que cuando se utiliza el "__" en algún atributo le pone automáticamente el nombre de la clase antes del nombre del atributo.

4) Lectura de código

```
class Base:
        def __init__(self):
                self._token = "abc"

class Sub(Base):
        def reveal(self):
                return self._token

print(Sub().reveal())
```

¿Qué se imprime y por qué no hay error de acceso?

RTA: Lo que el programa imprime es "abc", y no hay error de acceso porque el dato es "protegido", por lo que no pone ningún problema para ser llamado desde otra clase.

5) Name mangling en herencia

```
class Base:
        def __init__(self):
                self.__v = 1

class Sub(Base):
        def __init__(self):
                super().__init__()
                self.__v = 2

        def show(self):
                return (self.__v, self._Base__v)

print(Sub().show())
```

¿Cuál es la salida?

RTA: La salida es “2,1”, esto debido a que el método “show” esta llamando al atributo “__v” de su propia clase(Clase Sub), y está llamando al atributo “_Base__v” de la clase “Base” mediante el name mangling

6) Identifica el error

```
class Caja:
        __slots__ = ('x',)

c = Caja()
c.x = 10
c.y = 20
```

¿Qué ocurre y por qué?

RTA: no hay ningún atributo de nombre “y”

7) Rellenar espacios

Completa para que b tenga un atributo “protegido por convención”.

Código original

```
class B:
     def __init__(self):
          self ______ = 99
```

Código completado

```
class B:
        def __init__(self):
                self._x= 99
        def imprimir
                return(self._x)

print(B().imprimir())
```

Escribe el nombre correcto del atributo.

8) Lectura de métodos “privados”

```
class M:
        def __init__(self):
                self._state = 0
        def _step(self):
                self._state += 1
                return self._state
        def __tick(self):
                return self._step()
m = M()
print(hasattr(m, '_step'), hasattr(m, '__tick'), hasattr(m, '_M__tick'))
```

¿Qué imprime y por qué?

RTA: El código imprime “true, false, true”, esto debido a que el método “step” solo esta protegido, por lo que no se le agrega el name mangling, al método “tick” si se le agrega el name mangling, esto por que ya es un atributo privado, el print llama al método dos veces, una de la manera incorrecta y la segunda de manera correcta, por eso se imprime “true, false, true”

9) Acceso a atributos privados

```
class S:
        def __init__(self):
                self.__data = [1, 2]
        def size(self):
                return len(self.__data)
s = S()

# Accede a __data (solo para comprobar), sin modificar el código de la
clase:

# Escribe una línea que obtenga la lista usando name mangling y la
imprima.
```

Escribe la línea solicitada.

RTA= print(s._S__data)

10) Comprensión de dir y mangling

```
class D:
        def __init__(self):
                self.__a = 1
                self._b = 2
                self.c = 3
d = D()
names = [n for n in dir(d) if 'a' in n]
print(names)
```

¿Cuál de estos nombres es más probable que aparezca en la lista: __a, _D__a o a?
Explica.

RTA: El atributo que es mas posible que aparezca en la lista es “_D__a”, pues “a” es un atributo privado, por lo que Python (y por lo tanto la función dir) reconoce ese atributo de manera automática como “_D__a”

Parte B. Encapsulación con @property y validación

11) Completar propiedad con validación

Completa para que saldo nunca sea negativo.

```python
class Cuenta:
    def __init__(self, saldo):
        self._saldo = 0
        self.saldo = saldo
    @property
    def saldo(self):
        return self._saldo

    @saldo.setter
    def saldo(self, value):
        if value < 0:
            raise ValueError("El saldo no puede ser negativo")

        self._saldo = value

c=Cuenta(100)
```

12) Propiedad de solo lectura
Convierte temperatura_f en un atributo de solo lectura que se calcula desde temperatura_c.

```
class Termometro:
    def __init__(self, temperatura_c):
        self._c = float(temperatura_c)

    @property
    def temperatura_c(self):
        return self._c

    @property
    def temperatura_f(self):
        return (self._c * 9/5) + 32



t = Termometro(input("Ingrese el valor en grados celcius"))

print(f"En celcius es {t.temperatura_c} y en farenheits es
{t.temperatura_f}")
```

13) Invariante con tipo
Haz que nombre sea siempre str. Si asignan algo que no sea str, lanza TypeError.

```
class Usuario:
    def __init__(self, nombre):
        if isinstance(nombre, str):
            self._nombre = nombre
        else:
            raise TypeError("El dato tiene que ser str")

    @property
    def nombre(self):
        return self._nombre


nombre = Usuario("s")
print(nombre.nombre)
```

14) Encapsulación de colección
Expón una vista de solo lectura de una lista interna.

```python
class Registro:
    def __init__(self):
        self.__items = ["obj.Escencial"]

    def add(self, x):
        self.__items.append(x)

    @property
    def items(self):

        return tuple(self.__items)
r=Registro()
r.add("obj.modificable1")
r.add("obj.modificable2")
print(r.items)
```

Crea una propiedad 'items' que retorne una tupla inmutable con el contenido

Parte C. Diseño y refactor

15) Refactor a encapsulación

Refactoriza para evitar acceso directo al atributo y validar que velocidad sea entre 0 y 200.

```python
class Motor:
    def __init__(self, velocidad):
        self.__velocidad = velocidad # refactor aquí
    @property
    def velocidad(self):
        if (0<= self.__velocidad <=200):
            return self.__velocidad
        else:
           raise ValueError("la velocidad tiene que estar entre 0 y 200")

m=Motor(100)
print(m.velocidad)
```

Escribe la versión con @property.

16) Elección de convención

Explica con tus palabras cuándo usarías _atributo frente a __atributo en una API pública de una librería.

RTA= Lo usaría en caso de que solo necesitara avisar que es mejor no utilizar el atrbibuto que tenga el “_”, pero que si se modifica ese atributo no conllevara inconveninetes.

17) Detección de fuga de encapsulación

¿Qué problema hay aquí?

class Buffer:

```
    def __init__(self, data):
        self._data = list(data)
    def get_data(self):
        return self._data
```

Propón una corrección.

RTA: El método “get_data” no tiene el @property por lo que no sirve de nada, simplemente le pondría el @property al método get data, además otro problema es que cualquiera puede modificar el atributo original, y en caso de que no se quiera eso, se podría mostrar una tupla, como se hizo en el ejercicio 14

18) Diseño con herencia y mangling

¿Dónde fallará esto y cómo lo arreglas?

```
class A:
    def __init__(self):
        self.__x = 1
class B(A):
    def get(self):
        return self.__x
```

RTA=Est código falla en que el método de la clase “B” no puede acceder asi a “self.__x”, pues este es privado y se le aplica el name mangling, para arreglar esto es tan fácil como poner “_A__x” en vez de “self.__x”

19) Composición y fachada

Completa para exponer solo un método seguro de un objeto interno.

```python
class _Repositorio:
    def __init__(self):
        self._datos = {}

    def guardar(self, k, v):
        self._datos[k] = v



    def _dump(self):
        return dict(self._datos)

class Servicio:
    def __init__(self):
        self.__repo = _Repositorio()

    def guardar(self, k, v):
        self.__repo.guardar(k,v)
```

```
s=Servicio()
s.guardar("animal", "raton")
```
Expón un método 'guardar' que delegue en el repositorio,
pero NO expongas _dump ni __repo.

20) Mini-kata

```
class ContadorSeguro:
    def __init__(self, n=0):
        self._n = n

    def inc(self):
        self._n += 1
        self.__log()

    @property
    def n(self):
        return self._n

    def __log(self):
        print("tick")

c = ContadorSeguro()
c.inc()
c.inc()
print("Valor final:", c.n)
```