Respuestas taller POO y modificadores de acceso en Python :

1. Las opciones A, B y D son accesibles
    - La opción A al ser un atributo público es accesible directamente.
    - La opción B es protegido pero sigue siendo accesible
    - La opción C NO ES accesible debido a que es una convención privada
    - La opción D aunque privada, por su manera de escribirse es accesible

2. El programa imprimirá: False True
    - Este programa imprime dos datos booleanos debido a los dos “hasattr”, para el primero imprime FALSE debido a el prefijo “__”, el segundo imprime TRUE por que aunque sea privado la manera en la que se escribe permite acceder al atributo.

3. FALSO, FALSO, VERDADERO
    - a.) Los atributos/métodos sujetos a este prefijo se pueden acceder incluso desde fuera de la clase.
    - b.) No lo hace imposible, solo activa en “Name mangling”
    - c.) Si, al acceder a un atributo/método privado mediante el Name mangling el nombre de la clase debe hacer parte de la línea de código en cuestión

4. Imprime “abc”, en otras palabras, el token.
    - Sub hereda de Base.
    - Cuando se instancia Sub(), se ejecuta el constructor de Base , creando el atributo protegido _token = "abc".
    - El método reveal() retorna self.token, osea “abc”.

5. 2 1

6. Hay un error de sintaxis debido a que el atributo “y” no existe dentro de __slots__

7. class B:
    def __init__(self):
    self._dato = 99

8. El programa imprimirá: True False True
    - hasattr(m, '_step') imprime TRUE porque efectivamente es accesible
    - hasattr(m, '__tick') imprime FALSE porque es privado y no es accesible de esta manera
    - hasattr(m, '_M__tick') imprime TRUE porque pese a ser privado, escrito de esta manera si es accesible

9. 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.

    print(s._S__data)

10. _D__a
    - Esto pasa porque “a” y “__a” están restringidos por su manera de escribirse y lo que hace dir es darnos el atributo “a” de una manera diferente

11. class Cuenta:
        def __init__(self, saldo):
            self._saldo = 0
            self.saldo = saldo
        @property
        def saldo(self):
        @saldo.setter
        def saldo(self, value):
            # Validar no-negativo
            if value >= 0:
                self._saldo = value
            else:
                print(“El saldo no puede ser negativo”)

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

    # Define aquí la propiedad temperatura_f: F = C * 9/5 + 32
        @property
        def temperatura_f(self):
            return self* 9/5 + 32
13. class Usuario:
        def __init__(self, nombre):
            self.nombre = nombre

         # Implementa property para nombre
        @property
        def nombre(str(self)):
            return self.nombre

        @nombre.setter
        def nombre(self, valor):
            if not isinstance(valor, str):
                raise TypeError("El nombre debe ser de tipo string")
            self.nombre = valor
14. class Registro:
        def __init__(self):
            self.__items = []
        def add(self, x):
            self.__items.append(x)
        # Crea una propiedad 'items' que retorne una tupla inmutable con el
    contenido
        @property
        def items(self):
            return tuple(self.__items)
15. class Motor:
        def __init__(self, velocidad):
            self._velocidad = 0
            self.velocidad = velocidad  # refactor aquí
        #versión con @property.
        @property
        def velocidad(self)
            return self._velocidad

        @velocidad.setter
        def velocidad(self, value)
            if 0 < velocidad < 200
                self._velocidad = value
        else:
            raise ValueError("La velocidad debe estar entre 0 y 200")
```

16. Elegiría “_atributo” cuando necesite un atributo protegido al cual pueda acceder desde fuera de la clase pero no pueda modificarlo fuera de ella, mientras tanto “__atributo” lo usaría para un atributo privado al que solo quiera acceder y modificar desde dentro de la clase, más que todo para evitar choques entre nombres y mantener el código organizado teniendo en cuenta que no hay una privacidad total.

    Siendo así usaría “_atributo” para advertir al usuario que no toque nada, reservando “__atributo” sólo si lo necesito ocultar de verdad (por ejemplo para seguridad).

17. #El problema en el código es que se puede modificar la información de self._data
    ```
    class Buffer:
            def __init__(self, data):
            self._data = list(data)
    def get_data(self):
            return tuple(self._data)  #se agrega tupla para que no se pueda modificar
    ```

18. #Cuando se llama a self.__x en la clase B python lo interpreta como “_B__x” y al no existir hay un error de atributos, se puede arreglar fácilmente de la siguiente manera

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

    class B(A):
            def get(self):
                    return self._A__x  #Cambio de mangling
    ```

19. 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)

20. class contador_seguro:
        def __init__(self, n):
            self._n = n

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

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

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

    c = contador_seguro(0)
    c.inc()
    c.inc()
    print(c.n)