RESPUESTAS:
1) Selección 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
RESPUESTA:
Las opciones A, B, D son accesibles directamente desde a.

1. Opción a: `x` es un atributo público. Python permite acceder directamente a todos los atributos públicos.
2. Opción b : _y es un atributo protegido, sugiere que no debería accederse directamente desde fuera de la clase, pero no impide el acceso Se puede acceder a `a._y` sin problemas, aunque no es una buena práctica.
3. Opción d: Esta es la forma correcta de acceder a un atributo privado en Python. El nombre se transforma internamente a `objeto_NombreDeClase__NombreDeAtributo=a._A__z` (*name mangling*).

2) Salida del programa class A:

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

a = A()
print(hasattr(a, '__secret'), hasattr(a, '_A__secret'))
¿Qué imprime

La primera llamada `hasattr(a, '__secret')` retorna `False` porque intenta verificar un atributo privado utilizando su nombre original, sin considerar el *name mangling* que utiliza Python. Cuando un atributo se define con doble guión bajo (`__secret`), Python lo renombra internamente como `_NombreDeClase__NombreDeAtributo`, en este caso `_A__secret`. Por lo tanto, el nombre `'__secret'` no existe directamente en el objeto a.

La segunda llamada `hasattr(a, '_A__secret')` retorna `True` porque utiliza el nombre interno correcto generado por el *name mangling*. Esto permite acceder al atributo privado de forma explícita, aunque no sea la forma recomendada para uso externo.

En resumen, `hasattr()` permite verificar si un atributo está definido en un objeto, pero en el caso de atributos privados, es necesario utilizar el nombre transformado por Python para que la verificación sea exitosa.

3) Verdadero/Falso (explica por qué)
a) El prefijo _ impide el acceso desde fuera de la clase.
b) El prefijo __ hace imposible acceder al atributo.
c) El name mangling depende del nombre de la clase.
RESPUESTAS:

a) FALSO El prefijo _ no impide el acceso desde fuera de la clase. En Python, este tipo de atributo se considera protegido, lo que sugiere que no debería usarse externamente. El atributo puede ser accedido directamente como `objeto._atributo`.

b) FALSO El prefijo __ convierte el atributo en privado, por medio *name mangling*. Esto significa que el nombre original del atributo se transforma en objeto.`_NombreDeClase__NombreDeAtributo`, aunque el acceso directo con `objeto.__atributo` falla, sí es

posible acceder al atributo si se utiliza el nombre transformado, como `objeto._Clase__atributo`.

c) VERDADERO

El name mangling depende del nombre de la clase siguiendo la estructura objeto.`_NombreDeClase__NombreDeAtributo`, ya que Python lo utiliza como parte de la transformación interna del atributo privado. Esto permite evitar conflictos entre atributos con el mismo nombre en diferentes clases.

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?

imprime abc

No hay error de restricción porque el atributo `_token` está definido como protegido, usando un solo guión bajo (_). En Python, esto no impide el acceso desde fuera de la clase ni desde clases hijas. Sugiere que el atributo está destinado para uso interno, pero sigue siendo técnicamente accesible.

La clase Sub hereda de `Base`, por lo tanto, puede acceder directamente a `_token` sin problema. Por eso el método

`reveal()` funciona correctamente y el programa imprime `"abc"` sin generar errores.

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())
```

RESPUESTA:

El programa imprime `(2, 1)` porque se están utilizando dos atributos privados con el mismo nombre `__v`, pero definidos en clases distintas. En Python, los atributos privados se transforman internamente mediante *name mangling*, lo que evita que se sobrescriben entre clases relacionadas por herencia.

Cuando se crea una instancia de Sub, primero se ejecuta el constructor de `Base` mediante `super().__init__()`, lo que define el atributo privado de `Base` con valor `1`. Luego el constructor de Sub define su propio atributo privado con el mismo nombre, pero con valor 2.

El método `show()` retorna ambos valores: el primero (`self.__v`) accede al atributo privado de Sub, y el segundo (`self._Base__v`) accede al atributo privado de `Base` usando el nombre transformado. Por eso la salida es `(2, 1)`.

6) Identifica el error

```
class Caja:
    __slots__ = ('x',)
c = Caja()
c.x = 10
c.y = 20
```

¿Qué ocurre y por qué?

La clase `Caja` define `__slots__ = ('x',)`, lo que significa que solo se permite crear el atributo `x` en las instancias. Cuando intentas asignar `c.y = 20`, Python no encuentra y en la lista de atributos permitidos por `__slots__`, y como no hay espacio para atributos dinámicos (no hay `__dict__`), lanza un error.

7) Rellenar espacios completos para que b tenga un atributo “protegido por convención”.

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

Escribe el nombre correcto del atributo.

RESPUESTA:

self._x=99

código:

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

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?

TRUE, FALSE, TRUE

El código define una clase `M` con un atributo protegido `_state`, un método protegido `_step()` y un método privado `__tick()`. Al crear una instancia `m` y usar `hasattr`, se verifica si existen ciertos métodos. `hasattr(m, '_step')` devuelve `True` porque los métodos protegidos son accesibles por convención. `hasattr(m, '__tick')` devuelve `False` porque los métodos privados se

ocultan mediante *name mangling*, lo que impide acceder a ellos directamente con su nombre original. En cambio, `hasattr(m, '_M__tick')` devuelve `True` porque ese es el nombre interno transformado del método privado, lo que permite acceder a él si se conoce la convención de renombramiento.

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.

```
print(s._S__data)
```

Esta línea accede al atributo privado `__data` usando *name mangling*, que transforma `__data` en `_S__data` para permitir su acceso desde fuera de la clase sin modificar su definición.

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.

Cuando se define un atributo como `self.__a`, Python lo considera privado y aplica *name mangling*, lo que significa que internamente lo renombra como `_D__a`, donde `D` es el nombre de la clase. Esto evita colisiones en herencia y protege el acceso directo.

La función `dir(d)` devuelve todos los nombres accesibles en el objeto d, incluyendo los renombrados por Python.

11) Completar propiedad con validación completa para que el saldo nunca sea negativo.

```
class Cuenta:
    def __init__(self, saldo):
        self._saldo = 0
        self.saldo = saldo
    @property
    def saldo(self):
        return self._saldo
    @saldo.setter
    def saldo(self, valor):
        if valor >= 0:
            self._saldo = valor
```

else:

raise ValueError("No puede ser negativo")

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_f(self):

return self._c * 9/5 + 32

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):

self.nombre = nombre

@property

def nombre(self):

return self._nombre

```
    @nombre.setter
    def nombre(self, valor):
        if isinstance(valor, str):
            self._nombre = valor
        else:
            raise TypeError("El nombre debe ser texto")
```

14) Encapsulación de colección

Expón una vista de solo lectura de una lista interna.

```
class Registro:
        def __init__(self):
                self.__items = []
        def add(self, x):
                self.__items.append(x)
        @property
        def items(self):
                return tuple(self.__items)
```

15) Refactor a encapsulación

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

200.

```
class Motor:
```

```
    def __init__(self, velocidad):
        self.velocidad = velocidad


    @property
    def velocidad(self):
        return self._velocidad


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

16) Elección de convención

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

En una API pública de una librería, eligen entre las dos opciones depende del nivel de seguridad que deseo según el programa, usar _atributo indica que es interno y no debería ser accedido por fuera de la clase, es posible utilizarlo cuando quiero proteger un dato pero permitir utilizarlo por ejemplo en una subclase.

Usar __atributo activa el name mangling que renombra al atributo para asegurarlo, esto impide el acceso directo y protege el dato, es recomendable usarlo cuando el atributo contiene información seguro o sensible.

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
```

Se define una lista protegida por convención (`_data`), pero el método `get_data()` la expone directamente permitiendo que cualquier código externo modifique el contenido.

```
class Buffer:
    def __init__(self, data):
        self._data = list(data)


    def get_data(self):
        return tuple(self._data)
```

18)

¿Dónde fallará esto y cómo lo arreglas?

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

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

El atributo `__x` está definido como privado en la clase `A` mediante doble guión bajo. Esto activa el name mangling, que renombra internamente el atributo como `_A__x` para protegerlo. Por lo tanto, cuando la subclase `B` intenta acceder a `self.__x` está mal.

forma correcta:

self._A__x

código corregido:

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


class B(A):
    def get(self):
        return self._A__x
```

19) Composición y fachada

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

```
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()  # Composición con atributo privado

    def guardar(self, clave, valor):
             self.__repo.guardar(clave, valor)  # Fachada: delega sin exponer
```

20) Mini-kata

Escribe una clase ContadorSeguro con:

• atributo “protegido” _n

• método inc() que suma 1

• propiedad n de solo lectura

• método “privado” __log() que imprima "tick" cuando se incrementa

Muestra un uso básico con dos incrementos y la lectura final.

```
class ContadorSeguro:
    def __init__(self):
        self._n = 0  # atributo protegido

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

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

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