Seminario de Lenguajes de Programación.
Seminarios de Lenguajes Dinámicos II

Yamilé Reynoso Díaz
Dayrene Fundora González
Camilo González Hurtado
Gelin Equinosa Rosique
Rafael Horrach Santiago

Equipo 8

22 de marzo de 2020

1. Problema 1:

Implemente en python los siguientes patrones:
1. Singleton: Clase de la que solo se puede crear una instancia, por lo que todos los intentos de instanciación deben resultar en un mismo objeto (el del primer intento). Tenga en cuenta en su implementación que las clases que hereden de esta clase deberán heredar también el comportamiento Singleton.

1.1. Patrón Singleton:

Es un patrón diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Garantiza que solo una instancia de la clase pueda existir. Crea un punto global de acceso para ese objeto, o sea, en el interior del objeto existe la instancia real del objeto y todo el que necesite usar el objeto solo puede hacerlo a través de esta instancia. Su intención consiste en garantizar que la clase tenga solamente una instancia. O sea, el patrón Singleton garantiza que pase lo siguiente:

```
>>> a = NewClass()
>>> b = NewClass()
>>> a is b
True
```

El resultado de lo anterior siempre es True, debido a que Singleton siempre devuelve la misma instancia, o sea, la instancia es creada en a y al tratar de crearla en b, lo que se devuelve es el propio a, por ende, a y b son el mismo objeto.

1.1.1. Usos del Patrón Singleton:

1. Cuando se necesita garantizar la existencia de una instancia de una clase (antivirus, conexión a base de datos, impresora o hardware, administrador de ventanas, etc.)

2. Si usamos otros patrones de los que requerimos solo una instancia como: fachada, prototipo, builder, fábrica.

3. Cuando se necesita controlar el acceso a la instancia

2. Solución:

Para la solución del problema planteado se consideraron 3 posibles vías de trabajo a seguir, a continuación, analizaremos cada una de ellas

2.1. Patrón Singleton Mediante Decoradores:

Un decorador es el nombre de un patrón de diseño. Los decoradores alteran de manera dinámica la funcionalidad de una función, método o clase sin tener que hacer subclases o cambiar el código fuente de la clase decorada. A partir de la implementación 2.4 del lenguaje de programación Python, se añadió una nueva característica al lenguaje que, si bien también se popularizó con el nombre decorador, no representa exactamente al mismo comportamiento definido por el patrón de diseño. De ahí, que pueda existir cierta confusión en la utilización de ambos conceptos y por tanto es conveniente aclarar.
El patrón de diseño Decorador, también conocido como wrapper, es uno de los patrones más utilizados. Tiene como propósito asignar nuevas responsabilidades a un objeto dinámicamente, independientemente de otras instancias de su misma clase. Es una alternativa flexible a la creación de subclases por herencia para añadir funcionalidades. Entre los efectos positivos de su uso se encuentran su mayor flexibilidad en comparación con la herencia estática, pues permite adherir y eliminar responsabilidades y comportamientos de forma sencilla en tiempo de ejecución, mientras que la herencia requiere una nueva clase para cada responsabilidad adicional.
A su vez, los decoradores de Python, no constituyen una implementación de este patrón. La diferencia radica en que los decoradores de Python permiten añadir funcionalidades a funciones y clases junto a su definición y no en la

invocación o creación de una instancia, lo que los convierte en una de las características más utilizadas del lenguaje. Los decoradores y su utilización nos ayudan a hacer nuestro código más limpio, a auto documentarlo y, a diferencia otros lenguajes, Python, no requiere que nos aprendamos otro lenguaje de programación (cómo pasa con las anotaciones de Java, por ejemplo). Cómo dice Michele Simionato nos aportan lo siguiente:

. Reducen el código común y repetitivo (el llamado código boilerplate).
. Favorecen la separación de responsabilidades del código
. Aumentan la legibilidad y la mantenibilidad
. Los decoradores son explícitos.

2.1.1. Código:

```
# Usando decoradores
def singleton(cls):
    instance = None

    def getinstnce(*args, **kwargs):
        nonlocal instance
        if not instance:
            instance = cls(*args, **kwargs)
        return instance
    return getinstnce

#Prueba
@singleton
class Prueba:
    def __init__(self):
        print("Yami")


#Clase que hereda de Prueba
class Heredero(Prueba):
    def __int__(self):
        print("Holaaaaaa")
```

En el código anterior se muestra primeramente la definición de una función singleton, la cual se usará como decorador y es la encargada, a su vez, de proporcionarnos el comportamiento del Patrón Singleton al objeto decorado. Luego se crean dos clases, Prueba y Heredero, la primera es decorada con la función singleton, por lo cual le proporciona este patrón, o sea de Prueba so-

lo tendremos una instancia, y la segunda es creada a través de Prueba, o sea, heredando de la clase Prueba. Veamos como utilizamos el código anterior:

```
a = Prueba()
b = Prueba()
print('a is b: ', a is b) #True
```

Como se puede apreciar en el código anterior, al crear las variables a y b como instancias de Prueba, vemos que, evidentemente, ambas instancias son la misma por lo cual se cumple el patrón Singleton mediante el uso de los decoradores, pero, si recuerdan, en el problema planteaban, además, que las clases que heredan de la clase con el comportamiento Singleton deben heredar este comportamiento. Veamos que pasa:

```
#1er problema
#Heredero NO se puede usar
c = Heredero()#Error
print('c is a ', c is a)
```

No podemos crear una instancia de la clase Heredero y esto se debe a que luego de ser decorada la clase Prueba, deja de ser un objeto y pasa a ser una función y NO se hereda de una función, luego en tiempo de ejecución Python nos lanza un error por tratar de heredar de una función, No podemos realizar la herencia usando los decoradores de funciones. Veamos un ejemplo más claro de que es lo que esta pasando:

```
#2do problema
#Singleton se puede evitar
d = type(a)()
print('d is a ', d is a) #False


print('type(a): ', type(a)) #Es una clase <class '__main__.MyClass'>
print('Prueba: ', Prueba) #Es una funcion <function singleton.<locals>.getinstance at 0x000001DDBBCD6400>
```

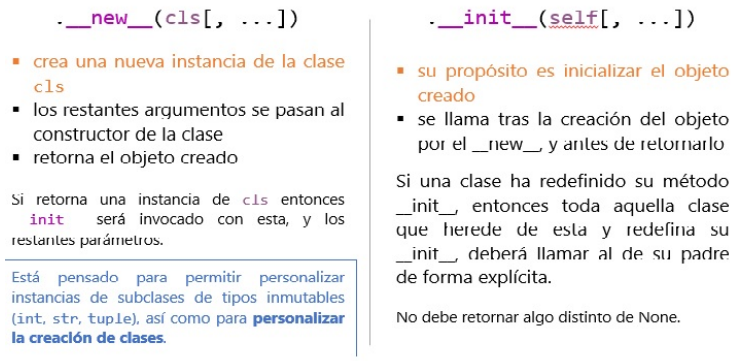
Por tanto, esta vía no es la correcta para hallar la solución del problema planteado

2.2. Patrón Singleton Mediante Clase Base:

Todo programador de Python que trabaje creando y/o utilizando clases debe estar al tanto de los “métodos mágicos“ o “métodos especiales“. Son aquellos que comienzan y terminan con doble guión bajo; como es el caso de __init__() que es uno de los más usados, dichos métodos no están pensados para ser invocados manualmente sino que son llamados por Python en situaciones

particulares (por ejemplo, cuando se realiza una comparación entre dos instancias vía los operadores >, ==, etc., o bien cuando se ejecuta alguna operación aritmética. Un ejemplo más amplio de estos métodos mágicos es:

. __init__(): Inicializa los atributos del objeto que se crea. Es el primer método que se ejecuta cuando se crea un objeto y es llamado automáticamente. Crea el objeto y luego lo inicializa
. __new__(): Construye el objeto
. __call__(): Permite crear clases donde las instancias se comporten como funciones
. __and__(): Ampersan
. __or__(): Or
. __it__(): Menor que
. __le__(): Menor igual que
. __eq__(): Igual Igual
. __iter__(): Itera sobre objetos



2.2.1. Código:

```
class Singleton:
    intance = None

    def __new__(cls, *args, **kwargs):
        if not isinstance(cls.intance, cls):
            cls.intance = super(Singleton, cls).__new__(cls, *args)
        return cls.intance



class Prueba():
    def __init__(self):
        print("Hello")

class Heredero(Prueba):
    def __init__(self):
        print("Hello again")
```

Similar a la primera alternativa de solución planteada, tenemos el objeto Singleton, que en este caso es una clase, a la cual se le modifico el método mágico new para crear una y solo una instancia del objeto que sea heredero de dicha clase Singleton. Luego tenemos la clase Prueba que hereda de Singleton, por lo cual adquiere en su comportamiento este patrón, así como la clase Heredero que se crea a partir de Prueba. Veamos cómo se puede usar el código anterior:

```
a = Prueba()
b = Prueba()
print('a is b: ', a is b) #True


#Heredero
c = Heredero()
print('a is c: ', a is c) #False
d = Heredero()
print('c is d: ', c is d) #True

#type
#Corrige el error de los decoradores con el type
e = type(a)()
print('e is a: ', e is a)#True
```

Como se puede apreciar esta alternativa de solución corrige los problemas presentados en la solución mediante decoradores. Se permite crear sin dificultad la clase Heredero y cumple con el Patrón Singleton. ¿Pero qué pasa si hacemos lo siguiente?:

```
class OtraPrueba:
    def __new__(cls, *args, **kwargs):
        return object.__new__(cls)

class Otra(OtraPrueba, Prueba):
    def __init__(self):
        print("another Hello ")
```

En este caso tenemos la clase OtraPrueba, que no implementa un comportamiento Singleton, o sea, crea instancias distintas cada vez que se llama, y tenemos la clase Otra que hereda de OtraPrueba y de Prueba. ¿Cómo debería comportase la creación de instancias de la clase Otra? ¿Cumplirán las instancias con el Patrón Singleton?

```
#Otra
f = Otra()
g = Otra()
print('f is g: ', f is g)#False
```

Como se puede apreciar las instancias de la clase Otra no se rigen bajo el patrón Singleton y esto se debe a la forma en que Python gestiona la Herencia Múltiple.
La Herencia Múltiple en Python se define como una lista de superclases ordenadas, con una especie de algoritmo por orden de profundidad y ejecuta el primer método que encuentra. Para la curiosidad, esta lista se encuentra guardada en la propiedad mro de la clase.
Por tanto, si volvemos al ejemplo, el primer método que encontrará la clase Otra para crear sus instancias será el método de la clase OtraPrueba, lo que anulará el que brinda el comportamiento Singleton que se encuentra en la clase Prueba. Esto queda comprobado aplicando Otra.__mro__:

```
print(Otra.__mro__)
#(<class '__main__.Otra'>, <class '__main__.OtraPrueba'>, <class '__main__.Prueba',
#<class '__main__.Singleton'>, <class '__main__.Object'>)
```

Por tanto, esta vía, aunque mejor que la anterior, tampoco será la correcta para hallar la solución del problema planteado

2.3. Patrón Singleton Mediante Metaclases:

> [Metaclasses] are deeper magic than 99% of users should ever worry about. **If you wonder whether you need them, you don't** (the people who actually need them know with certainty that they need them, and don't need an explanation about why).
>
> Tim Peters (c.l.p post 2002-12-22)

Las Metaclases son una clase cuyas instancias son clases en lugar de objetos. Es decir, si para construir un objeto usas una clase, para construir una clase usas una metaclase. El propósito fundamental de una metaclase es cambiar la clase automáticamente, cuando se crea, suele hacerlo para APIs, donde desea crear clases. Es de especial interés para los programadores a la hora de tratar de escribir APIs flexibles(Django) o herramientas de programación para que otros la usen.

Son muy útiles cuando no es posible determinar el tipo de un objeto hasta el momento de la ejecución del programa. Cuando sea necesario crear una clase a la medida de las circunstancias, funciona como una fábrica de clases. Cuando se desea componer o modificar el comportamiento o características de una clase en el momento de su creación por medio de herencia o por mecanismos de construcción dinámicos, o sea, es como una generalización del patrón decorador
Toda clase es instancia de una metaclase (type por defecto), del mismo modo que todo objeto es instancia de una clase (object por defecto).
Type es la madre de todas las metaclases, permite crear nuevos tipos, es una metaclase. Es la metaclase con la que están creados todos los tipos "build-in" de Python y todas las clases del "nuevo-estilo" (las que heredan de object). Para crear una clase con type() se usa la siguiente sintaxis:

```
type(name, bases, dct)
```

- **name**: es el nombre de la nueva clase
- **bases**: son las clases de las que hereda
- **dct**: es un diccionario con los métodos que implementa

Por ejemplo, puedes crear una nueva clase llamada Saludo que implemente un método hola() simplemente con:

```
Saludo = type('Saludo', (), {'hola': lambda self: 'hola metamundo
s = Saludo()
print s.hola()
```

De hecho, cualquier sentencia class de las que hemos usado hasta ahora, implica realmente una invocación similar, puesto que type es la metaclase implícita cuando defines clases "normales". Type puede ser heredada. Por lo cual cualquier subclase de type es una metaclase. __new__ e __init__ son dos de los métodos especiales que tienen todas las clases, para crear e inicializar instancias respectivamente, no reciben self, reciben como primer parámetro la metaclase y la clase creada por el método new respectivamente. En las metaclases, también contamos con estos métodos, pero aquí sirven para crear y modificar clases.
El atributo __metaclase__ puede ser añadido cuando se escribe una clase, si lo hace, Python utilizara la metaclase para crear la clase a la que fue añadido el atributo. Si Python no puede encontrar el __metaclass__ en la clase que hace referencia, lo buscará en su clase padre; si no lo puede encontrar en cualquier padre, buscará un __metaclass__ a nivel de modulo; si no encuentra ninguno en absoluto, utilizará type para crear el objeto; en una __metaclase__ ponemos "algo" que cree una clase y lo que crea una clase es type o subclases de type

2.3.1. Código:

```
class Singleton(type):
    def __init__(cls, *args, **kwargs):
        cls.instance = None
        super(Singleton, cls).__init__(*args, **kwargs)

    def __call__(cls, *args, **kwargs):
        if not cls.instance:
            cls.instance = super(Singleton, cls).__call__(*args, **kwargs)
        return cls.instance



class Prueba(metaclass=Singleton):
    def __init__(self):
        print("Hello")

class Heredero(Prueba):
    def __init__(self):
        print("Hello again")
```

En este caso se crea una metaclase, mediante Type, que va a contener la lógica del Patrón Singleton, o sea, toda clase que se cree mediante esta metaclase, se regirá mediante el Patrón Singleton. Pudiera llamar la atención el uso del método super() en la metaclase.
La función super nos permite invocar y conservar un método o atributo de una clase padre(primaria) desde una clase hija(secundaria) sin tener que nombrarla explícitamente. Hay dos casos de uso típico para super(). En una jerarquía

de clases con herencia única, super puede usarse para referirse a clases primarias sin nombrarlas explícitamente, lo que hace que el código sea más fácil de mantener. Este Uso es muy similar al uso de super en otros lenguajes de programación como java o C++.
El segundo caso de uso es admitir la herencia múltiple cooperativa en un entorno de ejecución dinámico. Este caso de uso es exclusivo de Python y no se encuentra en idiomas compilados estáticamente o en idiomas que solo admiten herencia simple. Esto hace posible implementar "Diagramas de Diamantes" donde múltiples clases base implementan el mismo método.
En el ejemplo anterior se usa super para crear el objeto mediante el método __init__ de su padre, en el caso anterior la metaclase Type, así como para las llamadas a la función mediante el __call__. Gracias a super, en el ejemplo vemos que, siempre que se crea una instancia de algún objeto derivado de Singleton, se entra al __call__ de la metaclase y dichos objetos son construidos a partir del __init__ de Singleton.
Continuando con el ejemplo se tiene la clase Prueba, creada mediante la metaclase Singleton y la clase Heredero que se crea mediante Prueba. Veamos que se obtiene de instanciar la clase Prueba:

```
a = Prueba()
b = Prueba()
print('a is b: ', a is b) #True

#Heredero
c = Heredero()
print('c is a: ', c is a ) #False

d = Heredero()
print('c is d: ', c is d) #True

#Type
e = type(a)()
print('e is a: ', e is a)#True
```

Funciona correctamente, pero hagamos un poquitico más transparente el proceso, usando __dict__. O sea, cada clase u objeto en Python tiene un atributo denotado por __dict__. Este diccionario contiene todos los atributos definidos del objeto. Entonces, veamos los atributos de nuestra clase Prueba:

```
Imprimiendo Pureba.__dic__ antes de crear instancias:
Prueba.__dict__:  {'__module__': '__main__', '__init__': <function Prueba.__init__ at 0x00000232BB0DE9D8>, '__dict__': <attribute '__dict__' of 'Prueba' objects>,
ef__' of 'Prueba' objects>, '__doc__': None, 'instance': None}

Imprimiendo Pureba.__dic__ despues de crear la instancia a:
Prueba.__dict__:  {'__module__': '__main__', '__init__': <function Prueba.__init__ at 0x00000232BB0DE9D8>, '__dict__': <attribute '__dict__' of 'Prueba' objects>,
ef__' of 'Prueba' objects>, '__doc__': None, 'instance': <__main__.Prueba object at 0x00000232BB0FF848>}

Imprimiendo Pureba.__dic__ despues de crear la instancia b:
Prueba.__dict__:  {'__module__': '__main__', '__init__': <function Prueba.__init__ at 0x00000232BB0DE9D8>, '__dict__': <attribute '__dict__' of 'Prueba' objects>,
ef__' of 'Prueba' objects>, '__doc__': None, 'instance': <__main__.Prueba object at 0x00000232BB0FF848>}
```

Ya tiene el instance de la metaclase

Las instancias de a y b son las mismas

Ahora sí, se observa que el objeto Prueba es creado con el atributo instance, que obtuvo de la metaclase Singleton, además se refleja que una vez creadas las instancias a y b de Prueba, ambas son las mismas. Pasemos a crear las clases OtraPrueba y Otra, lo cual presento error en la vía anterior de solución por la herencia múltiple

```
class OtraPrueba:
    def __new__(cls, *args, **kwargs):
        return object.__new__(cls)

class Otra(OtraPrueba, Prueba):
    def __init__(self):
        print("another Hello ")
```

Veamos que se obtiene de instanciar la clase Otra:

```
#Otra
f = Otra()
g = Otra()
print('f is g: ', f is g) #True

print(Otra.__mro__)
#(<class '__main__.Otra'>, <class '__main__.OtraPrueba'>,
#<class '__main__.Prueba'>, <class '__main__.Object'>)
```

Otro dato digno de mencionar es el funcionamiento del __call__. Si hacemos un print() dentro de la función __call__ para ver cuando es llamada, se obtiene lo siguiente:

```
a = Prueba() #Entro al llamado __call__()
b = Prueba() #Entro al llamado __call__()
print('a is b: ', a is b) #True

#Heredero
c = Heredero() #Entro al llamado __call__()
print('c is a: ', c is a ) #False

d = Heredero() #Entro al llamado __call__()
print('c is d: ', c is d) #True

#Type
e = type(a)() #Entro al llamado __call__()
print('e is a: ', e is a)#True

#Otra
f = Otra() #Entro al llamado __call__()
g = Otra() #Entro al llamado __call__()
print('f is g: ', f is g) #True
```

Como se puede observar, el __call__ es llamado en todas y cada una de las instancias de los objetos, puesto que se crean mediante la metaclase.Esto ocurre por el uso del método super que se mencionó anteriormente. Luego esta vía de solución, sí, nos brinda la respuesta esperada. Por lo que esa será la solución al primer problema del seminario.

3. Problema 2:

Implemente la clase ObjetoInmutable en Python. Un objeto inmutable, una vez creado, no podrá ser modificado. No ser modificado significa que después de su creación no se puede hacer o.t = valor ni del(o.t), siendo o una instancia de ObjetoInmutable. Realice su diseño de forma tal que cualquier clase que herede de ObjetoInmutable tenga el mismo comportamiento.

3.1. Objeto Inmutable:

Es un objeto cuyo estado no puede ser modificado una vez creado. En algunos casos un objeto puede ser considerado como inmutable, aunque algunos de sus atributos internos cambien, siempre y cuando el estado del objeto parezca no cambiar desde un punto de vista externo al mismo. Por ejemplo, un objeto que use memoización para cachear los resultados de cálculos costosos puede ser considerado un objeto inmutable. Los objetos inmutables suelen ser útiles dado que son seguros en entornos multihilo. Otro beneficio es que son más fáciles de entender y razonar sobre ellos, además de que ofrecen mayor seguridad que los objetos mutables.
En Python, algunos tipos primitivos (números, booleanos, cadenas, tuples, frozensets) son inmutables, pero las clases creadas por el usuario son en general mutables. Para simular inmutabilidad en una clase, se deben sobre escribir las características de modificación y borrado del atributo para activar excepciones. Para redefinir el comportamiento de un objeto, podemos usar otras de las funciones mágicas que nos brinda Python, en este caso, __getattr__, __setattr__, __delattr__.

```
__setattr__(self, nombre, valor):
```

Se le llama cuando se intenta una asignación a un atributo, en lugar de proceder al mecanismo normal (almacenar el valor en el diccionario de atributos de la instancia). 'nombre' es el nombre del atributo; y no debería hacer "self.nombre = valor"porque esto ocasionaría una llamada recursiva a sí mismo, sino insertar el valor en el diccionario de los atributos de la instancia (self.__dict__[nombre] = valor).

```
__getattr__(self, nombre):
```

Se le llama cuando una búsqueda de atributo no ha encontrado el atributo en los lugares habituales. Es decir, no es un atributo de instancia, ni se encuentra en el árbol de clases por sí mismo

```
__delattr__(self, nombre):
```

Igual que __setattr__ pero para el borrado de atributos en lugar de para su asignación. Este método sólo debería implementarse si la expresión "del objeto.nombre"tiene algún significado para el objeto.

3.2. Código:

```
class ObjetoInmutable:
    def __init__(self, **kwargs):
        for key, value in kwargs.items():
            super(ObjetoInmutable, self).__setattr__(key, value)

    def __setattr__(self, *args):
        raise AttributeError("Este es un objeto inmutable")

    __delattr__ = __setattr__
    __dict__ = None



class Prueba(ObjetoInmutable):
    pass

class Hereda(Prueba):
    pass
```

Como se puede apreciar en el código anterior se crea una clase ObjetoInmutable en la cual se modifican los atributos __setattr__ y __delattr__, de tal forma que, si el usuario quiere modificar el valor de una propiedad o eliminar dicha propiedad, se lance una excepción puesto que los objetos inmutables no se pueden modificar. Además, se crea la clase Prueba que hereda de ObjetoInmutable, por lo cual cumplirá con las normativas de los objetos inmutables, y la clase Hereda, que es un hijo de la clase Prueba. Veamos si se cumple el problema planteado, con esta solución:

```
a = Prueba(name="Yami")
print(a.name)

#a.name = "Dayrene"
#sprint(a.name) #Error, objeto inmutable

b = Hereda(lastname = "Reynoso")
print(b.lastname)

#b.lastname = "Diaz"
#print(b.lastname) #Error, objeto inmutable
```

Pues se puede apreciar que sí, los resultados son satisfactorios, tanto en la clase Prueba como en su sucesora, Hereda, si creamos un atributo con un valor e intentamos modificarlo después, se manda una excepción al usuario alertándolo de que esa operación no es válida, puesto que el objeto sobre el que está trabajando es inmutable.