



Solución Control 1

Pregunta 1

- a) ¿Cuál es el output del siguiente código si se ejecuta completamente? (2 ptos.)

```
X = [[12, 7, 3], [4, 5, 6], [7, 8, 9]]
Y = [[5, 8, 1], [6, 7, 3], [4, 5, 9]]
result = [[1, 0, 0], [0, -1, 0], [0, 0, 1]]

for i in range(len(X)):
    for j in range(len(X[0])):
        result[i][j] += X[i][j] + Y[i][j]
for r in result:
    print(r)
```

Solución:

```
[18, 15, 4]
[10, 11, 9]
[11, 13, 19]
```

- b) Indique como se declaran en Python atributos cuyos valores son compartidos por todos los objetos pertenecientes a una clase. (2 ptos.)

Solución: Deben declararse en una clase, fuera de cualquier función, sin utilizar `self`. Por ejemplo:

```
class Example:
    var = 2

instance1 = Example()
instance2 = Example()
print(instance1.var, instance2.var, Example.var) # 2 2 2
Example.var = 3
print(instance1.var, instance2.var, Example.var) # 3 3 3
```

- c) ¿Cómo podría implementarse en Python un mecanismo que permita avisar cuando el valor de un atributo es modificado, sin depender de la buena fe de quien está modificando el valor? (2 ptos.)

Solución: La manera más directa es utilizar *properties*, de manera de capturar modificaciones a una

variable. Por ejemplo:

```
class Example:
    var = 2

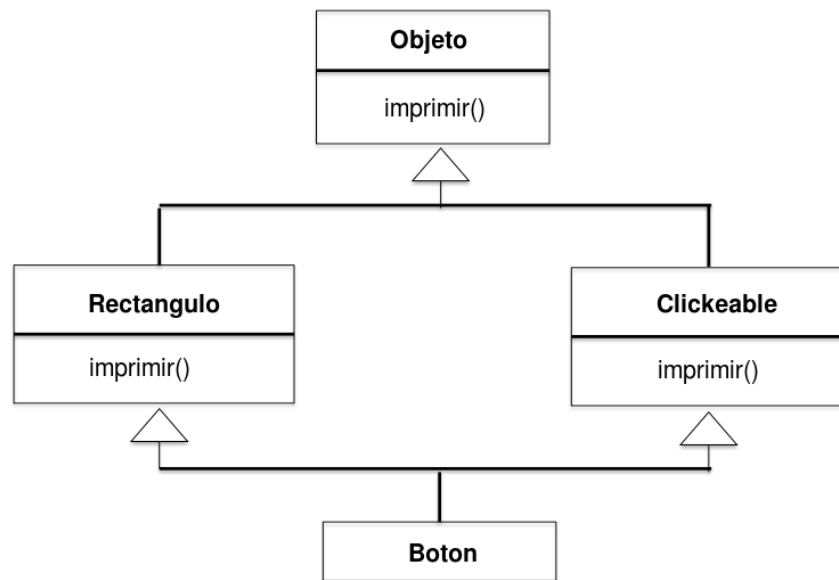
instance1 = Example()
instance2 = Example()
print(instance1.var, instance2.var, Example.var) # 2 2 2
Example.var = 3
print(instance1.var, instance2.var, Example.var) # 3 3 3
```

Pregunta 2

Considere el caso de una librería incompleta para diseñar interfaces de usuario gráfica en un lenguaje con soporte para orientación a objetos. En la versión actual de la librería, sólo existen tres clases, *Objeto*, *Rectángulo*, y *Clickeable*, donde las últimas dos heredan de la clase *Objeto*.

- a) Utilizando herencia múltiple, modele la clase *Botón* (entregue su respuesta tanto en Python como en un diagrama de clases). **(1 pto.)**
- b) Considere ahora que tanto las clases *Objeto*, *Rectángulo* y *Clickeable* implementan el método **imprimir**, que imprime en pantalla el nombre de la clase. En base a su respuesta del ítem anterior, ¿qué ocurre al llamar el método **imprimir**, en un objeto de tipo *Botón*? **(2 ptos.)**

Solución: El siguiente diagrama y código presentan la solución para las partes a) y b):



```
class Objeto:
    def imprimir(self):
        print("Objeto")

class Rectangulo(Objeto):
    def imprimir(self):
```

```

        print("Rectangulo")

class Clickable(Objeto):
    def imprimir(self):
        print("Clickable")

class Boton(Rectangulo, Clickable):
    pass

boton = Boton()
boton.imprimir() # Rectangulo

```

- c) Asuma ahora que no puede utilizar multiherencia. Modele la clase *Botón*, de manera que tenga el mismo comportamiento que tendría si se usara multiherencia. No es válido modificar la estructura de herencia de las clases *Objeto*, *Rectángulo*, y *Clickable*. (3 ptos.)

Solución:

```

class Objeto:
    def imprimir(self):
        print("Objeto")

class Rectangulo(Objeto):
    def imprimir(self):
        print("Rectangulo")

class Clickable(Objeto):
    def imprimir(self):
        print("Clickable")

class Boton(Rectangulo):
    def __init__(self):
        self.clickable = Clickable()

boton = Boton()
boton.imprimir() #Rectangulo

```

Pregunta 3

- a) Considere la clase *MezclaCompuestos* definida en Python, que al **ser llamada** se encarga de mezclar una serie de compuestos químicos, provistos como parámetros, y entrega como resultado el nuevo compuesto generado. Considere además que no existe un caso de mezcla de compuestos por defecto, o sea, esta depende de los elementos usados. Defina esta clase, considerando explícitamente el hecho que los compuestos utilizados pueden variar en cantidad y orden (el orden importa al momento de hacer la mezcla). (3 ptos.)

Solución:

```

class Compuesto:
    pass

class MezclaCompuestos:

```

```

def __call__(self, *argv):
    nuevo_compuesto = Compuesto()
    for arg in argv:
        ... # Se usa elemento arg en la mezcla de nuevo_compuesto.
    return nuevo_compuesto

mezclador = MezclaCompuestos()
mezclador('Hidrogeno', 'Hidrogeno', 'Oxigeno')

```

- b) Después de un largo tiempo, los desarrolladores de la clase *MezclaCompuestos* notaron que la complejidad de esta era demasiado alta, principalmente debido a la gran variedad de casos que debía manejar para realizar las mezclas. Para disminuir la complejidad, decidieron dividirla en una serie de especializaciones, donde cada una implementa una posible mezcla. Defina en Python esta nueva estructura de clases (utilice los nombres que quiera para las especializaciones). **(3 ptos.)**

Solución:

```

from abc import ABCMeta, abstractmethod

class Compuesto:
    def __init__(self, nombre = ""):
        self.nombre = nombre

class MezclaCompuestos(metaclass=ABCMeta):
    @abstractmethod
    def __call__(self, *argv):
        pass

class Especializacion1(MezclaCompuestos):
    def __call__(self, *argv):
        if (argv[0] == "Hidrogeno" and argv[1] == "Hidrogeno"
            and argv[2] == "Oxigeno"):
            ... # solo si los elementos son los correspondientes a la
            ... # especializacion se hace algo.
            return Compuesto("H2O")
        else:
            return Compuesto()
    ...

class EspecializacionN(MezclaCompuestos):
    def __call__(self, *argv):
        if (argv[0] == "Sodio" and argv[1] == "Cloro"):
            ... # solo si los elementos son los correspondientes a la
            ... # especializacion se hace algo.
            return Compuesto("NaCl")
        else:
            return Compuesto()

mezclador1 = Especializacion1()
mezcladorN = EspecializacionN()
print(mezclador1('Hidrogeno', 'Hidrogeno', 'Oxigeno').nombre)
print(mezcladorN('Hidrogeno', 'Hidrogeno', 'Oxigeno').nombre)
print(mezclador1('Sodio', 'Cloro').nombre)
print(mezcladorN('Sodio', 'Cloro').nombre)

```
