UNIDAD 4

PROGRAMACIÓN ORIENTADA A OBJETOS

TEMA 1:

Desarrollando Orientado a Objetos

FR0018/ v1.0 / 18-11-2019

UNEMI ONLINE

ÍNDICE

1. Unidad 3: Desarrollando Orientado a objetos 3

1.1 Objetivo 3

1.2 Introducción 3

2. Información de los subtemas 4

2.1 Abstracción y Encapsulamiento 4

2.2 Herencia y Polimorfismo 10

3. Recursos complementarios 15

4. Bibliografía 16

1. Unidad 4:

» **Objetivo:**

Elaborar programas utilizando técnicas de programación orientada a objetos aplicando un lenguaje de programación en un entorno de desarrollo.

» **Introducción:**

La programación orientada a objetos tiene como pilar la reutilización de código.

Los dos subtemas que se desarrollan ayudaran al estudiante a tener un enfoque de los pilares de la Programación Orientada a Objetos. Se analizarán la Abstracción y Encapsulamiento.

En el segundo subtema esta la Herencia y Polimorfismo la cual ayuda a los desarrolladores de software con la reutilización de código.

© Universidad Estatal de Milagro – UNEMI

2. Información de los subtemas

2.1 Abstracción y Encapsulamiento

Abstracción

Según Joyanes Aguilar (2003) la abstracción "es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado" (p. 57).

La encapsulación se encuentra dentro de los conceptos de abstracción. Es decir, es una forma de abstracción.

Clases Abstractas

Una clase Abstracta es aquella que no se puede instanciar.

En Python se encuentra el modulo ABC, lo cual permite definir métodos abstractos dentro de una clase abstracta.

© Universidad Estatal de Milagro – UNEMI

Dado el siguiente ejemplo:

```
1   #Clase abstracta
2   from abc import ABCMeta,abstractclassmethod
3   import math
4
5   class Figura(metaclass=ABCMeta):
6       @abstractclassmethod
7       def area(self):
8           pass #vacio
9
10      @abstractclassmethod
11      def perimetro(self):
12          pass #vacio
13
14  class Rectangulo(Figura):
15      def __init__(self, ancho, altura):
16          self.ancho = ancho
17          self.altura = altura
18
19      def area(self):
20          return self.ancho*self.altura
21
22      def perimetro(self):
23          return 2 * (self.ancho*self.altura)
24
25  class Circulo(Figura):
26      def __init__(self, radio):
27          self.radio = radio
28
29      def area(self):
30          return math.pi * self.radio **2
31
32      def perimetro(self):
33          return 2 * math.pi * self.radio
```

Figure 1: Ejemplo Clase Abstracta

Fuente: (George, 2019)

En el ejemplo de la Figura 1 se observa que la clase abstracta Figura tiene dos métodos abstractos área y perímetro los cuales no hay implementación, son vacíos, dado que las clases que heredan de Figura que son Rectángulo y Circulo deben implementar área y perímetro ya que es obligatorio.

© Universidad Estatal de Milagro – UNEMI

Las clases hijas deben implementar todos los métodos abstractos de Figura.

```
35  #prueba
36  r = Rectangulo(3, 5)
37  print(r.area())
38  print(r.perimetro())
39
40  print()
41  c = Circulo(5.0)
42  print(c.area())
43  print(c.perimetro())
44
```

Figure 2: Prueba Ejemplo Clase Abstracta

Fuente: (George, 2019)

En la Figura 2 se observa la instanciación de las clases Rectángulo y Circulo

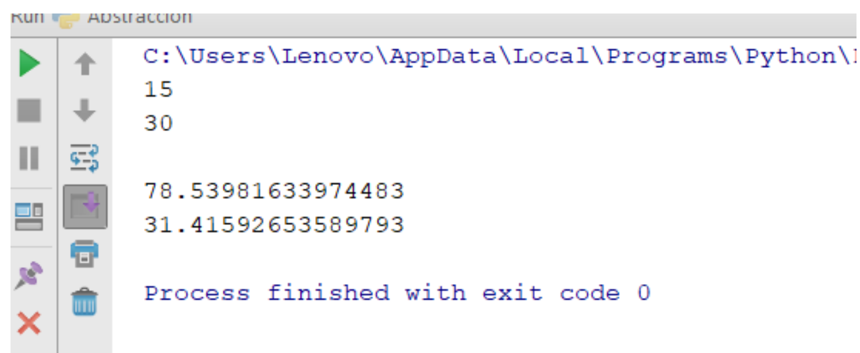

Figure 3: Ejecutando Clase Abstracta

Fuente: Elaborado por el autor

© Universidad Estatal de Milagro – UNEMI

Como se observa en la Figura 3, muestra el resultado de las clases hijo

Pero si se crea una instancia de la clase abstracta padre como se observa en la Figura 4 da un error.

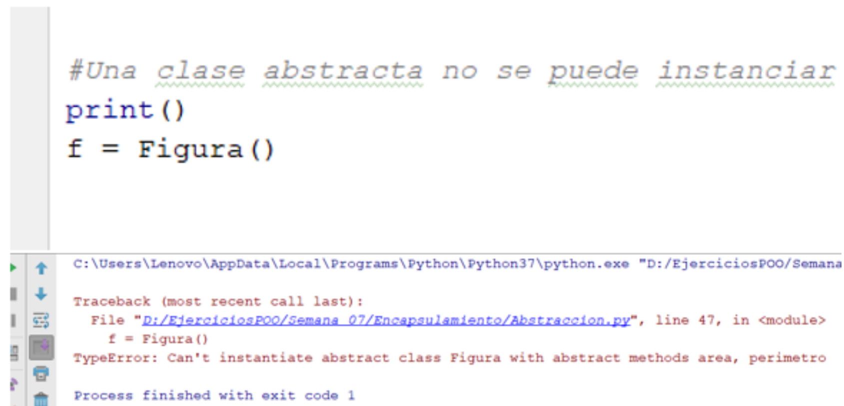

Figure 4: Error Instanciar Clase Abstracta

Fuente: Elaborado por el autor

Dado eso una clase abstracta no se puede instanciar.

Encapsulamiento

Según González Duque (2014) el encapsulamiento “se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase” (p. 48).

En otras palabras, es poder definir el grado de acceso y modificación a los atributos y métodos. En Python los accesos son públicos y se aplica el concepto de encapsulamiento mediante las convenciones. Para hacer un atributo o método privado se coloca dos guiones bajos antes del nombre del atributo o método “__nombre”

© Universidad Estatal de Milagro – UNEMI

Dada la clase Ejemplo con un método público y privado

```
1   class Ejemplo():
2       def publico(self):
3           return "Soy un metodo publico, a la vista de todo"
4       def __privado(self):
5           return "Soy un metodo privado, para ti no existo"
6
7   objeto = Ejemplo()
8   print(objeto.publico())
9   print(objeto.__privado())
10
```

```
Encapsulamiento
C:\Users\Lenovo\AppData\Local\Programs\Python\Python37\python.e:
Soy un metodo publico, a la vista de todo
Traceback (most recent call last):
  File "D:/EjerciciosPOO/Semana 07/Encapsulamiento/Encapsulamie:
    print(objeto.__privado())
AttributeError: 'Ejemplo' object has no attribute '__privado'

Process finished with exit code 1
```

Figure 5: Ejemplo Encapsulamiento

Fuente: (Unipython, n.d.)

Se observa en la Figura 5 que al instanciar un objeto privado da un error, dado que le objeto no puede visualizar un atributo privado

La forma correcta de acceder a estos métodos privados es por los métodos de acceso getter y setter.

© Universidad Estatal de Milagro – UNEMI

```
14  class Ejemplo():
15      def __init__(self):
16          self.__oculto = "Me encuentro oculto en la clase"
17      def publico(self):
18          return "Soy un metodo publico, a la vista de todo"
19      def __privado(self):
20          print("Dentro de la clase todos me pueden ver")
21      def get_oculto(self):
22          return self.__oculto
23      def set_oculto(self):
24          self.__oculto = self.__privado()
25
26  objeto = Ejemplo()
27  objeto.set_oculto()
```

```
C:\Users\Lenovo\AppData\Local\Programs\Python\Python
Dentro de la clase todos me pueden ver

Process finished with exit code 0
```

Figure 6: Ejemplo Encapsulamiento

Fuente: (Unipython, n.d.)

Como se observa en la Figura 6, gracias a los getter y setter se pueden acceder a los métodos privados.

© Universidad Estatal de Milagro – UNEMI

2.2 Herencia y Polimorfismo

Herencia

Según Severance (2016) la herencia tiene “la capacidad de crear una nueva clase extendiendo una clase existente” (p. 183). En otras palabras, permite crear una clase a partir de otra existente.

Dada la clase Animal:

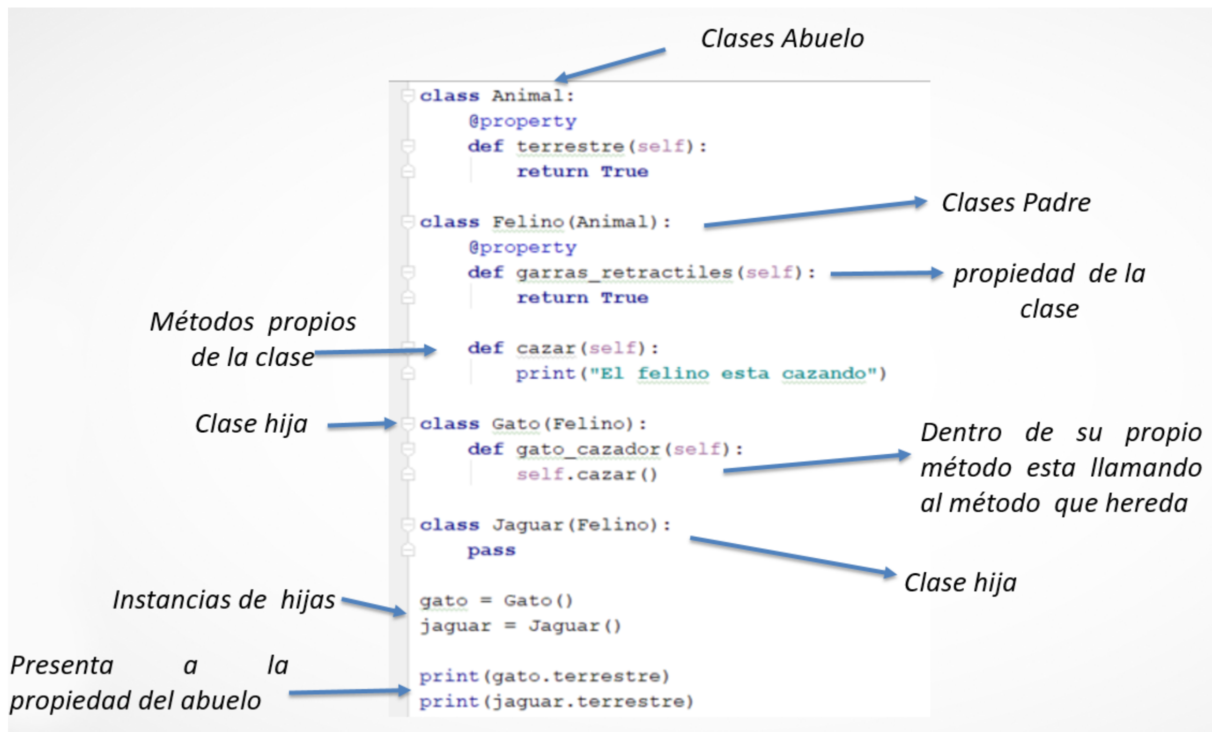

Figure 7: Ejemplo Herencia

Fuente: Elaborado por el autor

© Universidad Estatal de Milagro – UNEMI

Como se observa en la Figura 7, la clase Jaguar heredada de la clase Felino y la clase Felino hereda de la clase Animal.

Herencia Múltiple

En Python es posible usar herencia múltiple. Es cuando una clase hija hereda de más de 2 clases padres (Rossum, 2017).

Funciona igual que la herencia los atributos y métodos se las puede usar en las clases hijas.

Figure 8: Herencia Múltiple

Fuente: Elaborado por el autor

© Universidad Estatal de Milagro – UNEMI

Como se observa en la Figura 8, la Clase Gato hereda de la clase Felino y Macota

Polimorfismo

Según Rodó (n.d.) se denomina polimorfismo a “la propiedad que tiene muchos lenguajes de ejecutar código distinto en función del objeto que hace la llamada” (p. 31).

Dado el siguiente ejemplo sin aplicar polimorfismo:

```
#Sin aplicar polimorfismo
class Coche:
    def desplazamiento(self):
        print("Me desplazo utilizando cuatro ruedas")

class Moto:
    def desplazamiento(self):
        print("Me desplazo utilizando dos ruedas")

class Camion:
    def desplazamiento(self):
        print("Me desplazo utilizando seis ruedas")


miVehiculo = Moto()
miVehiculo.desplazamiento()

miVehiculo2 = Coche()
miVehiculo2.desplazamiento()

miVehiculo3 = Camion()
miVehiculo3.desplazamiento()
```

Figure 9: Ejemplo Polimorfismo

Fuente: Elaborado por el autor

© Universidad Estatal de Milagro – UNEMI

Como se observa en la figura 9, se crearon tres clases Coche, Moto, Camión, donde las tres tienen un método llamado desplazamiento, pero con comportamientos diferentes

Además, se crearon 3 objetos y se ejecutó el método desplazamiento de cada uno de los objetos

Al ejecutar el código de la Figura 9:

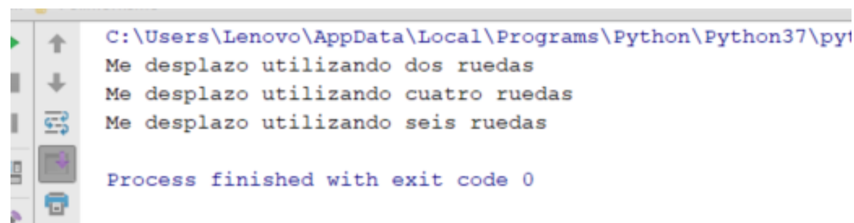

Figure 10: Ejecutando Ejemplo Polimorfismo

Fuente: Elaborado por el autor

Se observa en la Figura 10, los métodos desplazamiento llamados de las diferentes clases, si se desea crear más objetos se puede seguir la semántica de la Figura 9 o realizar el uso del polimorfismo.

Para usar polimorfismo se crea una función como se muestra en la Figura 11, línea 14-15

```
1  #aplicando polimorfismo
2  class Coche:
3      def desplazamiento(self):
4          print("Me desplazo utilizando cuatro ruedas")
5
6  class Moto:
7      def desplazamiento(self):
8          print("Me desplazo utilizando dos ruedas")
9
10 class Camion:
11     def desplazamiento(self):
12         print("Me desplazo utilizando seis ruedas")
13
14 def desplamientoVehiculo(vehiculo):
15     vehiculo.desplazamiento()
16
17 miVehiculo = Camion()
18 desplamientoVehiculo(miVehiculo)
19
```

Figure 11: Ejemplo polimorfismo

Fuente: Elaborado por el autor

© Universidad Estatal de Milagro – UNEMI

Como se observa en la Figura 11, el objeto vehículo puede adquirir el rol de cualquiera de los vehículos existentes.

Si se quiere que un objeto pase a comportarse como otro objeto, se debe volver a definir su clase.

```
17  miVehiculo = Camion()
18  desplamientoVehiculo(miVehiculo)
19
20  #Si queremos el un objeto se comporte como otro objeto, se
21  #debe volver a definir su clase
22  miVehiculo = Coche()
23  desplamientoVehiculo(miVehiculo)
24
25  miVehiculo = Moto()
26  desplamientoVehiculo(miVehiculo)
27
```

Figure 12: Cambio comportamiento objeto

Fuente: Elaborado por el autor

Al ejecutar el código de la Figuro 12 se observa:

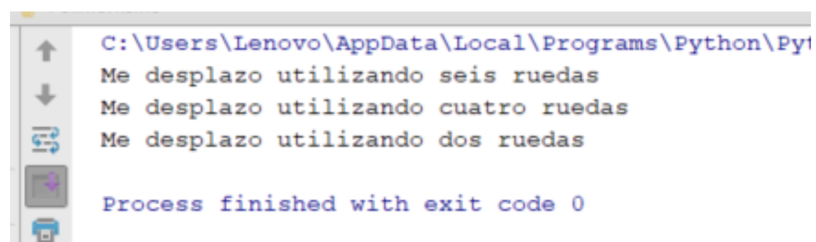

Figure 13: Ejecutando ejemplo polimorfismo

Fuente: Elaborado por el autor

Donde el resultado es el mismo de la Figura 9, pero aplicando el uso de polimorfismo.

© Universidad Estatal de Milagro – UNEMI

3.Recursos complementarios

Los siguientes recursos complementarios son sugerencias para que se pueda ampliar la información sobre el tema trabajado, como parte de su proceso de aprendizaje autónomo:

- (n.d.). Paradigma de la programación orientada a objetos. Retrieved from https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teoria/concepts.html
- (n.d.). Encapsulación | Curso de Python | Hektor Profe. Retrieved from https://docs.hektorprofe.net/python/programacion-orientada-a-objetos/encapsulacion/
- (n.d.). ¿Por qué Python está orientado a objetos? Retrieved from https://www.cursosgis.com/por-que-python-esta-orientado-a-objetos/

© Universidad Estatal de Milagro – UNEMI

4. Bibliografía

- George, J. (2019). *Clases base abstractas en Python (abc)*. 2019. https://www.tutorialspoint.com/abstract-base-classes-in-python-abc
- González Duque, R. (2014). Python para todos. *Creative Commons Reconocimiento*, *2*.
- Joyanes Aguilar, L. (2003). *Fundamentos de programación: algoritmos y estructura de datos y objetos*. http://combomix.net/wp-content/uploads/2017/03/Fundamentos-de-programación-4ta-Edición-Luis-Joyanes-Aguilar-2.pdf
- Rodó, D. M. (n.d.). *El lenguaje Python*. https://www.exabyteinformatica.com/uoc/Inteligencia_artificial/Inteligencia_artificial_avanzada/Inteligencia_artificial_avanzada_(Modulo_2).pdf
- Rossum, G. van. (2017). El tutorial de Python. *Recuperado de: Http://Docs. Python. Org. Ar/Tutorial/Pdfs/TutorialPython3. Pdf*. http://docs.python.org.ar/tutorial/pdfs/TutorialPython3.pdf
- Severance, C. R. (2016). *Python para todos*. http://do1.dr-chuck.com/pythonlearn/ES_es/pythonlearn.pdf
- Unipython. (n.d.). *Unipython - POO Programación Orientada a Objetos en Python*. Retrieved December 26, 2019, from https://unipython.com/programacion-orientada-objetos-python/

© Universidad Estatal de Milagro – UNEMI