UTN.BA | Centro de e-Learning
Universidad Tecnológica Nacional
Facultad Regional Buenos Aires
Secretaría de Cultura y Extensión Universitaria

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Diplomatura en Python:

Python nivel intermedio

UTN.BA | Centro de e-Learning
Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 1:

Nivel Intermedio I

Unidad 1:

POO - Programación Orientada a Objetos I.

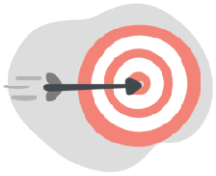
Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Presentación

Como todo lenguaje moderno, Python adopta tanto el paradigma de programación estructurado como el de Programación Orientado a Objetos – POO. Este paradigma surgió a principios de la década de los 90s y pretende comprender la programación de una forma más amena para el programador, introduciendo conceptos de objetos, clases, herencia, abstracción etc.

En el transcurso de esta unidad y las siguiente comenzaremos a ver las bases de este paradigma aplicado a python.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Objetivos

Que los participantes logren…

Comprendan los conceptos básicos de la POO.

Adopten las nociones básicas del paradigma.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bloques temáticos

1.- Introducción a POO.

2.- La clase object.

3.- Método __init__().

4.- Herencia múltiple.

5.- Uso de super.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

1. Introducción a POO

La programación orientada a objetos (POO) es un paradigma de programación introducido en la década del 90 que facilita y simplifica el diseño de una aplicación o programa. Posee varias ventajas con relación a la programación estructurada, entre las que se destaca la posibilidad de reutilización de código, sin embargo la velocidad de ejecución puede verse disminuida por lo que en aplicaciones de cálculo numérico es necesario analizar la conveniencia o no de su uso, en este caso particular un mix entre programación estructurada y POO podría ser lo más recomendable.

Este tipo de programación se basa en cuatro técnicas principales, sin contar la herencia de la cual trataremos más adelante, y son: ***abstracción , polimorfismo, acoplamiento (principio de ocultación) y encapsulamiento***.

La POO fue introducida para permitirnos programar de forma similar a cómo interaccionamos con el mundo que nos rodea, comprendiendo a cada parte del código como si fuera un objeto, por lo que su utilización, (salvada la barrera de aprender nuevas estructuras de programación) debería ser más intuitiva.

¿Cómo pensar en objetos?

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o estacionar. Pues en un esquema POO el coche sería el objeto, las propiedades (atributo o variable) serían las características como el color o el modelo y los métodos (operaciones o funciones) serían las funcionalidades asociadas como ponerse en marcha o parar.

Vale aclarar, que los objetos son únicos, no existen dos objetos iguales, de la misma forma que no existen dos personas iguales, por más que sean mellizos o en el mundo de hoy, un clon, cada objeto es único.

¿Qué es una clase?

La POO está formada básicamente por clases, estas estructuras, que vamos a utilizar para agrupar los atributos y métodos de un determinado tipo de objeto, nos van a permitir definir al mismo. Cuando programamos un objeto y definimos sus características y funcionalidades en realidad lo que estamos haciendo es programar una clase.

¿Cómo definimos una clase en Python?

Vayamos paso a paso, comencemos por mostrar un ejemplo de declaración de clase:

```
class PrimerClase:
    atributo1 = 2
objeto = PrimerClase()


print(objeto.atributo1)
```

Como vemos podemos indicar los siguientes componentes de la clase:

1.- Palabra reservada al inicio “class”

2.- Nombre de la clase “PrimerClase”

3.- Dos puntos “:”

A continuación para crear un objeto de dicha clase se llama a la clase de forma similar a como llamaríamos a una función, es decir por su nombre seguido de paréntesis:

```
objeto = PrimerClase()
```

Para poder acceder al atributo definido dentro de la clase lo podemos hacer mediante lo que se denomina notación de punto, es decir escribir el nombre del objeto, luego un punto y luego el atributo al cual hacemos referencia:

```
objeto.atributo1
```

Herencia

Las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.

Podemos considerar que una relación entre superclases y subclases. Las subclases heredan atributos y operaciones de las superclases. Esto nos permite escribir una sola vez determinadas operaciones y aplicarlas a distintas subclases.

Consideremos como en la imagen siguiente una superclase “bicicletas” y dos subclases “ConTraccionTrasera” y “ConTraccionTrasera”.

Por ejemplo ambas subclases podrían heredar de la clase bicicletas, el método (las funciones escritas dentro de una clase de denominan métodos) frenar.

Veamos un ejemplo de cómo escribir una herencia en python.

```
class ClasePadre1:
    atributo1 = 2
class ClasePadre2:
    atributo2 = 3

class ClaseHija(ClasePadre1, ClasePadre2):
    atributo3 = 4

    def imprimir(self, nombre):
        self.nombre=nombre
        print(self.nombre)

objeto = ClaseHija()
print(objeto.atributo1)
print(objeto.atributo2)
print(objeto.atributo3)
objeto.imprimir('juan')
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Como podemos observar, Python permite la herencia múltiple, en el ejemplo anterior la "ClaseHija" posee dos superclases o clases padre. Podemos observar que el objeto creado puede tener acceso a los atributos de sus superclases (también de sus métodos).

Hemos agregado también un método (función) dentro de la clase hija "imprimir()" que tiene definida una variable "nombre", notemos que tenemos acceso al método de forma similar al atributo mediante la notación de punto.

Nota: Luego entraremos en detalle en la forma correcta de escribir los métodos y las variables de clase, por ahora notemos el uso de self, la cual debe estar presente cada vez que se recibe una variable.

Abstracción

Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características.

La abstracción expresa las características esenciales de un objeto, que permite la diferenciación entre estos.

Si aplicamos la característica de abstracción a los autos:

1.- El objeto sería un auto.

2.- Las propiedades sería su color, marca, etc.

3.- Los métodos serían arrancar, parar, …

Encapsulamiento

Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente [1].

Principio de Ocultamiento - Acoplamiento

Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos [1].

Este punto introduce una diferencia sustancialmente importante entre Python y otros programas como PHP, JAVA, C++, etc.

En lo que se trata de la visibilidad de atributos y métodos, en Python no es posible declararlos como privados, protegidos o públicos, sino que son todos públicos. Los desarrolladores de Python han considerado que el programador en base a una buena documentación, asignación de nombres y estructura de trabajo se puede encargar de que los métodos y atributos no estén accesibles para el usuario común. La declaración de un atributo o método privado, más que para impedir el acceso a los datos por un hacker, se han establecido para que por error no modifiquemos datos sensibles a nuestro programa.

En python lo que si existe es la utilización de un doble guión bajo previo al nombre de un atributo o método (pero que no pueden finalizar con un doble guión bajo ya que estos están reservados en Python). Por convención el doble guión bajo al inicio indica que el atributo o método no puede ser llamado directamente y que se trata de un dato que no debe ser modificado por error, sin embargo si puede ser accedido si así lo quisiéramos.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

privadoa.py

```
class AccesoPrivado(object):


def __privado(self):
    print("Método privado ")


def getPrivado(self):
    self.__privado()


objeto = AccesoPrivado()
objeto.getPrivado()
objeto._AccesoPrivado__privado()
```

Dada la clase “AccesoPrivado” que hereda de la clase object (en realidad todas las clases en python heredan de la clase object aún cuando no lo especifique explicitamente), se considera un método __privado(), el cual si intentamos acceder desde fuera mediante:

```
objeto.__privado()
```

Nos retorna un error.

Hasta aquí uno (sobre todo los que vienen de lenguajes como JAVA y PHP en donde se declaran métodos y variables privadas mediante la palabra reservada “private”) podría pensar que el método es privado y que por eso no puedo accederlo desde fuera, con lo cual para poder acceder a su contenido, debo crear una función pública “getPrivado()” y accederlo a través de esta (esta en una práctica común en otros lenguajes), sin embargo en Python no se puede declarar un método o variable como privado y es posible accederlo como:

```
objeto._AccesoPrivado__privado()
```

Anteponiendo el nombre de la clase, precedido de un guión bajo.

En realidad la declaración de métodos y variables privados, no se establece para evitar que un Hacker experimentado acceda a determinadas variables o métodos, sino que se utiliza para que no modifiquemos datos sensibles por error al crear el código, ni lo pueda hacer el usuario de la API sin querer, por eso cuando indicamos un método o variable con el doble guión bajo, lo que estamos diciendo es “ojo, mira que lo que vas a modificar el creador del script lo considero sensible de ser modificado”. Existen formas que veremos posteriormente de que al importar un paquete estos métodos o variables no se encuentren disponibles a no ser que se conozca su nombre, pero esto lo veremos más adelante.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Polimorfismo

Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O, dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado [1].

Pasemos a ver un ejemplo:

polimorfismo.py

```
class ClasePadre1:
    atributo1 = 2

    def imprimir(self, nombre):
        self.nombre=nombre
        print("Este es un nombre: " + self.nombre)

class ClaseHija(ClasePadre1):
    atributo2 = 4

    def imprimir(self, nombre):
        self.nombre=nombre
        print(self.nombre)
```

```
objeto1 = ClasePadre1()
print(objeto1.atributo1)
objeto1.imprimir('Juan')
objeto2 = ClaseHija()
print(objeto2.atributo1)
print(objeto2.atributo2)
```

En este caso, tanto en la clase padre como en la clase hija se encuentran definidos los métodos "imprimir()", sin embargo vemos que son diferentes y que cuando creamos y objeto de la clase hija, el método imprimir() que se ejecuta, pisa al método definido en la clase padre.

La salida del script anterior da:

```
2
Este es un nombre: Juan
2
4
Marcelo
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

1. La clase object

Cada clase definida en Python hereda de la clase "object", por lo que puede utilizar los métodos establecidos en dicha clase.

Supongamos la siguiente clase, en la cual lo único que se encuentra definido es un valor de variable c1:

clase_object.py

```
class Auto:

    color = "azul"


objeto = Auto()

print(objeto.color)


print(dir(Auto))

input()
```

Salida:

```
azul

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'color']
```

Podemos ver dos cosas interesantes, la primera es la definición de un objeto de la clase “Auto” y la impresión del valor de “color” mediante el método print. En este caso la inicialización se está realizando a través del método __init__ de la clase “object”.

Lo segundo es que al utilizar el método dir() vemos que existen varios métodos asociados a “Auto” que no hemos definido, dichos métodos que comienzan y terminan con un doble guión bajo, son métodos de la clase object, que se encuentran asociados implícitamente a la clase Auto.

Nota: De hecho, podemos ver con el agregado del siguiente código, que la clase Auto tiene como clase padre o superclase a la clase "object", y que en si es un objeto de la clase "type"

clase_object2.py

```
class Auto:

    color = "azul"

# Instanciamos e imprimimos el objeto de la clase Auto

objeto = Auto()

print(objeto.color)

# Imprimimos los métodos de la clase Auto

print(dir(Auto))

# Vemos cual es su clase padre

print(Auto.__class__.__base__)

# Vemos si es un objeto de alguna clase

print(Auto.__class__)

input()
```

Salida:

```
azul

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'color']

<class 'object'>

<class 'type'>
```

2. Método __init__().

Cuando trabajamos con programación orientada a objetos POO, una de las cosas más importantes a tener en cuenta, es la forma en que creamos dichos objetos. Como veremos un poco más adelante la forma correcta de crear los objetos es mediante la utilización de un método (función) __init__ el cual es un método reservado de Python que cumple la función de constructor de la clase a la cual se encuentra asociada, es decir una función destinada a inicializar (crear) los objetos de la clase utilizada.

La función __init__ puede tomar diferente tipo de argumentos, y para su correcta utilización, debemos comprender como vimos, que cada clase de Python hereda en forma implícita de la clase "object", es decir que "object" es una superclase de cada clase definida en Python. Si no implementamos un método __init__ en nuestra clase, por defecto se ejecuta el método __init__ de la clase object.

Es posible, aunque no es la forma correcta crear variables de instancia sin utilizar un constructor como ejemplo veamos el siguiente caso en donde podemos ver que para trabajar con variables de instancia se utiliza dentro de la clase la palabra clave "self"

init1.py

```
class Comentario:
    def imprimir( self ):
        print(self.texto)


objeto = Comentario()
objeto.texto = "Hola variable de instancia"
objeto.imprimir()
input()
```

3. Herencia múltiple

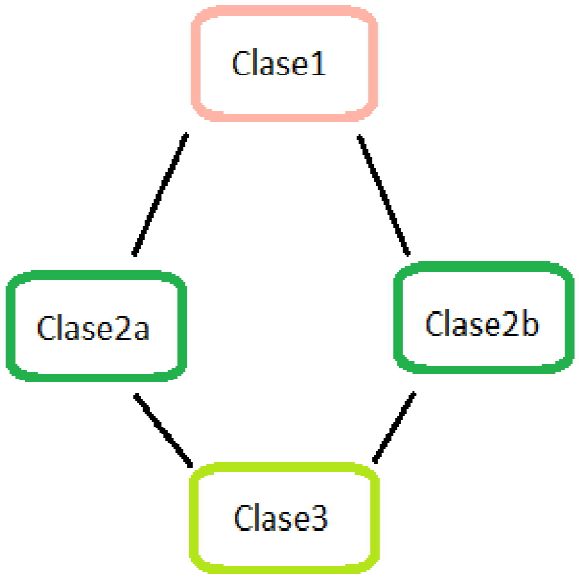
Python soporta herencia múltiple, es decir que una clase puede heredar de más de una clase, para indicarlo, lo único que tenemos que hacer es separar mediante una coma cada una de las clases de las cuales se hereda en la definición de la clase. Así podríamos tener la clase “Comentario” que tiene “N” superclases de la siguiente forma.

```
class Comentario(Superclase1, Superclase2,.....,SuperclaseN):
def imprimir( self ):
print(self.texto)
```

Al trabajar con herencia múltiples, debemos tener mucho cuidado, para no cometer errores, ya que si dos superclases poseen el mismo nombre de método en su definición, ¿de cuál de las dos hereda el método la clase hija?.

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Estas cuestiones las iremos resolviendo con ejemplos y al estudiar el uso de "**super**", ahora nos ocuparemos de un problema denominado "**problema del diamante**" el cual se da cuando una clase tiene dos superclases, las cuales tiene una superclase en común. Veamos el siguiente esquema, de tres niveles, en donde la Clase3 hereda de las Clase2a y Clase2b y en donde estas dos últimas a su vez heredan de la Clase1:



Supongamos ahora que generamos una instancia de la Clase3 e implementamos un método declarado en la Clase1, ¿la herencia se lleva a cabo a través de la rama que contiene a Clase2a o la que contiene la Clase2b?

Python toma la convención de crear una lista de clases que:

1.- Busca de izquierda a derecha, y de abajo hacia arriba.

Clase3, Clase2a Clase1, Clase2b Clase1

2.- Se eliminan todas las apariciones repetidas de una clase salvo la última.

Clase3, Clase2a **Clase1**, Clase2b Clase1

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

queda: Clase3, Clase2a Clase2b, Clase1

Esto lo podemos ver si utilizamos el método reservado **__mro__,** veámoslo en el siguiente ejemplo:

herenciaMultiple.py

```
class Clase1():
    def tipo(self):
        print("Soy clase1")


class Clase2a(Clase1):
    def tipo(self):
        print("Soy clase2a")


class Clase2b(Clase1):
    def tipo(self):
        print("Soy clase2b")


class Clase3(Clase2a, Clase2b):
    def tipo(self):
        print("Soy clase3")


print(Clase3.__mro__)
```

Como era de esperar la salida nos retorna:

```
(<class '__main__.Clase3'>, <class '__main__.Clase2a'>, <class '__main__.Clase2b'>, <class '__main__.Clase1'>, <class 'object'>)
```

4. Uso de super

Cuando queremos invocar a un método de una superclase desde una clase hija, podemos utilizar la palabra reservada “super”, veamos un ejemplo:

super.py

```
class Vehiculo(object):
    def tipo(self):
        print("Dos ruedas")

class Material(object):
    def tipo(self):
        print("plástico")

class Moto(Vehiculo, Material):
    def modelo(self):
        print("Modelo 1")
```

```
        super(Moto, self).tipo()
        super().tipo()
        Material.tipo(self)
        Vehiculo.tipo(self)


class Bicicleta(Material, Vehiculo):
    def modelo(self):
        print("Modelo 2")
        super(Bicicleta, self).tipo()
        super().tipo()
        Material.tipo(self)
        Vehiculo.tipo(self)
print("---------- Prioridad Clases ----------")
print(Moto.__mro__)
print(Bicicleta.__mro__)
print("---------- para objeto 1 ----------")
objeto1 = Moto()
objeto1.modelo()
objeto1.tipo()
print("---------- para objeto 2 ----------")
objeto2 = Bicicleta()
objeto2.modelo()
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

```
objeto2.tipo()
```

La salida nos retorna:

```
---------- Prioridad Clases ------------

(<class '__main__.Moto'>, <class '__main__.Vehiculo'>, <class '__main__.Material'>, <class 'object'>)

(<class '__main__.Bicicleta'>, <class '__main__.Material'>, <class '__main__.Vehiculo'>, <class 'object'>)

---------- para objeto 1 ------------

Modelo 1

Dos ruedas

Dos ruedas

plástico

Dos ruedas

Dos ruedas

---------- para objeto 2 ------------

Modelo 2

plástico

plástico

plástico

Dos ruedas

plástico
```

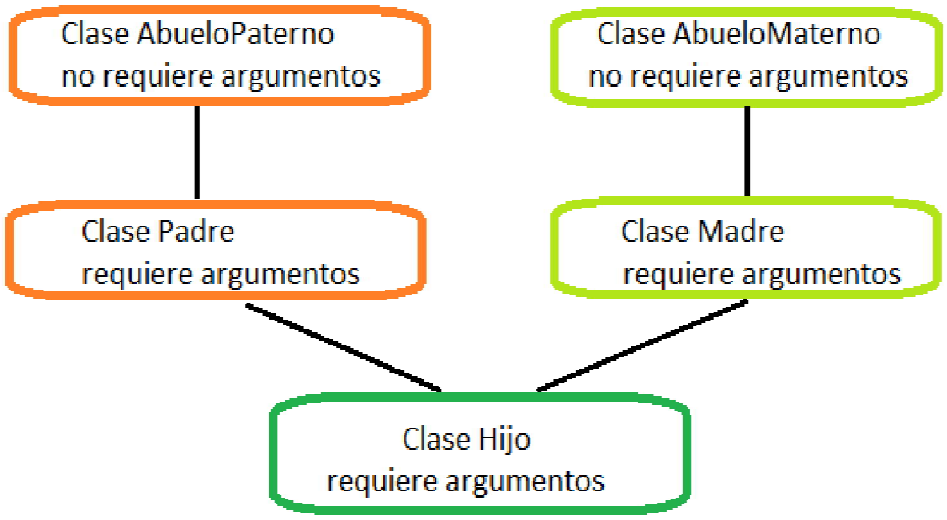
Nota: En python 3, la sintaxis se simplifica de super(ClaseHija,self).__init__() a super().__init().

Nota: En python 3 podemos llamar al método de la clase padre desde dentro de la clase hija, directamente indicando el nombre de la clase padre de la cual queremos heredar, y con notación de punto el nombre del método a utilizar:

Ejemplo: **Vehiculo.tipo(self)**

Tener en cuenta al usar "super" con argumentos.

Tenemos que tener cuidado cuando trabajamos con argumentos, ya que en ocasiones se pueden dar casos en los cuales se pueden perder argumentos por el camino y dejar de estar presentes clases que los requieren. En el ejemplo siguiente si la clase Hijo requiere argumentos, estos van a tener que ser ingresados y van a ser pasados a la clase Padre, dado que la clase AbueloPaterno no los requiere, estos se pierden y cuando la clase Madre los requiere ya no están disponibles.



El código del esquema anterior queda como sigue:

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

super.py

```
class AbueloPaterno(object):
    def __init__(self):
        print("AbueloPaterno")
        super(AbueloPaterno, self).__init__()

class AbueloMaterno(object):
    def __init__(self):
        print("AbueloMaterno")
        super(AbueloMaterno, self).__init__()

class Padre(AbueloPaterno):
    def __init__(self, arg):
        print("Padre", "arg = ", arg)
        super(Padre, self).__init__()
class Madre(AbueloMaterno):
    def __init__(self, arg):
        print("Madre", "arg = ", arg)
        super(Madre, self).__init__()
```

```
class Hijo(Padre, Madre):
    def __init__(self, arg):
        print("Hijo", "arg = ", arg)
        super(Hijo, self).__init__(arg)



objeto = Hijo("celeste")
```

Para corregir este problema, lo que podemos hacer es utilizar el * y ** para permitir que las superclases permitan el pasaje de argumentos, aún cuando ellas no los requieran.

super2.py

```
class AbueloPaterno(object):
    def __init__(self, *args, **kwargs):
        print("AbueloPaterno")
        super(AbueloPaterno, self).__init__(*args, **kwargs)


class AbueloMaterno(object):
    def __init__(self, *args, **kwargs):
        print("AbueloMaterno")
        super(AbueloMaterno, self).__init__(*args, **kwargs)
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

```
class Padre(AbueloPaterno):
    def __init__(self, arg, *args, **kwargs):
        print("Padre", "arg = ", arg)
        super(Padre, self).__init__(arg, *args, **kwargs)

class Madre(AbueloMaterno):
    def __init__(self, arg, *args, **kwargs):
        print("Madre", "arg = ", arg)
        super(Madre, self).__init__(*args, **kwargs)

class Hijo(Padre, Madre):
    def __init__(self, arg, *args, **kwargs):
        print("Hijo", "arg = ", arg)
        super(Hijo, self).__init__(arg, *args, **kwargs)
objeto = Hijo("celeste")
```

Centro de e-Learning SCEU UTN - BA. Medrano 951 2do piso
(1179) // Tel. +54 11 7078- 8073 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Bibliografía utilizada y sugerida

Libros y otros manuscritos:

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

Manual online

https://docs.python.org/3.7/tutorial/

https://docs.python.org/3.7/library/index.html

https://docs.python.org/3/distutils/introduction.html