PROGRAMACIÓN II - UNIDAD 2

Ing. Gastón Weingand (gaston.weingand@uai.edu.ar)

Agenda

1. Desarrollo y utilización de clases y objetos. Variables de clases e instancias.
2. Constructores y destructores.
3. Encapsulamiento.
4. Herencia simple y herencia múltiple.
5. Polimorfismo.
6. Clases de “nuevo-estilo”.
7. Métodos especiales.
8. Tipos en Python.

Agenda

1. Desarrollo y utilización de clases y objetos. Variables de clases e instancias.
2. Constructores y destructores.
3. Encapsulamiento.
4. Herencia simple y herencia múltiple.
5. Polimorfismo.
6. Clases de “nuevo-estilo”.
7. Métodos especiales.
8. Tipos en Python.

4-Herencia Simple

```python
class Instrumento:
    def __init__(self, precio):
        print("Contruyo Instrumento")
        self.__precio = precio

    def tocar(self):
        print("Estamos tocando musica")

    def romper(self):
        print("Eso lo pagas tu")
        print("Son", self.__precio, "$$$")
```

pass: Definición sin implementación

```python
class Bateria(Instrumento):
    pass

class Guitarra(Instrumento):
    pass
```

4-Herencia Simple

Sobreescritura de constructor

```
class Bateria(Instrumento):
    def __init__(self, tipo_parche, precio):
        super().__init__(precio)
        print("Contruyo batería")
        self.__tipo_parche = tipo_parche
```

super() o el nombre de la clase padre para invocar el constructor con el parámetro

(En este caso hay que agregar el parámetro self también)

```
def __init__(self, tipo_parche, precio):
    #super().__init__(precio)
    Instrumento.__init__(self, precio)
```

4-Herencia Simple

Instanciando…

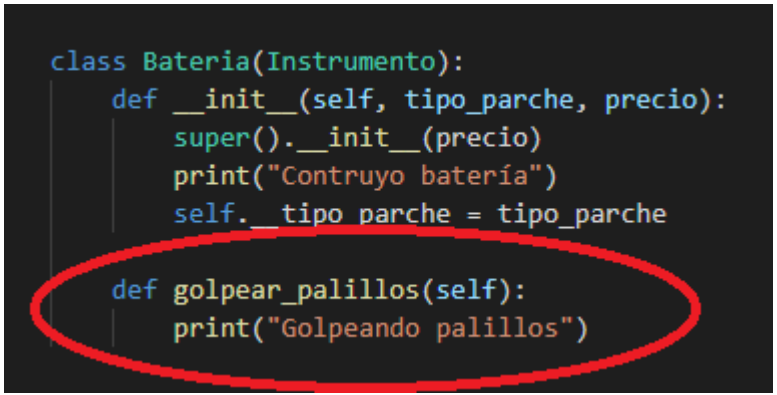
```
instrumento = Instrumento(5000)
bateria = Bateria("Blando", 5000)

instrumento.tocar()
instrumento.romper()
bateria.tocar()
bateria.romper()
```

4-Herencia Simple

Conversión de tipos

Primero agreguemos un método a la clase Bateria



4-Herencia Simple

Conversión de tipos

Luego instanciamos y convertimos…

```
instrumento = bateria #Permitido
instrumento.romper()
instrumento.golpear_palillos() #golpear_palillos es de batería por lo tanto esto funciona...

instrumento2 = Instrumento(5000)
bateria_mana = Bateria("Duro", 10000)
bateria_mana = instrumento2 #Python me permite asignar un tipo padre a hijo...
#bateria_mana.golpear_palillos() #Pero...Esta línea daría error ya que golpear_palillos no es de instrumento
```

4-Herencia Simple

Otro ejemplo…

```
class AgregarElemento(list): #Creamos una clase Agregarelemento heredando atributos de clase list
    def append(self, alumno): #Definimos que el método append (de listas) añadirá el elemento alumno
        print ("Añadido el alumno", alumno) #Imprimimos el resultado del método
        super().append(alumno) #Incorporamos la función super SIN INDICAR LA CLASE ACTUAL, seguida
                               #del método append para la variable alumno

lista1 = AgregarElemento() #Definimos la clase de nuestra lista llamada "Lista1"
lista1.append ('Matias') #Añadimos un elemento a la lista como lo haríamos normalmente
lista1.append('Jorge') #Otro elemento...
print(lista1)  # Imprimimos la lista para corroborar los alumnos...
```

4-Herencia Múltiple

```python
class Terrestre:
    def desplazar(self):
        print("El animal anda")

class Acuatico:
    def desplazar(self):
        print("El animal nada")

class Cocodrilo(Terrestre, Acuatico):
    pass

c = Cocodrilo()
c.desplazar()

#Salida -> El animal anda
```

¿A qué se debe esta salida de programa?

4-Herencia Múltiple

La función *issubclass()*
Para saber si una clase es subclase de otra, se utiliza la función *subclass()*

```
print(issubclass(Terrestre, Cocodrilo)) #False
print(issubclass(Cocodrilo, Acuatico)) #True
```

Agenda

1. Desarrollo y utilización de clases y objetos. Variables de clases e instancias.
2. Constructores y destructores.
3. Encapsulamiento.
4. Herencia simple y herencia múltiple.
5. Polimorfismo.
6. Clases de “nuevo-estilo”.
7. Métodos especiales.
8. Tipos en Python.

5-Polimorfismo

En Python al no ser necesario especificar explícitamente el tipo de los parámetros que recibe una función, las funciones son naturalmente polimórficas.

En otros lenguajes, puede darse que sólo algunas funciones específicas sean polimórficas (como en C++, por ejemplo)

5-Polimorfismo

```python
#Polimorfismo...

def frecuencias(secuencia):
    """ Calcula las frecuencias de aparición de los elementos de
        la secuencia recibida.
        Devuelve un diccionario con elementos: {valor: frecuencia}
    """
    # crea un diccionario vacío
    frec = dict()
    # recorre la secuencia
    for elemento in secuencia:
        frec[elemento] = frec.get(elemento, 0) + 1
    return frec

numeros = frecuencias([2,4,5,7,3,2])
print(numeros)
nombres = frecuencias(["pedro","juan","pedro","tito","pedro","juan"])
print(nombres)
```

5-Polimorfismo

```python
#Con clases

class Leon:
    def desplazar(self):
        print("Avanzo en 4 patas...")

class Bicicleta():
    def desplazar(self):
        print("Avanzo en 2 ruedas...")

leon = Leon()
leon.desplazar()

bicicleta = Bicicleta()
bicicleta.desplazar()
```

5-Polimorfismo

Agrego un método polimórfico…

```
leon = Leon()
#leon.desplazar()

bicicleta = Bicicleta()
#bicicleta.desplazar()

def mover(object):
    object.desplazar()

mover(leon)
mover(bicicleta)
```

5-Polimorfismo - Sobrecarga Métodos

```python
class Persona():

    def __init__(self):
        pass

    def mensaje(self):
        print("mensaje desde la clase Persona")

class Obrero(Persona):

    def __init__(self, horas):
        self.__horas_trabajo = horas

    def mensaje(self):
        print("mensaje desde la clase Obrero")

obrero_planta = Obrero(40)
obrero_planta.mensaje()
```

5-Polimorfismo - Sobrecarga Métodos

```
obrero_planta = Obrero(40)
obrero_planta.mensaje()

obrero_planta.__class__ = Persona #Trato al obrero como una persona
obrero_planta.mensaje()

obrero_planta.__class__ = Obrero #Trato nuevamente como obrero al objeto conservando el estado del objeto
obrero_planta.mensaje()
```

5-Polimorfismo - Sobrecarga Operadores

```python
#Sobrecarga de operadores
class Punto:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __add__(self, other): #Sobrecarga de método de suma
        x = self.x + other.x
        y = self.y + other.y
        return x, y

punto1 = Punto(4, 6)
punto2 = Punto(1, -2)
print (punto1 + punto2)
```

5-Polimorfismo - Sobrecarga Operadores

Operadores binarios

OPERATOR	MAGIC METHOD
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)

Operadores de comparación

OPERATOR	MAGIC METHOD
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

5-Polimorfismo - Sobrecarga Operadores

Operadores de asignación

OPERATOR	MAGIC METHOD
-=	__isub__(self, other)
+=	__iadd__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
%=	__imod__(self, other)
**=	__ipow__(self, other)

Operadores unarios

OPERATOR	MAGIC METHOD
–	__neg__(self, other)
+	__pos__(self, other)
~	__invert__(self, other)

Agenda

1. Desarrollo y utilización de clases y objetos. Variables de clases e instancias.
2. Constructores y destructores.
3. Encapsulamiento.
4. Herencia simple y herencia múltiple.
5. Polimorfismo.
6. Clases de “nuevo-estilo”.
7. Métodos especiales.
8. Tipos en Python.

6-Clases de nuevo estilo

La diferencia principal entre las clases antiguas y las de nuevo estilo consiste en que a la hora de crear una nueva clase anteriormente (Python 2) no se definía realmente un nuevo tipo.

```
class Fecha(object):
    def __init__(self):
        self.__dia = 1

    def getDia(self):
        return self.__dia

    def setDia(self, dia):
        if dia > 0 and dia < 31:
            self.__dia = dia
        else:
            print "Error"

    dia = property(getDia, setDia)
```

Se utilizaba object para poder asignar properties en pyhton 2. Esto ya no es necesario.