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.

7-Métodos especiales

__init__(self, args)

Método llamado después de crear el objeto para realizar tareas de inicialización

7-Métodos especiales

__new__(cls, args)

Método exclusivo de las clases de nuevo estilo que se ejecuta antes que __init__ y que se encarga de construir y devolver el objeto en sí. Es equivalente a los constructores de C++ o Java. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es self, sino la propia clase: cls

7-Métodos especiales

__del__(self)

Método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.

7-Métodos especiales

__str__(self)

Método llamado para crear una cadena de texto que represente a nuestro objeto. Se utiliza cuando usamos print para mostrar nuestro objeto o cuando usamos la función str(obj) para crear una cadena a partir de nuestro objeto.

7-Métodos especiales

__len__(self)

Método llamado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función len(obj) sobre nuestro objeto. Como es de suponer, el método debe devolver la longitud del objeto.

7-Métodos especiales

__cmp__(self, otro)

Método llamado cuando se utilizan los operadores de comparación para comprobar si nuestro objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores <=, > o >= se lanzará una excepción. Si se utilizan los operadores == o != para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).

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.

8-Tipos en python

En Python, todos los tipos son objetos.
Pero no en todos los lenguajes de programación es así.

En python tenemos dos casos particulares de tipos:

1) La definición de un tipo, indica cuáles son los atributos y métodos que van a tener todas las variables que sean de ese tipo. Esta definición se llama específicamente, la **clase** del objeto.

8-Tipos en python

2) A partir de una clase es posible crear distintas variables que son de ese tipo. A las variables que son de una clase en particular, se las llama **instancia** de esa clase.

8-Tipos en python

```
#Usando type para una clase
print(type(Persona))  ------>  <class 'type'>
print(type(str))      ------>  <class 'type'>

#Usando type para un objeto
s = "test"
persona1 = Persona()

print(type(s))        ------>  <class 'str'>
print(type(persona1)) ------>  <class '__main__.Persona'>
```

Para aprender más…

1) Uso de init vs new. Ejemplos.
2) Uso de clases abstractas. módulo abc. Ejemplos.
3) Uso de interfaces. Duck Typing.
4) Ordenar una lista de objetos con sort y lambda

Para aprender más…

1) Uso de init vs new. Ejemplos.

El método __init__ crea el objeto y luego lo inicializa, no es el constructor como tal, en cambio el método __new__ sólo construye el objeto. No obstante, en la mayoría de los casos se utiliza __init__

```
class A():
    def __new__(cls):
        print ("A.__new__ es llamado")
        return super(A, cls).__new__(cls)

    def __init__(self): #Init nuno puede devolver nada, salvo un None
        print ("A.__init__ es llamado")
        #return None

a = A()
```

```
A.__new__ es llamado
A.__init__ es llamado
```

Para aprender más…

1) Uso de init vs new. Ejemplos.

Si comento la línea de retorno de __new__ la clase no se inicializaría correctamente

```
class A():
    def __new__(cls):
        print ("A.__new__ es llamado")
        #return super(A, cls).__new__(cls)

    def __init__(self): #Init nuno puede devolver nada, salvo un None
        print ("A.__init__ es llamado")
        #return None

a = A()
```

```
A.__new__ es llamado
```

Para aprender más…

1) Uso de init vs new. Ejemplos.

Esto daría una excepción de devolución de tipos…

```
class A():
    def __new__(cls):
        print ("A.__new__ es llamado")
        return super(A, cls).__new__(cls)

    def __init__(self): #Init nuno puede devolver nada, salvo un None
        print ("A.__init__ es llamado")
        return 33

a = A()
```

Para aprender más…

1) Uso de init vs new. Ejemplos.

Un uso posible para new podría ser para el patrón singleton…

```
class SoyUnico():

    __instance = None
    nombre = None

    def __str__(self):
        return self.nombre

    def __new__(cls):
        if SoyUnico.__instance is None:
            SoyUnico.__instance = object.__new__(cls)
        return SoyUnico.__instance

ricardo = SoyUnico()
ricardo.nombre = "Ramon Valdes"
print(ricardo)
ramon = SoyUnico()
ramon.nombre = "Roberto Gomez"
print(ramon)
print(ricardo)
print(ramon)
```

Para aprender más…

2) Uso de clases abstractas. módulo abc. Ejemplos.
Ver ejemplo en la próxima unidad

Para aprender más…

3) Uso de interfaces. Duck Typing.

La tipificación de pato significa que una operación no especifica formalmente los requisitos que deben cumplir sus operandos, sino que simplemente lo prueba con lo que se le da.

Para aprender más…

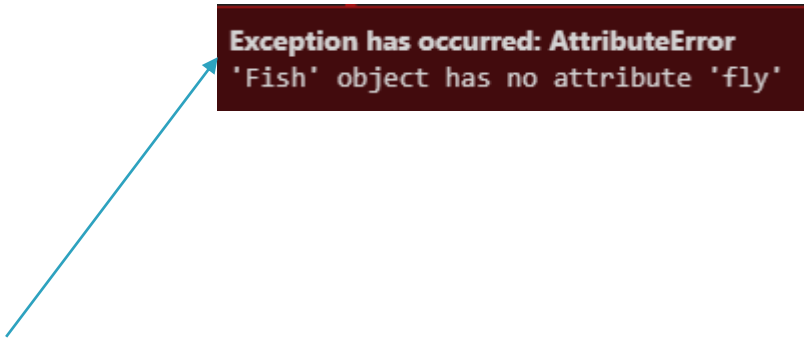
3) Uso de interfaces. Duck Typing.

```python
class Bird:
    def fly(self):
        print("fly with wings")

class Airplane:
    def fly(self):
        print("fly with fuel")

class Fish:
    def swim(self):
        print("fish swim in sea")

# Atributos con el mismo nombre
# se consideran como duck typing
for obj in Bird(), Airplane(), Fish():
    obj.fly() #En Fish daría excepción
```



Para aprender más…

4) Ordenar una lista de objetos con sort y lambda

Definimos la clase…

```python
from datetime import datetime #Para trabajar con fechas...

class Factura:

    def __init__(self, nro, fecha):
        self.__nro = nro
        self.__fecha = fecha

    def getFecha(self):
        return self.__fecha

    def setFecha(self, fecha):
        self.__fecha = fecha

    def getNro(self):
        return self.__nro

    def setNro(self, nro):
        self.__nro = nro

    nro = property(getNro, setNro)
    fecha = property(getFecha, setFecha)
```

Para aprender más…

4) Ordenar una lista de objetos con sort y lambda.

Ordenamos…

```
f1 = Factura(1, datetime(2019, 1, 1))
f2 = Factura(8, datetime(2019, 1, 5))
f3 = Factura(9, datetime(2019, 1, 2))
f4 = Factura(2, datetime(2019, 1, 9))

lista_facturas = [f1, f2, f3, f4]


facturas_ordenada_nro = sorted(lista_facturas, key=lambda objeto: objeto.nro)
print("Ordenamiento por nro:")
for s in facturas_ordenada_nro:
    print(s.nro, s.fecha)

facturas_ordenada_fecha = sorted(lista_facturas, key=lambda objeto: objeto.fecha)
print("Ordenamiento por fecha:")
for s in facturas_ordenada_fecha:
    print(s.nro, s.fecha)
```