Python para Análisis de datos: Introducción

Sesión 3

Alejandro Villar (avillar@ticnor.es)

22 Octubre 2018

Programación Orientada a Objetos

POO en Python

Ejercicios

Programación Orientada a Objetos

Diferentes paradigmas de programación

- **Orientado a objetos**
- Procedural
- Imperativo
- Funcional (<100%)

Qué es

- Concepto central: **objeto**
  - Datos (atributos o campos)
  - Comportamiento (métodos)

Objeto: Coche de Pedro

- Datos:
  - Color: Gris
  - Año: 2012
- Comportamiento:
  - Arrancar
  - Parar
  - Acelerar
  - Frenar

Clases y objetos

- Clase: Definición a partir de la que crear objetos (*contrato*).
  - Normalmente, mayúscula inicial
- Objeto: Instancia individual de una clase.
  - Normalmente, minúscula inicial (son variables)

Características de OOP

- Herencia
- Encapsulamiento
- Abstracción
- Polimorfismo
- Composición

Herencia

- Jerarquía: atributos y comportamiento
- La clase *Perro* hereda de *Mamífero* y ésta de *Animal*.
- Pero también de *Cuadrúpedo* <- Herencia múltiple
- Las subclases completan o modifican a sus clases ancestro

Encapsulamiento

- Limitar acceso al estado interno del objeto
- Campos y métodos públicos y privados
  - ¡Python no limita el acceso! -> Convenio

Abstracción

- No nos preocupamos de los detalles de funcionamiento interno
- *Confiamos* en que el objeto hará lo que le pedimos correctamente
- Ejemplo:
  - En Coche.arrancar() nos da igual el sistema de arranque interno, queremos que el coche pase a estar arrancado

Polimorfismo

- Tratamos a una familia de objetos de la misma forma
- Ejemplo 1: lo que haga mensaje.enviar() dependerá de su clase
  - SMS
  - Email
  - MensajeWhatsApp
- Ejemplo 2: si la clase Mensaje tiene campos "asunto" y "contenido", podemos implementar Bandeja.recibir(mensaje) sin importarnos la subclase concreta de mensaje.

Composición

- Combinación de varios objetos con diferentes características
- Ejemplo:
  - Coche contiene objetos de clase Motor, Freno, Rueda, etc.
  - coche.frenar() llama a frenos_delante.activar() y frenos_detras.activar()

POO en Python

Primero, un repaso

- `dict`: estructura clave -> valor
  - Claves únicas
  - Valores: cualquier cosa
  - Permite anidamiento

```
midic = {
    'clave1': 'valor1',
    'clave2': 23,
    'clave3': [1, 2, 3],
    'clave4': {
        'subclave1': [1, 2]
    }
}
```

La clase más simple:

```
class MiClase:
    pass
```

Una clase más completa:

```
class Persona:
    """Una clase para almacenar datos de personas"""

    nombre = None
    apellidos = None
    email = None


    def saludar(self):
        print("Hola, {0} {1}".format(self.nombre, self.apellidos))


    def linea_email(self):
        return "{1}, {0} <{2}>".format(self.nombre, self.apellidos, self.email)
```

```
>>> p = Persona()
>>> p.nombre = 'Alejandro'
>>> p.apellidos = 'Villar'
>>> p.email = 'avillar@ticnor.es'
>>> p.saludar()
Hola, Alejandro Villar
>>> print(p.linea_email())
Villar, Alejandro <avillar@ticnor.es>
```

`self` (I)

- Los métodos dentro de una clase tienen un argumento inicial: `self`
- Ese argumento representa a la **instancia** (objeto) que lo llama
- Python automáticamente lo añade a los argumentos de llamada
- Necesario para modificar atributos de la instancia.

self (II)

```
    def saludar(self):
        print("Hola, {0} {1}".format(self.nombre, self.apellidos))
```

```
>>> p.saludar()  # No hay self
```

Constructor

Realizar tareas de inicialización, según se crea la clase.

- Sin constructor (por defecto)
- Constructor sin argumentos.
- Constructor con argumentos.

Constructor sin argumentos

```
class Persona:
    def __init__(self):
        self.nombre = 'Pendiente'
        print("Persona inicializada")
```

```
>>> p = Persona()
Persona inicializada
>>> p.nombre
'Pendiente'
```

Constructor con argumentos

```
class Persona:
    def __init__(self, nombre, apellidos="Sin apellidos"):
        self.nombre = nombre
        self.apellidos = apellidos
        print("Persona inicializada")

    def saludar(self):
        print("Hola, {0} {1}".format(self.nombre, self.apellidos))
```

```
>>> p1 = Persona('Pedro')
Persona inicializada
>>> p2 = Persona('Juan', 'Pérez')
Persona inicializada
>>> p1.saludar()
Hola, Pedro Sin apellidos
>>> p2.saludar()
Hola, Juan Pérez
```

Variables de clase y de instancia

- De clase: compartida por todas las instancias
- De instancia: única para cada instancia

```
class Coche:
    ruedas = 4          # Variable de clase

    def __init__(self, marca):
        self.marca = marca   # Variable de instancia
```

```
>>> honda = Coche("Honda")
>>> honda.ruedas
4
>>> Coche.ruedas
4
>>> honda.marca
'Honda'
>>> Coche.marca
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Coche' has no attribute 'marca'
```

Variables y métodos "privados"

```
class Empleado:
    estado = 'Ocupado'  # Pública
    __empresa = None    # Privada
    __historial = []    # Privada

    def set_empresa(self, empresa):   # Público
        self.__empresa = empresa
        self.__actualizar_historial()

    def __actualizar_historial(self): # Privado
        self.__historial.append(self.__empresa)
```

Herencia (I)

```
class Persona: # Igual que class Persona(object)
    pass

class Empleado(Persona):
    pass

class Multifuncion(Impresora, Escaner):
    pass
```

Herencia (II)

- Todas las clases heredan, en última instancia, de `object`
- Las subclases heredan atributos y métodos de clases ancestro (superclases)
- Herencia múltiple: se puede heredar de varias clases
  - MRO (Method resolution order)

Herencia (III)

```
>>> e = Empleado()
>>> isinstance(e, Empleado)
True
>>> isinstance(e, Persona)
True
>>> issubclass(Empleado, Persona)
True
>>> mf = Multifuncion()
>>> isinstance(mf, Multifuncion)
True
>>> isinstance(mf, Impresora)
True
>>> isinstance(mf, Persona)
False
```

Herencia (IV)

Overriding de métodos

```
class Transformador:
    def transformar(self, texto):
        return texto


class Mayusculas(Transformador):
    def transformar(self, texto):
        return texto.upper()
```

Herencia (V)

- super() permite llamar a métodos definidos en las clases ancestro

```
class Mamifero:
    def __init__(self):
        print("Soy un mamífero")
    def comer(self, comida):
        print("Como {0}".format(comida))

class Raton(Mamifero):
    def __init__(self):
        super().__init__()
        print("Soy un ratón")
    def comer(self, comida):
        super().comer(comida)
        print("y lo royo") # o roo o roigo
```

Herencia (VI)

```
class A:
    def ping(self):
        print("Ping - A")


class B:
    def ping(self):
        print("Ping - B")


class BA(B, A):
    def ping(self):
        super().ping() # Llama a B.ping
        A.ping(self)   # Llama a A.ping
```

Herencia (VII)

Linearization

```
class A:
    def ping(self):
        print("Ping - A")

class B(A):
    def ping(self):
        print("Ping - B")
        super().ping()

class C(A):
    def ping(self):
        print("Ping - C")
        super().ping()

class D(B, C):
    def ping(self):
        print("Ping - D")
        super().ping()
```

Ejercicios

1. Definir una jerarquía de clases para alimentos, partiendo de la clase `Alimento`.
2. Escribir una clase con un método para obtener el siguiente número de Fibonacci, habiéndole pasado los dos primeros números como argumentos del constructor.

3. Definir una clase `Forma` que tenga atributos de posición (`x` e `y`) y un método `descripcion()` que escribirá en pantalla la información de la forma.
   - Crear las subclases `Rectangulo` y `Circulo`, que aceptarán argumentos de su tamaño en el constructor, y reimplementarán `descripcion()`.
   - Crear un programa que genere varias formas y muestre sus descripciones.