UNL VIRTUAL

Tecnicatura Universitaria **en Software Libre**

Introducción al Desarrollo de Software

Unidad Nº 5

Introducción a la programación orientada a objetos

Autor
Emiliano López

Colaborador
Maximiliano Boscovich

FICH UNL

Tecnicatura Universitaria en Software Libre

Introducción al Desarrollo de Software

Docente: Emiliano López
Tutor: Maximiliano Boscovich

Contenidos

Introducción al Desarrollo de Software - Unidad 5

Este documento fue generado el 2015-12-01 13:00

LICENCIA CC BY-SA 4.0

1 Unidad 5: Introducción a la Programación Orientada a Objetos

Python es un lenguaje de Programación Orientado a Objetos (POO), lo que significa que puede manipular construcciones llamadas objetos. Se puede pensar en un objeto como una única estructura que contiene tanto datos como funciones, solo que las funciones en este contexto son llamadas *métodos*. En definitiva, los objetos son una manera de organizar datos y de relacionarlos con el código apropiado para manejarlo.

La Programación Orientada a Objetos introduce terminología, y una gran parte es simplemente darle un nuevo nombre a cosas que ya estuvimos usando.

Si bien Python nos provee un gran número de tipos ya definidos (int, float, str, dict, list, etc.), en muchas situaciones resultarán insuficientes, por lo que será necesario crear nuestros propios tipos, que almacenen la información relevante para el problema a resolver y contengan las funciones para operar con esa información.

Supongamos un programa que gestiona jugadores de fútbol de un club, independientemente de los detalles de implementación, contar con un tipo de dato *jugador* que permita cargar los datos personales y profesionales nos brinda la posibilidad de tener un código mas legible y organizado.

Por ejemplo, para cargar los datos de un nuevo jugador el código podría ser del siguiente modo:

```
pipa = Jugador('Lucas Alario', '8-10-1992', 'Delantero')
pipa.AgregarClub('Colon')
pipa.AgregarClub('River')
print("Club Actual: ", pipa.ClubActual())
```

Del fragmento de código previo podemos destacar:

- `pipa = Jugador(...)` crea una nueva instancia de la clase `Jugador` y le asigna este objeto al identificador `pipa`.
- La nueva estructura contiene un conjunto de datos denominados atributos o propiedades (nombre, fecha de nacimiento y posición) y un conjunto de funciones asociadas al objeto denominados métodos (`AgregarClub()`, `ClubActual()`)

1.1 Atributos y métodos

Veamos el modo de declarar este nuevo tipo `Jugador` con sus atributos y métodos.

```
class Jugador(object):
    """Clase Jugador"""
    def __init__(self, nombre=None, fechaNac=None, posicion=None):
        self.nombre = nombre
        self.fechaNac = fechaNac
        self.posicion = posicion
        self.clubes = []

    def setNuevoClub(self, club):
        '''agrega club a la lista de clubes'''
        self.clubes.append(club)

    def getClubActual(self):
        '''retorna último club'''
        return self.clubes[-1]
```

```
pipa = Jugador('Lucas Alario', '08-10-1992', 'Delantero')

pipa.setNuevoClub('Colon')
pipa.setNuevoClub('River')
print("Club Actual: ", pipa.getClubActual())

d10s = Jugador('EL Diego', '30-10-1960', 'Enganche')
```

```
Club Actual:  River
```

La clase anterior define la estructura de aquellos objetos que sean de tipo `Jugador()`. De los tres métodos que se observan, hay uno que merece especial atención:

- `__init__`: este método se denomina constructor, ya que está directamente asociado a la declaración e inicialización de un objeto. Esto es, en la el fragmento de código `pipa = Jugador('Lucas Alario', '8-10-1992', 'Delantero')` se lo invoca implícitamente (automáticamente).

  Los argumentos se corresponden con `nombre`, `fechaNac` y `posicion`. El primer argumento, `self`, hace referencia al mismo objeto y es utilizado para definir sus atributos dentro del constructor.

Los métodos restantes son funciones asociadas al objeto, :

- `setNuevoClub()`: agrega un club donde jugó
- `getClubActual()`: retorna el último club

Los datos relativos al club se cargan en una lista almacenada en el atributo `clubes`. El uso de métodos para modificar atributos es denominado **encapsulamiento**.

Es común encontrar métodos cuyos nombres empiecen con la palabra *set*, en aquellos casos donde se realizan modificaciones sobre los atributos del objeto, y métodos cuyos nombres comienzan con la palabra *get* para retornan propiedades de los objetos.

Si bien es una convención opcional es recomendable llamarlos de este modo, al igual que respetar el encapsulamiento (esto es, modificar y obtener los datos, mediante el uso de un método propio del objeto).

1.2 Métodos especiales

Así como el constructor `__init__`, existen otros métodos especiales que, si están definidos en nuestra clase, Python los llamará por nosotros cuando se lo utilice en determinadas situaciones. Veamos algunos.

1.2.1 Impresión

Si está definido el método `__str__` dentro de la clase, entonces será invocado automáticamente cada vez que se utilice la función `print()` con el objeto como argumento. Veamos la implementación:

```
def __str__(self):
    salida = self.nombre
    salida += '\n' + '='*len(self.nombre) + '\n'
    salida += 'Club: ' + self.getClubActual() + '\n'
    salida += 'Posición: ' + self.posicion + '\n'
    return salida
```

Luego, al imprimir directamente el objeto en pantalla obtendremos lo siguiente:

```
print(pipa)
```

```
Lucas Alario
============
Club: River
Posición: Delantero
```

Esto es equivalente a invocar directamente el método especial del siguiente modo:

```
pipa.__str__()
```

1.2.2 Comparación

Para resolver las comparaciones entre jugadores, será necesario definir algunos métodos especiales que permiten comparar objetos. En particular, cuando se quiere que los objetos puedan ser ordenados, los métodos que se debe definir son:

- `__lt__` menor que,
- `__le__` menor o igual,
- `__eq__` igual,
- `__ne__` distinto,
- `__gt__` mayor que,
- `__ge__` mayor o igual

Para dos objetos x, y:

- `x < y` llama a `x.__lt__(y)`,
- `x <= y` llama a `x.__le__(y)`,
- `x == y` llama a `x.__eq__(y)`,
- `x != y` llama a `x.__ne__(y)`,
- `x > y` llama a `x.__gt__(y)`,
- `x >= y` llama a `x.__ge__(y)`.

Para el ejemplo que estamos desarrollando, solamente programaremos el método `__lt__`, ya que al no ser un jugador menor que otro, nos retorna el complemento.

En la comparación formatearemos la fecha en el formato *aaaammmdd* ya que al convertirla a un entero podremos comprarla como un simple número, donde uno mas grande significa que el jugador es mas joven y, mas adulto, en caso contrario.

La implementación sería:

```
def __lt__(self, otro):
    '''si self es menor a otro'''
    dd1, mm1, aaaa1 = self.fechaNac.split('-')
    aaaammdd1 = aaaa1 + mm1 + dd1

    dd2, mm2, aaaa2 = otro.fechaNac.split('-')
    aaaammdd2 = aaaa2 + mm2 + dd2

    return (int(aaaammdd1) > int(aaaammdd2))
```

Luego, lo usamos:

```
d10s = Jugador('El Diego', '30-10-1960', 'Enganche')

print(pipa > d10s)
```

1.2.3 Algebraicos

Existen métodos especiales para todos los operadores matemáticos, de modo que al operar dos objetos, por ejemplo sumarlos, se invoca al método específico y se realiza la operación. Esto es también denominado sobrecarga de operadores, ya que se le asigna una función específica a un operador cuando es utilizado con objetos como operandos.

Para el ejemplo visto usaremos el monto del pase, así que se debe agregar el atributo *valor* a la clase e incorporar el método especial `__add__` de modo que al sumar objetos de tipo `Jugador()` se sumen estos campos.

```
def __add__(self, otro):
    return self.valor + otro.valor
```

Si ahora sumamos dos jugadores, obtendremos la suma de sus valores.

```
# otro jugador
higuain = Jugador('Gonzalo Higuaín', '10-12-1987', 'Desconocido')

# asignamos valor a cada jugador
pipa.valor = 1130000
d10s.valor = 9000000
higuain.valor = 1.20

# sumamos los jugadores
valor_equipo = pipa + d10s + higuain
print(valor_equipo)
```

Del mismo modo se implementan los métodos especiales para los siguientes operadores binarios

```
Operador            Método

 +          __add__(self, other)
 -          __sub__(self, other)
 *          __mul__(self, other)
 //         __floordiv__(self, other)
 /          __div__(self, other)
 %          __mod__(self, other)
 **         __pow__(self, other[, modulo])
 <<         __lshift__(self, other)
 >>         __rshift__(self, other)
 &          __and__(self, other)
 ^          __xor__(self, other)
 |          __or__(self, other)
```

Existen muchos otros métodos especiales como los de asignaciones extendidas y operadores unarios.

1.3 Herencia y polimorfismo

La herencia es un mecanismo de la programación orientada a objetos que sirve para crear clases nuevas a partir de otras preexistentes. Se heredan atributos y comportamientos y, partir de ella se crea una clase derivada con sus particularidades.

Por ejemplo, a partir de una clase `Jugador` podemos construir la clase `Capitan` que extiende a `Jugador` y agrega como atributo una lista de fechas de partidos que tuvo ese rol.

Se puede ver como un caso particular de la clase jugador, dado que tendrá los mismos atributos y métodos que un objeto de la clase `Jugador`, y a su vez tendrá algunos atributos y/o métodos extras.

El nombre de la clase base va entre los paréntesis de la definición de la nueva clase. Veamos el modo de implementarla:

```
class Capitan(Jugador):
    "Clase que representa al capitan."

    def __init__(self, nombre=None, fechaNac=None, posicion=None, capitan=[]):
        "Constructor de Capitan"
        # llama al constructor de Jugador
        Jugador.__init__(self, nombre, fechaNac, posicion)
        # nuevo atributo
        self.capitan = capitan

    def setCapitania(self, fecha):
        self.capitan.append(fecha)
```

En la implementación del método constructor (`__init__`) de `Capitan` se invoca al constructor de `Jugador`, luego, se agrega el atributo `capitan` y un método nuevo, `setCapitania`, que solamente existe en esta clase.

El hecho de heredar todas las características de la clase base hace que su uso sea prácticamente el mismo:

```
pulga = Capitan('Lionel Messi', '24-06-1987', 'Enganche')
pulga.setNuevoClub('Barcelona')
pulga.setCapitania('26-07-2008')
print(pulga)
```

Diferenciemos ahora el método de impresión, de modo que al imprimir en pantalla un jugador de tipo `Capitan`, muestre la última fecha de su capitanía. Para esto se debe modificar un método heredado, esta cualidad se denomina **Polimorfismo**. Veamos su implementación:

```
def __str__(self):
    salida = Jugador.__str__(self)
    salida += 'Última capitanía: ' + self.capitan[-1] + '\n'
    return salida
```

En la implementación del método `__str__` se invoca al de la clase base, y se agrega una línea más referida a la capitanía.

El presente capítulo ha sido una introducción a la POO presentada en forma de tutorial, a continuación se expone el código completo de lo desarrollado durante la unidad.

```
class Jugador(object):
    """Clase Jugador"""
    def __init__(self, nombre=None, fechaNac=None, posicion=None, \
        clubes=[], valor=None):
        self.nombre = nombre
        self.fechaNac = fechaNac
        self.posicion = posicion
        self.clubes = clubes
        self.valor = valor

    def setNuevoClub(self, club):
        '''agrega club a la lista de clubes'''
        self.clubes.append(club)

    def getClubActual(self):
        '''retorna último club'''
        return self.clubes[-1]

    def __str__(self):
        salida = self.nombre
        salida += '\n' + '='*len(self.nombre) + '\n'
        salida += 'Club: ' + self.getClubActual() + '\n'
        salida += 'Posición: ' + self.posicion + '\n'
        return salida

    def __lt__(self, otro):
        '''si self es menor a otro'''
        dd1, mm1, aaaa1 = self.fechaNac.split('-')
        aaaammdd1 = aaaa1 + mm1 + dd1

        dd2, mm2, aaaa2 = otro.fechaNac.split('-')
        aaaammdd2 = aaaa2 + mm2 + dd2

        return (int(aaaammdd1) > int(aaaammdd2))

    def __add__(self, otro):
        return self.valor + otro.valor

class Capitan(Jugador):
    "Clase que representa al capitan."
    def __init__(self, nombre=None, fechaNac=None, posicion=None, capitan=[]):
        "Constructor de Capitan"
        # llama al constructor de Jugador
        Jugador.__init__(self, nombre, fechaNac, posicion)
        # nuevo atributo
        self.capitan = capitan
    def setCapitania(self, fecha):
        self.capitan.append(fecha)

    def __str__(self):
        '''sobreescribe la clase heredada'''
        salida = Jugador.__str__(self)
        salida += 'Última capitanía: ' + self.capitan[-1] + '\n'
        return salida
```

```
pipa = Jugador('Lucas Alario', '08-10-1992', 'Delantero')
pipa.setNuevoClub('Colon')
pipa.setNuevoClub('River')
print(pipa)

d10s = Jugador('El Diego', '30-10-1960', 'Enganche')
d10s.setNuevoClub('Argentino Jr.')
d10s.setNuevoClub('Boca')
d10s.setNuevoClub('Barcelona')
d10s.setNuevoClub('Nápoles')
d10s.setNuevoClub('Sevilla')
d10s.setNuevoClub("Newell's")
d10s.setNuevoClub("Boca")
print(d10s)

pipa.valor = 1130000
d10s.valor = 9000000
monto = pipa + d10s

pulga = Capitan('Lionel Messi', '24-06-1987', 'Enganche')
pulga.setNuevoClub('Barcelona')
pulga.setCapitania('28-03-1981')
print(pulga)
```

```
Lucas Alario
============
Club: River
Posición: Delantero

El Diego
========
Club: Boca
Posición: Enganche

Lionel Messi
============
Club: Barcelona
Posición: Enganche
Última capitanía: 28-03-1981
```

Se recomienda profundizar este tema en el capítulo *Un primer vistazo a las clases (pag. 61)* del Tutorial de Python.