Actividad 03 - Orientación a objetos en Python

Paradigmas de la Programación

Carlos Augusto Zayas Guggiari

14 de marzo de 2024

Resumen

El alumno conocerá los principales conceptos relacionados con el paradigma de programación orientada a objetos y su implementación en el lenguaje Python. Como ejercicio, convertirá un programa estructurado/modular escrito anteriormente en uno orientado a objetos.

Contenido

1 Objetivos

- Clases y objetos en Python.
- Atributos y métodos en Python.
- Métodos especiales.

2 El paradigma orientado a objetos

La programación orientada a objetos (POO) es un paradigma de programación que permite representar un concepto del mundo real en una estructura denominada **clase**, que integra a las **propiedades** que describen al concepto, y a los **métodos** que son los procedimientos asociados a dichas propiedades.

Los objetos se crean (instancian) a partir de las clases, y pueden interactuar entre sí mediante mensajes. Se puede considerar a un objeto como compuesto de sustantivos (variables) y verbos (funciones).

En la actualidad, la orientación a objetos constituye el paradigma más importante de la industria del desarrollo de software. Smalltalk, C++, Objective-C, C#, Java, Javascript y Python son sólo algunos ejemplos de lenguajes de programación disponibles que implementan la orientación a objetos.

Un programa orientado a objetos puede verse como una colección de objetos que interactúan entre sí, en oposición a la visión del modelo convencional, en la que se ve a un programa como una lista de tareas (subrutinas) para llevar a cabo.

```
class MiClase:                          # Definición de una clase
    def MiMetodo(self):                 # Definición de un método ('self' es el objeto)
        self.MiPropiedad = "Mi Dato"    # Definición de una propiedad de objeto

MiObjeto = MiClase()                    # Instanciación de una clase
MiObjeto.MiMetodo()                     # Ejecución de un método
print(MiObjeto.MiPropiedad)             # Obtención de la propiedad de un objeto
```

Cada objeto es capaz de recibir mensajes, procesar datos, y enviar mensajes a otros objetos, como si se tratara de una "máquina" o **caja negra**, independiente, con un papel distinto o una responsabilidad única. Tanto los datos (llamados propiedades) como las acciones sobre esos datos (llamadas métodos) están estrechamente asociados con el objeto correspondiente.

2.1 Clase

Una clase es un modelo o plantilla que describe un tipo de objeto. El intérprete o compilador utiliza la clase cada vez que se tenga que crear un nuevo objeto. Este proceso se llama **instanciación**. Cada objeto es una instancia de alguna clase.

2.2 Objeto (Instancia)

Un objeto es una entidad de programación que contiene propiedades y métodos, de acuerdo a un modelo definido al que se denomina clase. Cada objeto tiene una **identidad** (nombre o identificador), un **estado** (datos o propiedades) y un **comportamiento** (acciones o métodos).

2.3 Propiedades

Las propiedades pueden ser de clase o de instancia:

```
class OtraClase:
    PropiedadClase = 0
    def __init__(self, n):          # Método constructor de la clase
        self.PropiedadObjeto = n    # Propiedad de instancia

Objeto1 = OtraClase(1)              # Instancia 1: Inicializa la propiedad de Objeto1
Objeto2 = OtraClase(2)              # Instancia 2: Inicializa la propiedad de Objeto2
```

```
print(Objeto1.PropiedadClase)     # Imprime: 0
print(Objeto1.PropiedadObjeto)    # Imprime: 1
print(Objeto2.PropiedadClase)     # Imprime: 0
print(Objeto2.PropiedadObjeto)    # Imprime: 2

OtraClase.PropiedadClase = 3      # Modifica la propiedad de la clase

print(Objeto1.PropiedadClase)     # Imprime: 3
print(Objeto2.PropiedadClase)     # Imprime: 3
```

- **de Clase**: Una propiedad de clase es aquella cuyo estado es compartido por todas las instancias de la clase.
- **de Instancia**: Una propiedad de instancia (o de objeto) es aquella cuyo estado puede ser invocado o modificado únicamente por la instancia a la que pertenece.

2.4 Métodos

Un método es una función, procedimiento o subrutina que forma parte de la definición de una clase. Los métodos, mediante sus parámetros de entrada, definen el comportamiento de las instancias de la clase asociada.

Los métodos son importantes para diseñar clases especializadas o "customizadas", sobre todo aquellos conocidos como métodos **especiales**, también llamados **mágicos** o **dunder** (de *double underscore*).

2.4.1 Métodos especiales

El término **dunder** (*double underscore*) se refiere a una convención particular de denominación que Python utiliza para nombrar a ciertos atributos (propiedades y métodos) predefinidos en el lenguaje. La convención consiste en usar el **doble subrayado** como prefijo y sufijo del nombre en cuestión, como en `__propiedad__` y `__metodo__()`.

Un método especial/mágico/dunder es aquel que Python llama implícitamente para ejecutar una determinada operación en un dato, como por ejemplo la suma. Estos métodos tienen nombres que comienzan y terminan con guiones bajos dobles.

```
>>> 5 + 2
7
>>> (5).__add__(2)
7
>>> _
```

2.4.2 El parámetro "self"

La palabra *self* representa a la instancia de una clase, es decir, a un objeto. Se utiliza dentro de la clase para hacer referencia a las propiedades del objeto en sí y diferenciarlas de las de los métodos y la propia clase.

```
class Persona:

    cantidad = 0

    def __init__(self, nombre):
        self.nombre = nombre
        Persona.cantidad += 1

    def saludar(self):
        print("Hola, mi nombre es", self.nombre)
        print("Somos", Persona.cantidad)

    def __del__(self):
        print("{} dice adiós.".format(self.nombre))
```

```
sobrino_donald = dict()
for nombre in ["Hugo", "Paco", "Luis"]:
    sobrino_donald[nombre] = Persona(nombre)
    sobrino_donald[nombre].saludar()
```

En el ejemplo, `nombre` es un atributo o parámetro del método `__init__`, mientras que `self.nombre` es una propiedad de instancia y `cantidad` es una propiedad de clase.

Python llama al método `.__init__()` cada vez que se genera una instancia de una clase determinada. El objetivo de este método especial es inicializar cualquier atributo de instancia que tenga en su clase, es decir, aquellos accesibles mediante `self` dentro de la clase.

2.4.3 Sobrecarga de operadores

Consiste en la posibilidad de contar con métodos del mismo nombre pero con comportamientos diferentes.

```
class Lista:
    def __init__(self):
        self.items = []
    def __str__(self):
        return '\n'.join(self.items)
    def __repr__(self):
        return repr(self.items)
    def agregar(self, item):
        self.items.append(str(item))
    def vaciar(self):
        self.items = []
    def vacio(self):
        return self.items == []
    def cantidad(self):
        return len(self.items)

class Archivo:
    def __init__(self, nombre='lista.txt'):
        self.items = open(nombre).readlines() if os.path.exists(nombre) else []
        self.items = [ item.strip() for item in self.items ] # Quita espacios
        ↪ sobrantes
        self.nombre = nombre
    def __del__(self):
        open(self.nombre,'w').write('\n'.join(self.items))

class ListArch(Archivo, Lista):
    pass

lista = ListArch()
lista.vaciar()
lista.agregar('Hola')
lista.agregar(2)
lista.agregar((3, 4))
lista.agregar('Chau')

print("Cantidad:", lista.cantidad(), '\n')
print(lista)
print('\nLista:',repr(lista))
```

En el código anterior, `__init__`, `__repr__`, `__str__` y `__del__` son ejemplos de **redefiniciones de métodos predefinidos** que modifican parte del comportamiento estándar de los objetos, en este caso la construcción y la impresión del objeto, respectivamente. A esta capacidad del lenguaje se le denomina **sobrecarga**.

- `__repr__` devuelve una representación del objeto en forma de cadena de caracteres, que puede volver a

generar el objeto si es evaluada nuevamente, por ejemplo con la función `eval()`.
- `__str__` devuelve una cadena de caracteres cuando se imprime el objeto con la sentencia `print()`.

2.4.4 Constructor y Destructor

En Python, los métodos `__init__` y `__del__` son lo que se conoce como método **constructor** y **destructor** respectivamente. No es obligatorio incluirlos cuando definimos una clase, pero como se verá a continuación, pueden ser de mucha utilidad, y a menudo son necesarios.

- El método **constructor** de una clase se ejecuta en cada instanciación de la clase, es decir, durante el proceso de creación de un objeto.
- El método **destructor** de una clase se ejecuta cuando el objeto ya no es referenciado dentro del programa, o simplemente cuando el proceso termina.

Recordemos que un proceso es un programa en ejecución, y durante su finalización, se borran de la memoria todos los objetos creados, lo que activa el método destructor correspondiente a cada uno de ellos.

2.4.5 Getter / Setter / Deleter

Las propiedades, tanto las de clase como las de instancia, se pueden acceder directamente utilizando la sintaxis "punto" (objeto.propiedad). Sin embargo, se recomienda que cualquier operación relacionada con propiedades se realice mediante métodos especiales, denominados **getters** y **setters** (obtenedores y colocadores). Opcionalmente, también puede definirse un **deleter** (borrador).

Estos métodos especiales son necesarios principalmente para el manejo de las propiedades más importantes del objeto, generalmente aquellas que necesitan ser accedidas desde otros objetos, no precisamente desde los métodos propios del objeto.

No es obligatorio definir getters y setters para todas las propiedades, sino sólo para aquellas en las que necesitemos algún tipo de validación previa antes de obtener o colocar el valor involucrado.

En Python contamos con dos maneras de definir estos métodos especiales:

```
# Ejemplo 1: Función 'property'

class Ejemplo1:
    def __init__(self):
        self._x = 1
    def getx(self):
        return self._x
    def setx(self, valor):
        self._x = valor
    def delx(self):
        del self._x
    x = property(getx, setx, delx, "Soy la propiedad 'x'.")

# Ejemplo 2: Decoradores ('@...')

class Ejemplo2:
    def __init__(self):
        self._x = 2
    @property
    def x(self):
        "Soy la propiedad 'x'."
        return self._x
    @x.setter
    def x(self, valor):
        self._x = valor
    @x.deleter
    def x(self):
        del self._x
```

Ambas sintaxis cumplen el mismo propósito y dan exactamente los mismos resultados, pero la segunda es la más nueva incorporación al lenguaje Python.

También contamos con los métodos `__getitem__` y `__setitem__` para manipular listas y/o diccionarios usando únicamente el nombre del objeto. De esta manera, en vez de escribir `objeto.lista[indice]` simplemente podemos escribir `objeto[indice]`.

```
class Dic:
    def __init__(self):
        self.dic = {}
    def __repr__(self):
        return repr(self.dic)
    def __getitem__(self, clave):
        return self.dic.get(clave, None)
    def __setitem__(self, clave, valor):
        self.dic[clave] = valor
```

De esta manera, una instancia de la clase `Dic` puede usarse como si se tratase de un diccionario:

```
d = Dic()
d['Nombre'] = 'Carlos'
print(d['Nombre'])
```

2.5 Conceptos Fundamentales de la POO

La programación orientada a objetos se basa en los siguientes conceptos fundamentales:

2.5.1 Encapsulamiento

Consiste en colocar al mismo nivel de abstracción a todos los elementos (estado y comportamiento) que pueden considerarse pertenecientes a una misma entidad (identidad). Esto permite aumentar la cohesión de los componentes del sistema.

```
class Persona:

    def __init__(self, nombre):
        self.setNombre(nombre)

    def getNombre(self):
        return ' '.join(self.__nombre)

    def setNombre(self, nombre):
        self.__nombre = nombre.split()

    nombre = property(getNombre, setNombre)
```

Las propiedades pertenecen al espacio de nombres del objeto (namespace) y pueden estar ocultas, es decir, sólo accesibles para el objeto. En Python, esto último se logra superficialmente anteponiendo dos guiones bajos (__) al nombre de la propiedad.

```
carlos = Persona("Carlos Zayas")
print(carlos.nombre.upper())

carlos.nombre = 'Carlos A. Zayas G.'
print(carlos.nombre.upper())
```

Decimos que en Python se logra sólo superficialmente el ocultamiento de las propiedades porque, si bien no podemos invocar la propiedad nombre de manera tradicional, sí podemos hacerlo de la siguiente manera:

```
print(carlos._Persona__nombre)
```

2.5.2 Herencia

Las clases se relacionan entre sí dentro de una jerarquía de clasificación que permite definir nuevas clases basadas en clases preexistentes, y así poder crear objetos especializados.

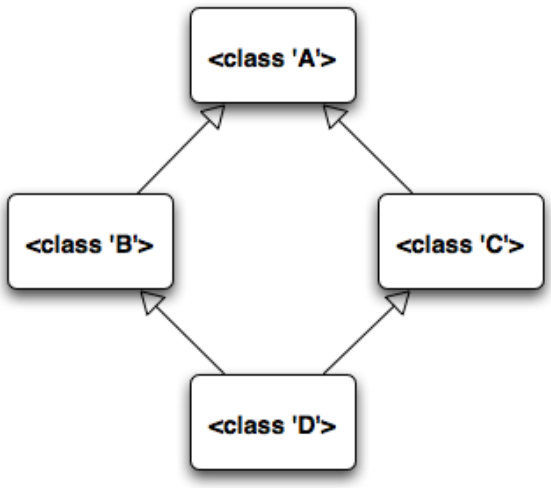
```
class Empleado(Persona):

    def __init__(self, nombre, cargo):
        Persona.__init__(self, nombre)
        self.cargo = cargo
        print("{} es {}".format(self.nombre, self.cargo))
```

La herencia es el mecanismo por excelencia de la programación orientada a objetos que permite lograr la reutilización de código. Mediante ella, podemos crear nuevas clases modificando clases ya existentes.

2.5.2.1 Herencia Múltiple Cuando la herencia involucra a más de una clase, hay herencia múltiple. El orden en el que las clases son invocadas determina la prevalencia de propiedades y métodos de idéntica denominación en el "árbol genealógico".

2.5.2.2 Orden de Resolución de Métodos El Orden de Resolución de Métodos o MRO (*Method Resolution Order*) es la manera en que un lenguaje de programación decide dónde buscar un método (o una propiedad) en una clase que hereda estos elementos de varias clases superiores.

La importancia del MRO se hace patente en presencia de la herencia múltiple donde, ante dos elementos con la misma denominación en clases distintas, es necesario definir qué método o propiedad prevalecerá en una instanciación.



Herencia múltiple en forma de diamante.

En el caso de Python, la evolución del lenguaje dio como resultado dos algoritmos MRO distintos, uno simple para las clases de estilo antiguo (versiones de Python anteriores a la 3) y otro más sofisticado para las clases de estilo nuevo (las de Python 3 y las que heredan de `object` en Python 2). Ambos algoritmos provienen de la teoría de grafos, y son los siguientes:

- **DFS – Depth First Search** (Búsqueda en profundidad): Recorre los nodos del grafo (árbol) de izquierda a derecha empezando de la raíz pero hasta el nodo más lejano. En el diagrama, empezando desde la clase D, el orden sería D, B, A, C.
- **BFS – Breadth First Search** (Búsqueda en anchura): Recorre los nodos del grafo efectuando barridos de izquierda a derecha. En el diagrama, empezando desde la clase D, el orden sería D, B, C, A.

```
# Este ejemplo es para Python 2

# Algoritmo: DFS - Depth First Search (Busqueda en profundidad)
```

```
class A: x = 'a'
class B(A): pass
class C(A): x = 'c'
class D(B, C): pass
print 'Viejo estilo: D.x = "%s"' % D.x

# Algoritmo: BFS - Breadth First Search (Busqueda en anchura)
class A(object): x = 'a'
class B(A): pass
class C(A): x = 'c'
class D(B, C): pass
print 'Nuevo estilo: D.x = "%s"' % D.x
```

Este es el resultado al ejecutar el programa en Python 2:

```
Viejo estilo: D.x = "a"
Nuevo estilo: D.x = "c"
```

El MRO puede obtenerse mediante mecanismos de **introspección**, como puede verse en la siguiente secuencia de sentencias ejecutadas durante una sesión con el intérprete interactivo de Python.

Empecemos creando dos clases vacías, únicamente a modo de ejemplo, haciendo uso de la sentencia comodín pass:

```
>>> class Animal: pass
...
>>> class Perro(Animal): pass
...
```

Acabamos de definir dos clases: `Animal`, y `Perro`, que hereda los métodos y propiedades de `Animal`. Ambas clases aparentemente son iguales, a ninguna de las dos se les definieron métodos o propiedades específicas, pero se diferencian en cuanto a la herencia – una deriva de la otra.

A continuación, creamos una instancia de la clase `Perro`, a la que llamamos `fido`.

```
>>> fido = Perro()
```

Mediante el método `__class__` y el operador `is` podemos ver que la clase del objeto `fido` es `Perro`, no `Animal`, aunque "desciende" de ella.

```
>>> fido.__class__ is animal
False
>>> fido.__class__ is perro
True
```

Sin embargo, con la función `isinstance`, vemos que `fido` es una instancia de la clase `Animal`, al igual que de la sub-clase `Perro`.

```
>>> isinstance(fido, Animal)
True
>>> isinstance(fido, Perro)
True
```

Por último, el método `__mro__` nos devuelve una **tupla** en la que podemos ver el orden de resolución de métodos:

```
>>> Animal.__mro__
(<class '__main__.Animal'>, <class 'object'>)
>>> Perro.__mro__
(<class '__main__.Perro'>, <class '__main__.Animal'>, <class 'object'>)
```

Los métodos y propiedades de una clase se superpondrán a los de la clase superior en el orden que aparecen en la tupla.

2.5.3 Polimorfismo

Consiste en definir comportamientos diferentes basados en una misma denominación, pero asociados a objetos distintos entre sí. Al llamarlos por ese nombre común, se utilizará el adecuado al objeto que se esté invocando.

```
a = Persona("Pablo")                # Nace Pablo
b = Persona("Juan")                 # Nace Juan
print(b)                            # Juan dice "Hola"
c = Empleado("Esteban", "Chofer")   # Nace Esteban
                                    # Esteban es Chofer
```

En el siguiente ejemplo, las dos clases derivadas de Animal comparten el método sonido, pero cada una le agrega su particularidad.

```
class Animal:

    cantidad = 0

    def __init__(self):
        print("Hola, soy un animal.")
        self.nombre = ""
        Animal.cantidad += 1
        print("Hay", Animal.cantidad, "animales.")

    def sonido(self):
        print("Este es mi sonido:")

    def __del__(self):
        print(self.nombre, "dice: Adios!")

class Perro(Animal):

    def __init__(self, nombre):
        Animal.__init__(self)
        print("Soy un perro.")
        self.nombre = nombre
        print("Me llamo", self.nombre)

    def sonido(self):
        Animal.sonido(self)
        print("Guau!")

class Gato(Animal):

    def __init__(self, nombre):
        Animal.__init__(self)
        print("Soy un gato.")
        self.nombre = nombre
        print("Me llamo", self.nombre)

    def sonido(self):
        Animal.sonido(self)
        print("Miau!")

fido = Perro("Fido")
fido.sonido()
```

```
tom = Gato("Tom")
tom.sonido()
```

Los objetos `fido` y `tom` descienden de `Animal` pero cada uno "emite su propio sonido".

Una subclase suele necesitar llamar al constructor de la superclase:

```
class Subclase(Superclase):
    def __init__(self):
        # Aquí la subclase hace sus cosas
        Superclase.__init__(self)
        # Aquí la subclase sigue haciendo sus cosas
```

Para no tener que nombrar a la superclase, puede usarse la función `super`:

```
super(Subclase, self).__init__()
```

En Python 3, es posible ahorrarse un poco de código escribiendo simplemente:

```
super().__init__()
```

Ejemplo en ambas versiones:

```
# Python 2.x

class A(object):
    def __init__(self):
        print "Mundo"

class B(A):
    def __init__(self):
        print "Hola"
        super(B, self).__init__()

# Python 3.x

class A:
    def __init__(self):
        print("Mundo")

class B(A):
    def __init__(self):
        print("Hola")
        super().__init__()
```

Al crear una instancia de la clase `B`:

```
b = B()
```

El resultado es:

```
Hola
Mundo
```

3 Ejercicio

1. Reescriba el programa de la Actividad 01 (“Adivina un número”) implementando lo que aprendió acerca del paradigma orientado a objetos, agregando opciones parametrizables al juego, por ejemplo cantidad límite de intentos y número límite del rango de números dentro del cual está el elegido al azar por el programa.
2. Cree un módulo que contenga toda la funcionalidad que usted considere pueda ser reutilizable en otros proyectos futuros. Por ejemplo, una función que elija un número al azar dentro de un rango. El programa principal deberá importar y usar este módulo.
3. Comprima los archivos de código fuente en un archivo ZIP llamado ‘actividad03.zip’ y envíelo por correo desde su cuenta FP-UNA a czayas@pol.una.py, poniendo como asunto “Actividad 03”.

```python
#!/usr/bin/env python3
'''
Ejemplo de manejo de parámetros recibidos desde la línea de comandos.
'''

import sys

parametros = sys.argv  # Lista de parámetros (argument values)

programa = parametros.pop(0)  # El primer parámetro es el nombre del programa.

print("Hola, soy el programa", programa)
print("Recibí {} parámetros.".format(len(parametros)))
print("Son los siguientes:")
for parametro in parametros:
    print(parametro)
```

NOTA:
Utilice un **analizador de código** durante la elaboración de los ejercicios. Es una herramienta que analiza el código fuente sin ejecutarlo, con el fin de identificar prácticas de codificación deficientes. Ofrece retroalimentación a los programadores durante la fase de desarrollo, incluso sobre fallas de seguridad que podrían introducirse en el código.
Hay entornos de desarrollo que integran el análisis de código al editor de programación, funcionando de manera similar a las facilidades de autocompletado y resaltado de sintaxis. Uno de los más conocidos para Python es PyCharm que es un ambiente sumamente completo y profesional, pero existen alternativas más sencillas que consumen menos recursos.

4 Recursos

Los siguientes recursos están disponibles online de manera gratuita:

4.1 Documentación

- El Tutorial de Python 3 - PDF
- Aprenda a Pensar como un Programador con Python - PDF
- Inmersión en Python 3 - PDF
- PEP 8: Guía de estilo para el código Python - PDF

4.2 Artículos

- El Pythonista: Guía de estilos en Python - Link
- El Pythonista: Entornos de desarrollo para Python - Link

4.3 Software

- Mu: Mini entorno de desarrollo para Python - Link
- Mu: Versión portable para Windows - Link
- Thonny: Entorno de desarrollo con herramientas didácticas - Link
- Thonny: Versión portable para Windows - Link

NOTA:
Los enlaces fueron verificados a la fecha de revisión de este documento. Sin embargo, esto no garantiza su validez al momento de leer esto. Favor enviar sugerencias y correcciones a czayas@pol.una.py