Clase 20: Objetos y Clases

Hemos visto dos paradigmas de programación:

- **paradigma funcional:** los problemas se modelan como funciones que toman datos de entrada y retornan un valor simple o compuesto que sólo depende de la entrada
- **paradigma imperativo:** en donde hay estructuras que actúan como memoria y por lo tanto los resultados retornados por las funciones no sólo dependen de los valores de los parámetros de la función, sino que también dependen del estado actual de estas estructuras.

Se estudiará un **tercer paradigma de programación**, conocido como **programación orientada al objeto**. En este paradigma de programación se utilizan dos conceptos fundamentales:

- **objetos:** Un `objeto` es un modelo computacional de un ente o concepto que posee ciertos atributos y con el cual podemos realizar ciertas operaciones.
- **clases:** Una `clase` permite describir en forma abstracta los atributos y operaciones del concepto modelado, que luego se instancia en un objeto.

Hemos visto datos complejos (`structs`) que nos permiten modelar los atributos del concepto, y definimos funciones sobre éstos para implementar las distintas operaciones posibles. En cambio, en la programación orientada a objetos, los **objetos contienen tanto los datos asociados a éstos como las operaciones que se pueden realizar con ellos**.

Un ejemplo: automóviles

Se necesita un programa para hacer simulación sobre tráfico de automóviles.
Por cada automóvil es necesario almacenar información como:

- color,
- velocidad actual,
- velocidad máxima,
- cantidad de gasolina en el estanque,
- etc.

Además, se requiere poder realizar las siguientes operaciones con los automóviles:

- acelerar,
- frenar,
- apagar,
- consultar cuánta gasolina queda en el estanque,
- cambiar su color, etc.

En programación orientada a objetos, es necesario definir una **clase `Automovil`** de la cual podamos crear objetos (**instancias especificas de automóviles**), y luego a través de interacciones con estos objetos programar la simulación.

Crear (instanciar) e interactuar con objetos

Supongamos que ya tenemos implementada la clase `Automovil`. Este es un ejemplo de como **instanciar** (crear) un objeto:

```
unAutomovil = Automovil()
```

Hemos creado un objeto de la clase `Automovil`.

Nuestro automóvil también puede moverse, acelerar, frenar, etc. Lo Los objetos tienen `comportamiento`, siguiendo nuestro modelo de la realidad. Supongamos que nuestros objetos de la clase `Automovil` tienen distintas funciones, o **`métodos`**, **que realizan acciones sobre el objeto**:

```
unAutomovil.encender()
unAutomovil.acelerar()
unAutomovil.frenar()
unAutomovil.apagar()
```

Los **métodos, al ser funciones, pueden recibir parámetros**. Un método indica qué parámetros recibe (en su contrato). Por ejemplo, nuestro automóvil puede esperar saber cuánto tiempo presionar el acelerador:

```
unAutomovil.acelerar(30) # presiona el acelerador durante 30 segundos
```

Estas llamadas a métodos del objeto son aplicaciones de funciones como las hemos visto siempre, salvo que su contexto está unido al **estado** de cada objeto.

En el ejemplo anterior, supongamos que nuestro automóvil entrega el nivel de gasolina **actual** tras un tiempo en movimiento:

```
gasolina = unAutomovil.obtenerNivelGasolina()
if gasolina > 0:
  unAutomovil.acelerar()
else:
  unAutomovil.frenar()
```

Los **objetos también son valores**, al mismo nivel que los enteros, strings, estructuras, etc. Por lo mismo, podemos incluso **pasar un objeto como parámetro** y **retornar** objetos. Por ejemplo,

nuestros métodos `acelerar` y `frenar` pueden retornar como resultado el **mismo** objeto, lo que resulta en un patrón particular como el que sigue a continuación:

```
unAutomovil.acelerar().acelerar().frenar().acelerar().frenar()
```

Múltiples instancias de una clase y estado de los objetos

Podemos **crear distintas instancias de una clase como objetos**:

```
unAutomovil = Automovil()
otroAutomovil = Automovil()
# a hacerlos competir!
```

¿En qué se diferencian dos objetos instanciados de la misma clase?

A simple vista no se ve mucha diferencia (ambos objetos pueden acelerar, frenar, encenderse, apagarse, etc.), pero lo que los caracteriza es lo que los diferencia. En lo que hemos visto hasta ahora, si el automóvil acelera durante 30 segundos, podemos esperar que su nivel de gasolina disminuya. Como dijimos al comienzo, si queremos representar un automóvil por su color, su velocidad, etc. estamos diferenciándolos. Pero también podemos cambiar estos valores (como la gasolina). Estos valores constituyen el **estado del objeto**. Tal como en las estructuras mutables, el estado de un objeto también puede cambiar. Veamos un par de ejemplos.

Si queremos crear un automóvil de cierto color y cierta velocidad máxima, podríamos hacerlo al momento de instanciar el objeto:

```
miAutomovil = Automovil(color="azul", velocidadMaxima=220)
```

Si queremos luego modificar estos valores (supongamos que `enchulamos` nuestro automóvil) también podemos hacerlo:

```
miAutomovil.setColor("rojo")
miAutomovil.setVelocidadMaxima(250)
```

Ejemplo: libreta de direcciones

Supongamos que tenemos dos clases: **`Registro`** y **`Libreta`**, con los cuales podemos crear objetos que representen registros, con nombre, teléfono y dirección; y una forma de crear distintas libretas con nombre:

```
libretaPersonal = Libreta("personal")
libretaTrabajo = Libreta("trabajo")
registro1 = Registro(nombre="Juan Gonzalez", \
  telefono="777-7777", direccion="Calle ABC 123")
libretaTrabajo.agregarRegistro(registro1)
```

También puede existir una función para buscar un registro dado un nombre:

```
john = libretaTrabajo.buscar("John")
john.setTelefono("133")
john.getTelefono()
"133"
```

Definición de Clases

Estudiaremos cómo definir una clase en Python, esto incluye definir:

- los **campos** de la clase
- la **construcción** de un objeto
- **métodos** en la clase

Clase

Para definir una clase en Python se utiliza la instrucción `class`, debe señalar:

- el **nombre que tendría la clase** (comienzan con mayúscula)
- los **campos (también llamados variables de instancia)** que tendría la clase y sus tipos

FraccionV1:

```
# Campos :
# numerador : int
# denominador : int
class FraccionV1:
```

Constructor

El **constructor** es el **primer método** que uno debe definir en una clase.

- Se ejecuta **cada vez** que se crea un nuevo objeto de la clase.

- usualmente, en este método es en donde se definen los campos de la clase y sus valores iniciales
- método constructor **siempre** tiene el nombre `__init__`
- sólo se puede definir **un constructor por clase**
- **todo método** de una clase en Python (incluyendo al constructor) tiene como primer parámetro la palabra clave `self`

```
    # Constructor
    def __init__(self, numerador = 0, denominador = 1):
        # Inicializacion de campos
        self.numerador = numerador
        self.denominador = denominador
```

```
f = FraccionV1(1, 2) # crea la fraccion 1/2
f = FraccionV1() # crea la fraccion 0/1
```

El constructor define e inicializa las dos variables de instacias:

- `self.numerador`
- `self.denominador`.

Note que es necesario anteponer `self.` cada vez que se desee accesar o modificar dichos campos (sino Python cree que se está refiriendo a variables locales del método).

Métodos

Se **definen igual que las funciones** en Python, con la diferencia que se definen dentro del contexto de una clase y deben tener como primer parámetro la referencia `self`.

El parámetro `self` no es parte del contrato del método:

```
    # suma: FraccionV1 -> FraccionV1
    # devuelve la suma de la fraccion con otra fraccion
    def suma(self, fraccion):
        num = self.numerador * fraccion.denominador + \
              fraccion.numerador * self.denominador
```

```
        den = self.denominador * fraccion.denominador
        return FraccionV1(num, den)
```

Note `self` corresponde al objeto que invoca al método `suma` y `fraccion` corresponde al objeto que se pasó por parámetro al método. Por ejemplo, en el siguiente código:

```
f1 = FraccionV1(1, 2)
f2 = FraccionV1(5, 6)
f3 = f1.suma(f2)
```

el objeto `f1` corresponde a `self` en el método `suma`, y el objeto `f2` corresponde al parámetro `fraccion` en dicho método.

Métodos accesores y mutadores

Los métodos de una clase se pueden dividir en dos categorías:

- accesores (**accesors**): sólo acceden al contenido de los campos del objeto, pero no los modifican. Ej: `suma`.
- mutadores (**mutators**): modifican (o pueden modificar) los valores de los campos de la clase.
    - igual que en las funciones con memoria que modifican el estado de una variable, debemos indicar el efecto que puede tener un método mutador.

Ejemplo método mutador

Suponiendo que disponemos de la función `mcd(x,y)` implementamos el método `simplificar`:

```
    # simplificar: None -> None
    # efecto: simplifica la fraccion, puede modificar los
    # valores de los campos numerador y denominador
    def simplificar(self):
        valor = mcd(self.numerador, self.denominador)
        if valor > 1:
            self.numerador = self.numerador / valor
            self.denominador = self.denominador / valor
```

Es común en una clase tener métodos accesores para obtener el valor de los distintos campos, y métodos mutadores para asignarles un nuevo valor. Por convención: los nombres de los métodos accesores comienzan con `get`, y los mutadores comienzan con `set`:

```
    # getNumerador: None -> int
    # devuelve el valor del campo numerador
    def getNumerador(self):
```

```
        return self.numerador

    # getDenominador: None -> int
    # devuelve el valor del campo denominador
    def getDenominador(self):
        return self.denominador

    # setNumerador: int -> None
    # efecto: modifica el valor del campo numerador
    def setNumerador(self, numerador):
        self.numerador = numerador

    # setDenominador: int -> None
    # efecto: modifica el valor del campo denominador
    def setDenominador(self, denominador):
        self.denominador = denominador
```

Receta de diseño de clases

Pasos:

1. **Antes de definir la clase** identificar campos que tendrá y sus tipos.
2. La definición de los **métodos sigue las reglas habituales de la receta de diseño**, pero los cuerpos de los métodos y los tests quedan pendientes.
3. Terminada la definición de métodos: **fuera de la clase se implementan los tests** para todos los métodos (ya que es necesario crear objetos de la clase con los cuales invocar los objetos, y dependiendo de los valores de los campos de cada objeto se puede determinar la respuesta).
4. Se **implementan los cuerpos de los métodos**, y luego se ejecutan los tests. Se corrigen los errores detectados en los tests, y se itera hasta que todos los tests sean exitosos.

```
# Campos :
# numerador : int
# denominador : int
class FraccionV1:

    # Constructor
    def __init__(self, numerador = 0, denominador = 1):
        # Inicializacion de campos
        self.numerador = numerador
        self.denominador = denominador
```

```
    # getNumerador: None -> int
    # devuelve el valor del campo numerador
    def getNumerador(self):
        return self.numerador

    # getDenominador: None -> int
    # devuelve el valor del campo denominador
    def getDenominador(self):
        return self.denominador

    # setNumerador: int -> None
    # efecto: modifica el valor del campo numerador
    def setNumerador(self, numerador):
        self.numerador = numerador

    # setDenominador: int -> None
    # efecto: modifica el valor del campo denominador
    def setDenominador(self, denominador):
        self.denominador = denominador

    # toString: None -> str
    # devuelve un string con la fraccion
    def toString(self):
        return str(self.numerador) + "/" + str(self.denominador)

    # suma: FraccionV1 -> FraccionV1
    # devuelve la suma de la fraccion con otra fraccion
    def suma(self, fraccion):
        num = self.numerador * fraccion.denominador + \
              fraccion.numerador * self.denominador
        den = self.denominador * fraccion.denominador
        return FraccionV1(num, den)

    # mcd: int int -> int
    # devuelve el maximo comun divisor entre dos numeros x e y
    # ejemplo: mcd(12, 8) devuelve 4
    global mcd
    def mcd(x, y):
        if x == y:
            return x
        elif x > y:
            return mcd(x-y, y)
        else:
            return mcd(x, y-x)

    # Test
    assert mcd(12, 8) == 4

    # simplificar: None -> None
    # efecto: simplifica la fraccion, puede modificar los
    # valores de los campos numerador y denominador
    def simplificar(self):
```

```
        valor = mcd(self.numerador, self.denominador)
        if valor > 1:
            self.numerador = self.numerador / valor
            self.denominador = self.denominador / valor

# Tests
f1 = FraccionV1(1, 2)
f2 = FraccionV1(5, 6)
# Test de accesors
assert f1.getNumerador() == 1
assert f2.getDenominador() == 6
# Test de mutators
f2.setNumerador(3)
f2.setDenominador(4)
assert f2.getNumerador() == 3 and f2.getDenominador() == 4
# Test de metodo suma
# El siguiente test es incorrecto
# assert f1.suma(f2) == FraccionV1(10, 8)
# El siguiente test es correcto
f3 = f1.suma(f2)
assert f3.getNumerador() == 10 and f3.getDenominador() == 8
# Test de metodo toString
assert f3.toString() == "10/8"
# Test de metodo simplificar
f3.simplificar()
assert f3.getNumerador() == 5 and f3.getDenominador() == 4
```

Otra versión sin mutadores, solo accesores

```
# Campos :
# numerador : int
# denominador : int
class FraccionV2:

    # Constructor
    def __init__(self, numerador = 0, denominador = 1):
        # Inicializacion de campos
        # campos invisibles al usuario
        self.__numerador = numerador
        self.__denominador = denominador

    # getNumerador: None -> int
    # devuelve el valor del campo numerador
    def getNumerador(self):
        return self.__numerador

    # getDenominador: None -> int
```

```
    # devuelve el valor del campo denominador
    def getDenominador(self):
        return self.__denominador

    # toString: None -> str
    # devuelve un string con la fraccion
    def toString(self):
        return str(self.__numerador) + "/" + str(self.__denominador)

    # suma: FraccionV2 -> FraccionV2
    # devuelve la suma de la fraccion con otra fraccion
    def suma(self, fraccion):
        num = self.__numerador * fraccion.__denominador + \
              fraccion.__numerador * self.__denominador
        den = self.__denominador * fraccion.__denominador
        return FraccionV2(num, den)

    # mcd: int int -> int
    # devuelve el maximo comun divisor entre dos numeros x e y
    # ejemplo: mcd(12, 8) devuelve 4
    global mcd
    def mcd(x, y):
        if x == y:
            return x
        elif x > y:
            return mcd(x-y, y)
        else:
            return mcd(x, y-x)

    # Test
    assert mcd(12, 8) == 4

    # simplificar: None -> FraccionV2
    # devuelve la fraccion simplificada
    def simplificar(self):
        valor = mcd(self.__numerador, self.__denominador)
        num = self.__numerador / valor
        den = self.__denominador / valor
        return FraccionV2(num, den)

# Tests
f1 = FraccionV2(1, 2)
f2 = FraccionV2(3, 4)
# Test de accesors
assert f1.getNumerador() == 1
assert f2.getDenominador() == 4
# Test de metodo suma
f3 = f1.suma(f2)
assert f3.getNumerador() == 10 and f3.getDenominador() == 8
# Test de metodo toString
assert f3.toString() == "10/8"
# Test de metodo simplificar
```

```
# Test de método simplificar
f4 = f3.simplificar()
assert f4.getNumerador() == 5 and f4.getDenominador() == 4
```