Clases

June 22, 2018

1 Resumen IIC-1103

Por: Nicolás Quiroz, Daniel Leal, Alfonso Irarrázaval.

---

2 Clases

Las clases son quizás una de las cosas más potentes de Python. La forma más fácil de describirlo es que son una forma de modelar cómo se comporta de manera **general** un **tipo** de objeto.

Ejemplos: Quiero modelar un condominio –> crear una clase `Casa` para modelar cómo se comportarán las casas del condominio. Ésto, sumado a una lista de casas, modela un condominio. Quiero modelar un bus escolar –> crear clase `Chofer` (Aún cuando es sólo un chofer por bus), crear una clase `Escolar`, y una clase `Bus`. `Bus` podría tener un atributo que sea una lista con los `escolares` abordo del bus y un atributo que guarde al chofer.

Ahora veremos ejemplos.

2.1 Instancia vs Clase

Algo que suele ser difícil de entender en programación es la diferencia entre una instancia de una clase, y la clase. Una analogía útil es pensar en el plano de una casa y en una casa. Una **Clase** es como un plano de una casa: define cómo se hace una casa, y que cosas tiene. Por otra parte, una **instancia** de una clase es un objeto del tipo `Casa`. Por lo tanto, es una Casa en sí, y puedo crear varias casas iguales a partir del mismo plano (la clase `Casa`).

Veamos los ejemplos anteriores en código.

```
In [1]: # Nota, no angustiarse si no entienden todo, abajo se explica en detalle, pero veanlo
        class Casa:
            def __init__(self, numero_casa, direccion):
                self.numero = numero_casa
                self.ubicacion = direccion
                self.ocupada = False
                print('Hemos creado una nueva casa!')
            def tocar_puerta(self):
                if self.ocupada:
                    print('Knock Knock!')
                else:
```

```
                print('No hay nadie.. :(')
        def habitar(self):
            if self.ocupada:
                print('Esta casa ya está ocupada.')
            else:
                self.ocupada = True
                print('La casa en', self.ubicacion, str(self.numero), 'ahora está ocupada.

    # Creamos el condominio
    condominio = []
    for i in range(1000, 1010): # Creo 10 casas. Con numeros del 1000 al 1009
        condominio.append(Casa(i, 'Avenida San Joaquin'))
    # Habitemos las 5 primeras casas

    habitadas = []

    for i in range(5):
        condominio[i].habitar()
        habitadas.append(condominio[i])
```

```
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
Hemos creado una nueva casa!
La casa en Avenida San Joaquin 1000 ahora está ocupada.
La casa en Avenida San Joaquin 1001 ahora está ocupada.
La casa en Avenida San Joaquin 1002 ahora está ocupada.
La casa en Avenida San Joaquin 1003 ahora está ocupada.
La casa en Avenida San Joaquin 1004 ahora está ocupada.
```

En el ejemplo anterior se muestran hartas cosas: - `class Casa`: Creo el modelo o clase de una casa. - `def __init__(self, valor0, valor1, ... , valorN)`: Define qué hacer cuando se crea una nueva instancia de la clase. **Siempre debe definirse** - `self.atributo = algo`: le doy propeidades o valores que son distintos en cada instancia de la clase, **pero** siguen un patron (ej. en la casa, `self.numero` es el número de la casa, que cambia para cada casa, pero siempre es un número) - `def metodo(self)`: corresponde a una función propia de clase. Toma los valores de la instancia, y hace cosas con estos. **importante**, siempre tiene un *self*, y si quiero agregar parametros al método, entonces me queda algo así: `def metodo(self, valor0, valor1, ... , valorN)`. Si no lo notaron **init**, es un metodo especial, pero es un metodo.

importante Así como en el ejemplo uso métodos de la clase en 5 instancias, tambien puedo obtener valores de sus atributos!

```
In [2]: for casa in habitadas:
            print('Habitada:', casa.ocupada)
```

```
Habitada: True
Habitada: True
Habitada: True
Habitada: True
Habitada: True
```

Veamos que ocurre cuando tratamos de habitar casa ya ocupadas:

```
In [3]: for casa in habitadas:
            casa.habitar()
```

```
Esta casa ya está ocupada.
Esta casa ya está ocupada.
Esta casa ya está ocupada.
Esta casa ya está ocupada.
Esta casa ya está ocupada.
```

Ahora mezclemos todo, habitando todas las casa desocupadas:

```
In [4]: for casa in condominio:
            if not casa.ocupada:
                casa.habitar()
```

```
La casa en Avenida San Joaquin 1005 ahora está ocupada.
La casa en Avenida San Joaquin 1006 ahora está ocupada.
La casa en Avenida San Joaquin 1007 ahora está ocupada.
La casa en Avenida San Joaquin 1008 ahora está ocupada.
La casa en Avenida San Joaquin 1009 ahora está ocupada.
```

2.2 Clases dentro de clases y metodos *dunder* ó operadores

Otra gran utilidad de las clases es usarlas dentro de otras. Sí, se puede, veamos el ejemplo del bus escolar.

```
In [2]: # Nuevamente, no angustiarse, a todos nos costó esta parte!!
        class Escolar:
            def __init__(self, nombre, apellido, edad, tiene_bebida_en_mochila):
                self.nombre_completo = nombre + ' ' + apellido
                self.tiene_bebida = tiene_bebida_en_mochila
                self.edad = edad
            def agitar(self):
                if self.tiene_bebida:
                    print('Oh no! Ha quedado la mansaca en la mochila de ', self)
```

```
    def __str__(self):
        return self.nombre_completo
    def __repr__(self):
        return self.nombre_completo

class Chofer:
    def __init__(self, nombre, porcentaje_vision):
        self.nombre = nombre
        self.porcentaje_vision = porcentaje_vision
    def __str__(self):
        string = self.nombre
        if self.porcentaje_vision <= 50:
            string += ' (medio ciego)'

class Bus:
    def __init__(self):
        self.chofer = None
        self.pasajeros = []
    def asignar_chofer(self, chofer):
        self.chofer = chofer
    def subir_escolar(self, escolar):
        print(escolar, 'ahora está dentro del bus!')
        self.pasajeros.append(escolar)
    def ir_a_colegio(self):
        if self.chofer.porcentaje_vision < 50:
            # Todos son agitados en el camino
            for escolar in self.pasajeros:
                escolar.agitar()
        else:
            # La mitad es agitada
            for escolar in self.pasajeros[:len(self.pasajeros) //2]:
                escolar.agitar()

# Creamos dos buses
bus_ciego = Bus()
bus_no_ciego = Bus()
# Creamos dos choferes y los asignamos

chofer_ciego = Chofer('Benjamin', 40)
chofer_no_ciego = Chofer('Pablo', 60)

bus_ciego.asignar_chofer(chofer_ciego)
bus_no_ciego.asignar_chofer(chofer_no_ciego)

nombres = ['A', 'B', 'C', 'D', 'E', 'F']
apellidos = ['1', '2', '3', '4', '5', '6']

# Creamos 10 escolares por bus y los subimos
```

```python
import random # Ignorar si no lo conocen

print('===Bus ciego===')
for _ in range(10): # El _ se usa cuando no me interesa usar el valor de i.
    # Nombre aleatorio, apellido aleatorio, edad aleatoria entre 1 y 18, y aleatorio s
    escolar = Escolar(random.choice(nombres), random.choice(apellidos), random.randint
    bus_ciego.subir_escolar(escolar)
print('===Bus no ciego===')
for _ in range(10):
    escolar = Escolar(random.choice(nombres), random.choice(apellidos), random.randint
    bus_no_ciego.subir_escolar(escolar)

print('Bus ciego va al colegio')
bus_ciego.ir_a_colegio()

print('Bus no ciego va al colegio')
bus_no_ciego.ir_a_colegio()
```

```
===Bus ciego===
B 5 ahora está dentro del bus!
C 3 ahora está dentro del bus!
A 6 ahora está dentro del bus!
E 2 ahora está dentro del bus!
A 6 ahora está dentro del bus!
C 5 ahora está dentro del bus!
A 5 ahora está dentro del bus!
C 6 ahora está dentro del bus!
A 1 ahora está dentro del bus!
A 3 ahora está dentro del bus!
===Bus no ciego===
D 6 ahora está dentro del bus!
E 6 ahora está dentro del bus!
C 6 ahora está dentro del bus!
C 4 ahora está dentro del bus!
B 5 ahora está dentro del bus!
D 2 ahora está dentro del bus!
F 1 ahora está dentro del bus!
C 3 ahora está dentro del bus!
C 3 ahora está dentro del bus!
D 2 ahora está dentro del bus!
Bus ciego va al colegio
Oh no! Ha quedado la mansaca en la mochila de  C 3
Oh no! Ha quedado la mansaca en la mochila de  E 2
Oh no! Ha quedado la mansaca en la mochila de  C 5
Oh no! Ha quedado la mansaca en la mochila de  A 1
Oh no! Ha quedado la mansaca en la mochila de  A 3
Bus no ciego va al colegio
```

```
Oh no! Ha quedado la mansaca en la mochila de  D 6
Oh no! Ha quedado la mansaca en la mochila de  B 5
```

Ok, harto paso harto aca arriba, veamos. - `Escolar` tiene un nombre_completo como apellido, y puede o no tener una bebida en su mochila. Si tiene una bebida, y el bus se agita en su asiento, su mochila se mancha. - `Chofer` tiene nombre y un porcentaje de visión, si el porcentaje es menor a 50, se considera ciego. - `Bus` tiene un solo chofer, y varios pasajeros. Si el chofer es ciego, al ir al colegio todos los puestos se agitan. Si no, la mitad de los `escolares` se agitan.

Ahora, qué son esas cosas extrañas que aparecen con `__algo__`??? Esos son los que se conocen como métodos *dunder*. Estos métodos son métodos mágicos que tiene python, y se ejecutan solos cuando ocurren ciertas cosas en el código:

- **`__str__(self)`**

Se llama cuando se necesita la versión string de una instancia de una clase (objeto). Por ejemplo:

```
In [3]: escolar_prueba = bus_ciego.pasajeros[0] # Sacamos una instancia de escolar, que tiene
        print(escolar_prueba)
        print(escolar_prueba, 'con otra cosa separada por espacios')
        print(str(escolar_prueba) + 'STRING')
        print('escolar {} formateado'.format(escolar_prueba))
```

```
B 5
B 5 con otra cosa separada por espacios
B 5STRING
escolar B 5 formateado
```

- **`__repr__(self)`**

Se llama en dos ocaciones: 1. Cuando se quiere la verisón string del objeto, y no se tiene el `__str__` 2. Cuando se imprime una lista con objetos de este tipo

Veamos el segundo caso, dado que el primero ya fue mostrado.

```
In [6]: print(bus_no_ciego.pasajeros)
```

```
[F 2, C 3, D 2, A 5, E 4, E 6, E 1, B 4, C 1, F 1]
```

Ambas anteriores **deben retornar un str** - **`__add__(self, other)`**

Se llama buando se hace instancia1 + instancia2. Ejemplo : `nuevo_escolar = escolar1 + escolar2`. Ahora bien, eso no resultaría ahora, dado que no está definido para la clase `Escolar`(porque no tiene sentido tampoco).

Lo que retorna add no esta definido, pero **se espera** que retorne una nueva instancia de la misma clase.

```
In [9]: class VectorR2:
            def __init__(self, x, y):
                self.x = x
                self.y = y
            def __add__(self, other):
                return VectorR2(self.x + other.x, self.y + other.y)
            def __str__(self):
                return 'Posición x: {}. Posición y: {}'.format(self.x, self.y)
        vector1 = VectorR2(1,1)
        print('Vector 1:')
        print(vector1)  # En Python, print(objeto) llama implícitamente al __str__ de objeto!!

        vector2 = VectorR2(1,2)
        print('Vector 2:')
        print(vector2)

        vector3 = vector1 + vector2 # Aqui se llama add del vector1 con other=vector2!
        print('Vector 3:')
        print(vector3)
```

```
Vector 1:
Posición x: 1. Posición y: 1
Vector 2:
Posición x: 1. Posición y: 2
Vector 3:
Posición x: 2. Posición y: 3
```

- **`__lt__(self, other)`**

  Se llama cuando se hace instancia1 < instancia2.

- **`__eq__(self, other)`**

  Se llama cuando se hace instancia1 == instancia2.
  Ambos deben retornar un *boolean* (`True`, `False`).

```
In [12]: class VectorR2:
             def __init__(self, x, y):
                 self.x = x
                 self.y = y
             def __add__(self, other):
                 return VectorR2(self.x + other.x, self.y + other.y)

             def __lt__(self, other):
                 norma1 = (self.x**2 +self.y**2)**0.5
                 norma2 = (other.x**2 + other.y**2)**0.5
                 return norma1 < norma2
             def __eq__(self, other):
```

```
            return self.x == other.x and self.y == other.y
        def __str__(self):
            return 'Posición x: {}. Posición y: {}'.format(self.x, self.y)
    vector1 = VectorR2(1,1)
    print('Vector 1:')
    print(vector1)

    vector2 = VectorR2(1,2)
    print('Vector 2:')
    print(vector2)

    vector3 = vector1 + vector2 # Aqui se llama add del vector1 con other=vector2!
    print('Vector 3:')
    print(vector3)

    print()

    print('Es el vector 1 menor al vector 2?')
    print(vector1 < vector2) # aqui other=vector2
    print('Es el vector 3 igual al vector 2?')
    print(vector3 == vector2) # other=vector2
```

```
Vector 1:
Posición x: 1. Posición y: 1
Vector 2:
Posición x: 1. Posición y: 2
Vector 3:
Posición x: 2. Posición y: 3

Es el vector 1 menor al vector 2?
True
Es el vector 3 igual al vector 2?
False
```