Calendario
Programación Orientada a Objetos

- Introducción
- Definición de Clases
- Métodos especiales de una clase.
- Sobrecarga de operadores
- Excepciones
- Polimorfismo
- Iteradores
- Herencia
- Delegación
- Ejercicios

Introducción

- **Objeto**: Permite organizar y agrupar datos (atributos) y operaciones a utilizar sobre esos datos (métodos)
- **Clase**: Se trata de la definición del tipo del objeto. Es un molde a partir del cual crear objetos. Cada instancia de la clase (instancia de clase = objeto creado a partir de la clase) tendrá los mismos atributos y métodos, pero con diferentes valores almacenados.
- Venimos utilizando objetos desde las primeras clases, pues todos los tipos de datos que Python nos provee son en realidad, objetos.

Recordemos

- Tipos
  - Ya hemos visto: números, cadenas de caracteres, listas, tuplas, diccionarios, archivos, excepciones, etc.
- Todos esos tipos son objetos y como tal, tienen atributos y métodos asociados. Contamos con las funciones:
  - type(a) → devuelve el tipo de la variable (a qué clase pertenece)
  - dir(a) → retorna todos los métodos y funciones asociados al tipo de la variable a (o sea, a la clase).

Definiendo nuevas clases

- Python provee varios tipos diferentes. Pero muchas veces es necesario definir nuestros propios tipos de objetos (clases).
- Por ejemplo:
  - Supongamos que queremos definir nuestra clase Punto que representa un punto en el plano de dos dimensiones x,y (coordenadas cartesianas).

Definiendo nuevas clases (2)

```
class Punto(object):
        """ Representación de un punto en el plano, los atributos son x e y
        que representan los valores de las coordenadas cartesianas."""
        def __init__(self, x=0, y=0):
                "Constructor de Punto, x e y deben ser numéricos"
                self.x = x
                self.y = y
```

En la primera línea de código, utilizando la palabra reservada **class**, indicamos que vamos a crear una nueva clase, llamada **Punto**

La palabra object entre paréntesis indica que la clase que estamos creando es un objeto básico y no hereda ningún otro comportamiento (podría omitirse).

Método constructor

- Por convención, en los nombres de las clases se escribe cada palabra del nombre con la primera letra en mayúscula. Ejemplos: Punto, ListaEnlazada.
- El **constructor** de una clase se define con uno de los métodos especiales que **debe tener** toda clase.
  - __init__(self) : Se llama cada vez que se crea (se construye) una nueva instancia de la clase (o sea, un objeto, o una instancia).
  - Recibe siempre como primer parámetro la instancia sobre la que se trabaja (**self**).
  - Pero además puede recibir otros parámetros (en el ejemplo de la clase Punto, el constructor recibe los parámetros x e y ("x" e "y" son nombres, podrían llamarse. coord_x y coord_y)

Atributos

- Para definir atributos de una clase, basta con hacerlo en el constructor precedidos por un atributo especial: **self** (llamado a veces **objeto implícito**).
- En el ejemplo de la clase Punto *self.x* y *self.y* (los atributos **x** e **y**), representan las coordenadas del punto (veremos luego que éstos atributos también se llaman atributos de instancia)
- Los atributos se pueden acceder desde fuera de la clase, anteponiendo el nombre del objeto, quedan públicos.
- Ejemplo: **creación de un objeto** (en este caso llamado p, podría llamarse de cualquier otra manera)

```
p = Punto(5,7)
```

- Se creó el objeto **p** de la clase Punto. Implícitamente se llamó al constructor (método __init__). Tal como definimos, al constructor, (usado para crear un objeto de la clase Punto), hay que pasarle en 2 atributos, x e y, que en este ejemplo valen 5 y 7.

Atributos (2)

- Si queremos imprimir p:

  print (p) → <__main__.Punto object at 0x8e4e24c>

- El tipo de p: *type(p)* → ***<class '__main__.Punto'>***
- *type* nos dice que p es una instancia de la clase Punto
- El acceso a los atributos es público, basta con <nombre_de_objeto>.<nombre_de_atributo>.
  - *print (p.x) → 5*
  - *print (p.y) → 7*
- Recordar: el nombre del atributo se define dentro del constructor de la clase
- También es posible definir atributos de clase o estáticos (en lugar de atributos de instancia). O sea, no necesitamos una instancia para acceder a ellos

```
class Punto(object):
    var = "pepe"
    def __init__(self, x, y):
        self.x = x
        self.y = y
print (Punto.var)
```

Atributos y métodos privados

- Python no incorpora el concepto privacidad para métodos y atributos.
- Todos los atributos y métodos son públicos
- Hay una convención para especificar si un atributo es privado.
- A diferencia de la mayoría de lenguajes, lo que hace que un método o un atributo de Python sea privado, es su nombre.
- Si el nombre del método o atributo comienza con (pero no termina con) dos guiones bajos (underscores), es privado; todo lo demás es público.

Métodos __str__

- Vimos el constructor **__init__** pero existen otros métodos (funciones) especiales que se invocan implícitamente en situaciones puntuales.
- Un método clásico y muy útil es **__str__** (usualmente llamado **to string**). Se usa para mostrar y castear objetos a string. Lo que hace __str__ es generar y retornar una representación en string de un objeto

```
def __str__(self):
        """ Muestra el punto como un par ordenado. """
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

print (p) → Muestra en pantalla (-6, 18)

- Se llama implícitamente al método __str__

Definiendo métodos y sobrecargando operadores

- Supongamos que queremos restar dos puntos, podemos implementar un método:

```
def restar(self, otro):
        """ Devuelve un nuevo punto, con la resta entre dos puntos. """
        return Punto(self.x - otro.x, self.y - otro.y)
```

Luego, de afuera de la clase:

p=Punto(6,4)

q=Punto(10,4)

punto_restado=p.restar(q)

- Pero también podemos utilizar el operador “-” y “+”. Son operadores que pueden ser sobrecargados fácilmente:
- Sobrecargar: redefinir el uso de un operador de acuerdo a los tipos con que opera.

```
def __sub__(self, otro):
        """ Devuelve la resta de ambos puntos. """
        return Punto(self.x - otro.x, self.y - otro.y)

def __add__(self, otro):
        """ Devuelve la suma de ambos puntos. """
        return Punto(self.x + otro.x, self.y + otro.y)
```

Ejemplo de sobrecarga

- Notar el uso del operador menos (-) para restar dos puntos (p1 y p2)

```
class Punto(object):
    var = "pepe"
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __sub__(self, otro):
        return Punto (self.x - otro.x, self.y - otro.y)
    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) +  ")"

p1 = Punto (2,3)
p2 = Punto (12,23)
print(p1 - p2)
```

Sobrecarga de operadores

- De la misma forma, si se quiere utilizar cualquier otro operador matemático (+, -, *, /), será necesario definir el método apropiado.
- __add__
- __sub__
- __mul__
- __pow__
- __truediv__
- __floordiv__
- __mod__
- Al momento de definir los métodos se debe devolver una nueva instancia; nunca modificar el objeto actual (self).

Métodos para comparar objetos

- También podemos utilizar los operadores de comparación <, <=, > y >=
  - Para ello deben redefinirse:
    - __lt__
    - __le__
    - __gt__
    - __ge__
    - __eq__
    - __ne__
- En particular, esto es útil cuando se desea que los objetos puedan ser ordenados.

Comparación de objetos

- Si no tenemos sobrecargado el método __eq__ (equal) o __ne__ (not equal), cuando queremos comparar objetos:

```
p = Punto(3,4)
q = Punto(3,4)
print (p==q) → False
q = p
print (p==q) → True
```

- Son objetos diferentes, a pesar de que tengan los mismos datos adentro. Luego, cuando hacemos que los objetos sean iguales (q = p) lo que se iguala son direcciones de memoria, pero no se mira lo que está dentro de los objetos.
- Sin embargo, cuando sobrecargamos los métodos:

```
p = Punto(3,4)
q = Punto(3,4)
print (p==q) → True
```

```
def __eq__(self, otro):
        """ Devuelve si dos puntos son iguales. """
        return self.x == otro.x and self.y == otro.y

def __ne__(self, otro):
        """ Devuelve si dos puntos son distintos. """
        return not self == otro
```

Ordenar colecciones de objetos

- Supongamos que queremos ordenar una lista de objetos:

```
p1 = Punto (2,3)
p2 = Punto (12,1)
lista_puntos = []
lista_puntos.append(p1)
lista_puntos.append(p2)
```

- Si tenemos sobrecargados los métodos de comparación, se usarán (por defecto) en **sorted**

```
sorted_default = sorted(lista_puntos)
```

- También podemos especificar por qué atributo queremos que se ordene(utilizando el parámetro opcional *key*), de cualquiera de estas dos maneras:

```
import operator
sorted_x = sorted(lista_puntos, key=operator.attrgetter('y'))
sorted_x = sorted(lista_puntos, key = lambda p: p.y)
```

Volviendo a las excepciones

- Podemos usar excepciones en el constructor, cuando algunos de los valores no se ajusta a lo requerido para crear objetos de esa clase.
- En este ejemplo, e el caso de que alguno de los valores no sea numérico, lanzamos una excepción del tipo **TypeError**

```
def __init__(self, x=0, y=0):
""" Constructor de Punto, x e y deben ser numéricos,
de no ser así, se levanta una excepción TypeError """
        if es_numero(x) and es_numero(y):
                self.x=x
                self.y=y
        else:
                raise TypeError("x e y deben ser valores numéricos")
```

Ejercicio

- Definir una clase perro que contenga un atributo de clase que especifique el tipo de animal: “Mamifero” y dos atributos del objeto que especifique el nombre y la edad.
- Definir el metodo str para que cuando se haga el print de un perro muestre el nombre y la edad del perro.
- Definir el metodo ladrar que imprime el texto correspondiente al ladrido de su perro, pasado por parámetro
- Crear 3 instancias de tipo perro con edades diferentes y retornar el mayor de esos. Para comparar la edad de los perros se debe utilizar los operadores > y >=
- Investigar cuál metodo habría que definir en una clase para que cuando se pregunte por el largo (len) de un perro devuelve su edad.