Programación orientada a objetos

Clases y objetos

- Hasta ahora hemos hemos visto que en Python existen diferentes tipos de datos.
- Estos tipos podían ser simples (números, texto) o estructurados (listas, tuplas, diccionarios, etc.).
- Además, hemos visto que cada tipo de dato traía consigo una serie de métodos (funciones) propias.
- Las **clases** amplían la definición de tipo introduciéndole métodos y propiedades propios.

Clases y objetos

- Llamamos **clase** al conjunto de datos, métodos y propiedades que abstraen o representan un concepto.
- Llamamos **objeto** a toda instancia creada a partir de una clase.
- Una forma de entender esto es considerar a la clase como una plantilla y a los objetos como elementos creados a través de ella.
- En Python 3 todos los datos son instancias de alguna clase, es decir, **todo son objetos**.

Ejemplos

- Perros:

```
toby = Perro('Toby')
toby.ladrar()
for pata in toby.patas():
    print(pata)
```

- Cuenta bancaria:

```
mi_cuenta = CCorriente(0)
otra_cuenta = CCorriente(12000)
otra_cuenta.transferencia(1000, mi_cuenta)
print(mi_cuenta.saldo())
```

Instancias de una clase

- Como hemos mencionado anteriormente, un objetos es una instancia de una clase, es decir, un modelo creado a partir de ella.
- La palabra instancia tradicionalmente se refiere a una petición administrativa o a los diferentes grados jurisprudenciales. Sin embargo, en programación, tomamos prestado del inglés el significado como ‘ejemplo’.
- Para crear instancias de una clase solo hay que invocarlas. Ej: mi_coche = Coche()
- Podemos saber si un objeto es instancia de una clase con la función isinstance().

Definición de una clase

- Para definir una clase usamos la palabra class:

```
class A:
    pass
```

- Podemos añadir datos que formarán parte de la clase. A estos datos los llamamos **atributos**:

```
class A:
    a = 3

x = A()

print(x.a) → 3

x.a = 5
```

Definición de una clase

- Una vez creado el objeto, los atributos de la clase pasan a ser suyos:

```
class A:
    a = 1

x = A()

x.a = 6

y = A()

print(y.a) → 1

print(A.a) → También podemos acceder a los atributos de la clase.
```

Definición de una clase

- Las funciones que definamos dentro de la clase pasarán a ser métodos de la misma.
- Cuando llamamos a un método desde la instancia de una clase estamos pasando como primer argumento la propia instancia. Por eso para usar los métodos desde un objeto necesitamos definirlos de la siguiente manera:

```
class A:
    def metodo(self):
        print('Hola')
x = A()
x.metodo() → 'Hola'
A.metodo() → Error, le falta el argumento 'self'.
A.metodo(x) → 'Hola'
```

Definición de una clase

- Podemos acceder a los atributos y métodos de la instancia a través de self.

```
class Cuenta:
    n = 0
    def suma(self, c = 1):
        self.n += c
        return self.n
    def reset(self):
        self.n = 0
x = Cuenta()
x.siguiente(2) → 2
x.reset()
```

Definición de una clase

- Podemos acceder a los atributos y métodos de la instancia a través de self.

```
class Cuenta:
    n = 0
    def suma(self, c = 1):
        self.n += c
        return self.n
    def reset(self):
        self.n = 0
    def muestra(self):
        self.suma(0)
```

```
x = Cuenta()
x.siguiente(2) → 2
x.reset()
x.muestra() → 0
```

Inicialización de una clase

- La forma más común de inicializar los atributos de una clase es a través de un método especial llamado __init__:

```
class A:
    def __init__(self, a):
        self.n = a
    def imprime(self):
        print(self.n)
```

```
x = A(2)
x.imprime() → 2
```

Ejemplo cuenta bancaria

```
class CCorriente:
  def __init__(self, saldo, nombre):
    self.saldo = saldo
    self.nombre = nombre
  def ver_saldo(self):
    print('El saldo es:', self.saldo)
  def ingreso(self, cantidad):
    self.saldo += cantidad
  def retirada(self, cantidad):
    if self.saldo - cantidad  >= 0:
      self.saldo -= cantidad
      return True
    return False
  def transferencia(self, dest, ctd):
    if self.retirada(ctd):
      dest.ingreso(ctd)
      return True
    return False
```

```
c1 = CCorriente(1000, 'Juan')
c2 = CCorriente(0, 'Pepe')

c1.ingreso(300)
c1.ver_saldo()

c2.ingreso(200)
if c2.retirada(500):
    print('Operación efectuada')
else:
    print('Fondos insuficientes')
c2.ver_saldo()

if c1.transferencia(c2, 400):
    print('Operación efectuada')
else:
    print('Fondos insuficientes')
c1.ver_saldo()
c2.ver_saldo()
```

Métodos mágicos

- Existen un grupo de métodos especiales que permiten modificar las clases y objetos a alto nivel. Estos métodos son conocidos familiarmente como ‘mágicos’.
- La característica más identificable de estos métodos es que el nombre de estos empieza y acaba con dos barras bajas (__init__, __len__, __eq__, etc.).
- Existen muchos métodos mágicos, a continuación veremos algunos de los más usados.

Métodos mágicos

- `__init__`: método que se ejecuta cuando se inicia el objeto.
- `__len__`: método que se ejecuta cuando pasamos el objeto por la función `len()`.

```
class A:
    a = []
    def __len__(self):
        return len(self.a)

x = A()

x.a.append(10)

len(x) → 1
```

Métodos mágicos

- __add__(+), __sub__(-), __mul__(*), __truediv__(/), __floordiv__(//), __mod__(%), __pow__(**), __and__(and), __or__(or) para usar los operadores.

```
class A:
    a = 0
    def __add__(self, other):
        return self.a + other.a
x, y = A(), A()
x.a, y.a = 1, 2
print(x+y) → 3
```

Métodos mágicos

- __lt__(<), __le__(<=), __eq__(==), __ne_(!=), __gt__(>) y __ge__(>=) para comparaciones.

```
class A:
    a = 0
    def __eq__(self, other):
        return self.a == other.a

x,y = A(), A()

x.a, y.a = 1,2

print(x==y) → False
```

- La función sorted funciona simplemente definiendo los métodos __lt__ y __eq__.

Métodos mágicos

- __getitem__(self, key): se ejecuta cuando se pretende obtener un valor a través del operador de acceso [].
- __setitem__(self, key, valor): se ejecuta cuando se pretende escribir un valor a través del operador de acceso [].

```
class matriz:
  def __init__(self, m, n):
    self.m = m # Filas
    self.n = n # Columnas
    self.data = [[0]*n]*m
  def __getitem__(self, key):
    return self.data[key]
  def __setitem__(self, a, b, c = None):
    if c is None:
      self.data[a] = b
    else:
      self.data[a][b] = c
```

```
x = matriz(3,3)
print(x[:])
for i in range(3):
    for j in range(3):
        if i == j:
            x[i][j] = 1
print(x[:]) # Matriz identidad
```

Métodos mágicos

- __repr__: muestra una representación del objeto (cuando lo pasamos por print(), por ejemplo).

  Ampliando el ejemplo anterior:

```
class matriz:
  def __init__(self, m, n):
    self.m = m # Filas
    self.n = n # Columnas
    self.data = [[0]*n]*m
  def __getitem__(self, key):
    return self.data[key]
  def __setitem__(self, a, b, c = None):
    if c is None:
      self.data[a] = b
    else:
      self.data[a][b] = c
  def __repr__(self):
    for fila in self.data:
      out += '\t' + str(fila) + '\t\n'
    return '[' + out[:-1] + ']'
```

```
x = matriz(3,3)
print(x)
for i in range(3):
    for j in range(3):
        if i == j:
            x[i][j] = 1
print(x) # Matriz identidad
```

IEEE UPCT SB

Métodos mágicos

- __contains__: para el operador in.
- __call__: se ejecuta cuando haces una llamada al objeto (como una función).

```
class cestaFruta:
  frutas = []
  def __call__(self):
    print(len(self.frutas), self.frutas)
  def __contains__(self, fruta):
    return fruta in self.frutas
cesta = cestaFruta()
cesta.frutas.append('Melocotón')
cesta.frutas.append('Manzana')
cesta()
if not 'Pera' in cesta:
  cesta.frutas.append('Pera')
```

Ámbito de la clase

- La definición de una clase es por sí misma un ámbito. Esto quiere decir que las variables, funciones y otras clases definidas en ella no son accesibles desde fuera, solo usando el operador de acceso '.'.

```
class A:
    class B:
        pass
    a = B()
x = B() → Error
x = A.B() → Ok
```

Programación orientada a objetos

- Con Programación Orientada a Objetos (POO) nos referimos al paradigma de programación que se enfrenta a los problemas creado una representación de todos los elementos que aparecen en él así como de las relaciones entre ellos.
- Cuando un problema es complejo o simplemente muy extenso para manejar todos los datos en un ámbito global la programación orientada a objetos es útil porque crea capas de abstracción.
- La ventaja que ofrece la POO es en realidad de cara al usuario, nunca al programa.

POO vs Programación Estructurada

- Hasta ahora hemos programado según las directrices de la programación estructurada (bueno, más o menos, todo hay que decirlo).
- Pero, ¿qué paradigma es mejor? ¿programación estructurada o POO?.
- **Depende**. Una programación orientada a objetos pura a veces complica más las cosas y la programación estructurada trae muchos problemas de organización en programas grandes (cientos/miles de funciones en el ámbito global :S).
- La clave es saber analizar el contexto y utilizar en cada caso el paradigma correcto además de abandonar el purismo. A veces mezclar paradigmas es la solución.

Ejercicio: POO vs. P. Estructurada

Realiza el siguiente ejercicio bajo los paradigmas de programación estructurada y POO.

- Una calculadora de matrices que:
  - Sume
  - Reste
  - Producto de matriz por escalar
  - Producto de matrices

¿Qué ventajas ofrece la P. Estr.? ¿Y la POO?

Estilo POO.

- La programación orientada a objetos tiene un estilo propio. El uso de clases modifica un poco nuestros scripts:

```
import modulo

class Nombre: # El nombre en mayúsculas.
   pass

def foo():
   pass

if __name__ == '__main__':
   pass
```

1º Importación de módulos

2º Definición de clases

3º Definición de funciones

4º Programa principal

Estilo POO.

- Las clases tienen esta estructura:

```
class Matriz:
    '''Documentación'''
    PI = 3.14 → Los atributos primero. Y las constantes antes.
    _data = []
    def __init__(self, m: int, n: int): → El primer método es el constructor
        self._m = m → Mejor definir las variables en el constructor
        self._n = n
        self._data = [[0]*n]*m
    def __add__(self, other: Matriz) -> Matriz: → Después los métodos mágicos
        (...)
    def get_with(self) -> int:
        return self._n
    def get_height(self) -> int:
        return self._m
    def print(self): → Por último el resto de métodos.
        (...)
```

Los datos que no queramos que se accedan desde fuera deben ir precedidos por _.

La manera 'correcta' de acceder y cambiar los atributos es definiendo funciones 'getters' y 'setters' respectivamente.

Clases predefinidas.

- Por último vamos a ver algunas clases (tipos) útiles que podemos encontrar en algunos módulos.
- collections.deque: Como una lista pero con el acceso al primer y último elemento más rápido:

```
from collections import deque
import time
x, y = list(), deque()
t0 = time.monotonic() # Tiempo
for i in range(10000000):
  x.append(i)
for i in range(10000000):
  x.pop()
t_list = time.monotonic() - t0
t0 = time.monotonic()
for i in range(10000000):
  y.append(i)
for i in range(10000000):
  y.pop()
t_deque = time.monotonic() - t0
print('Tiempo lista:', t_list)
print('Tiempo deque:', t_deque)
```

Clases predefinidas.

- collections.defaultdict: diccionario con un tipo de valor por defecto. Gracias a esto podemos hacer cosas como estas:

```
from collections import defaultdict
# Diccionarios a partir de listas
casas = [('Azul', 1), ('Verde', 3), ('Azul', 5), ('Negro', 7),
         ('Rojo', 2), ('Negro', 4), ('Azul', 6), ('Rojo', 8)]
colores_casas = defaultdict(list)
for color, numero in casas:
    colores_casas[color].append(numero)
print(colores_casas)
# Contadores
S = 'missisipi'
letras = defaultdict(int)
for c in S:
    S[c] += 1
print(S)
```

Clases predefinidas.

- collections.namedtuple: namedtuple es una función que genera una clase (¿os acordáis?, ¡todo son objetos, hasta las clases!) de tipo tupla pero con nombres para acceder a los miembros. Es una buena forma de hacer estructuras simples.

```
from collections import namedtupled
Punto2D = namedtuple('Punto2D', ('x', 'y'))
a = Punto2D(2, 3)
b = Punto2D(1, -1)
a.x = 1
print(a, b)
Punto3D = namedtuple('Punto3D', Punto2D._fields + ('z',))
c = Punto3D(1, 2, 3)
print(c)
```

Clases predefinidas.

- collections.OrderedDict: Diccionario que recuerda el orden en el que se insertan los objetos. Permite popitem() y mover un par clave-valor al final con move_to_end().
- Existen más tipos de datos predefinidos interesantes contenidos en módulos de la librería estándar de Python:
  - Para definir y operar el tiempo: time, datetime, calendar
  - Para hacer interfaces gráficas: tkinter
  - Gestión de constantes: enum
- Más allá de la librería estándar tenemos: numpy, pandas, el paquete SciPy, matplotlib, etc.