Conceptos de programación orientada a objetos

Fuente: tutorialesprogramacionya.com / docs.hektorprofe.net / https://cosasdedevs.com/post

Python nos permite utilizar distintas metodologías de programación. Hemos implementado inicialmente programas utilizando la programación lineal, luego vimos funciones y trabajamos con programación estructurada.
Ahora introduciremos los conceptos de programación orientada a objetos. A partir de este concepto mostraremos en forma sencilla la metodología de Programación Orientada a Objetos.
Se irán introduciendo conceptos de objeto, clase, atributo, método etc. y de todos estos temas se irán planteando problemas resueltos.
Prácticamente todos los lenguajes desarrollados en los últimos 25 años implementan la posibilidad de trabajar con POO (Programación Orientada a Objetos)
El lenguaje Python tiene la característica de permitir programar con las siguientes metodologías:

- **Programación Lineal:** Es cuando desarrollamos todo el código sin emplear funciones. El código es una secuencia lineal de comando.
- **Programación Estructurada:** Es cuando planteamos funciones que agrupan actividades a desarrollar y luego dentro del programa llamamos a dichas funciones que pueden estar dentro del mismo archivo (módulo) o en una librería separada.
- **Programación Orientada a Objetos:** Es cuando planteamos clases y definimos objetos de las mismas (Este es el objetivo de los próximos conceptos, aprender la metodología de programación orientada a objetos y la sintaxis particular de Python para la POO)

PARADIGMA

Un paradigma de programación es un estilo de desarrollo de programas. Es decir, un modelo para resolver problemas computacionales. Los lenguajes de programación, necesariamente, se encuadran en uno o varios paradigmas a la vez a partir del tipo de órdenes que permiten implementar, algo que tiene una relación directa con su sintaxis.
¿Cuáles son los principales paradigmas de programación?

- **Imperativo.** Los programas se componen de un conjunto de sentencias que cambian su estado. Son secuencias de comandos que ordenan acciones a la computadora.
- **Declarativo.** Opuesto al imperativo. Los programas describen los resultados esperados sin listar explícitamente los pasos a llevar a cabo para alcanzarlos.
- **Lógico.** El problema se modela con enunciados de lógica de primer orden.
- Funcional. Los programas se componen de funciones, es decir, implementaciones de comportamiento que reciben un conjunto de datos de entrada y devuelven un valor de salida.
- **Orientado a objetos**. El comportamiento del programa es llevado a cabo por objetos, entidades que representan elementos del problema a resolver y tienen atributos y comportamiento.

- **Dirigido por eventos.** El flujo del programa está determinado por sucesos externos (por ejemplo, una acción del usuario).
- **Orientado a aspectos.** Apunta a dividir el programa en módulos independientes, cada uno con un comportamiento bien definido.

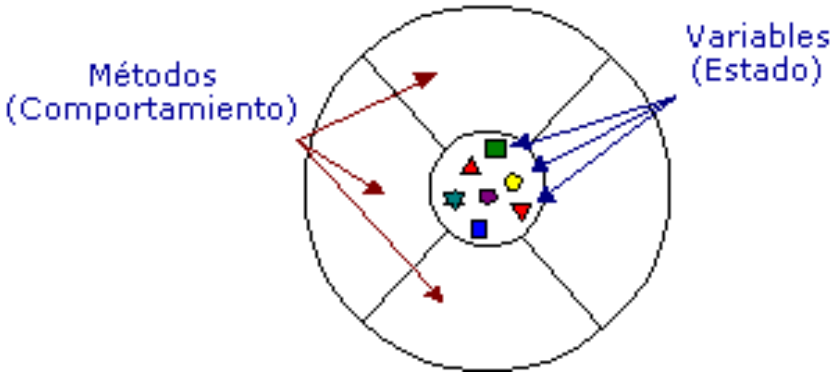
Cada paradigma es ideal para la resolución de un conjunto de problemas particular, por lo que no puede decirse que uno sea necesariamente mejor que otro.

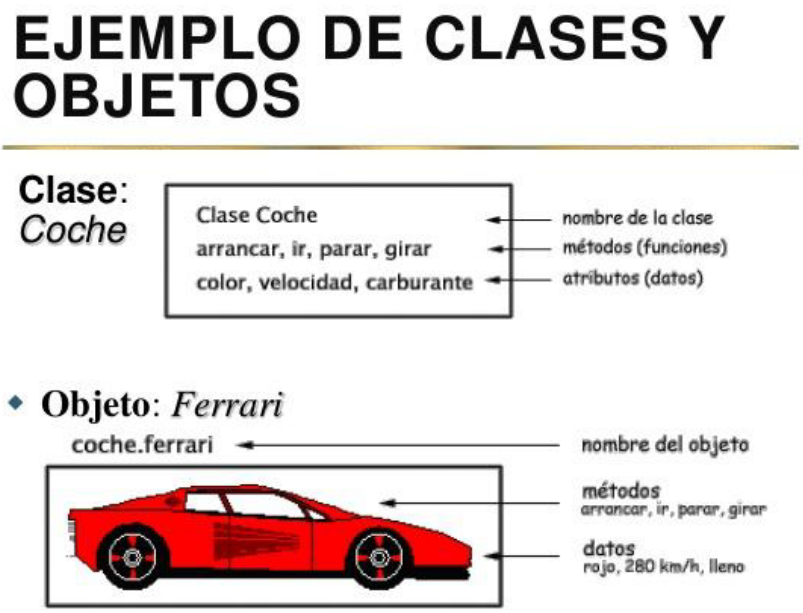
OBJETO

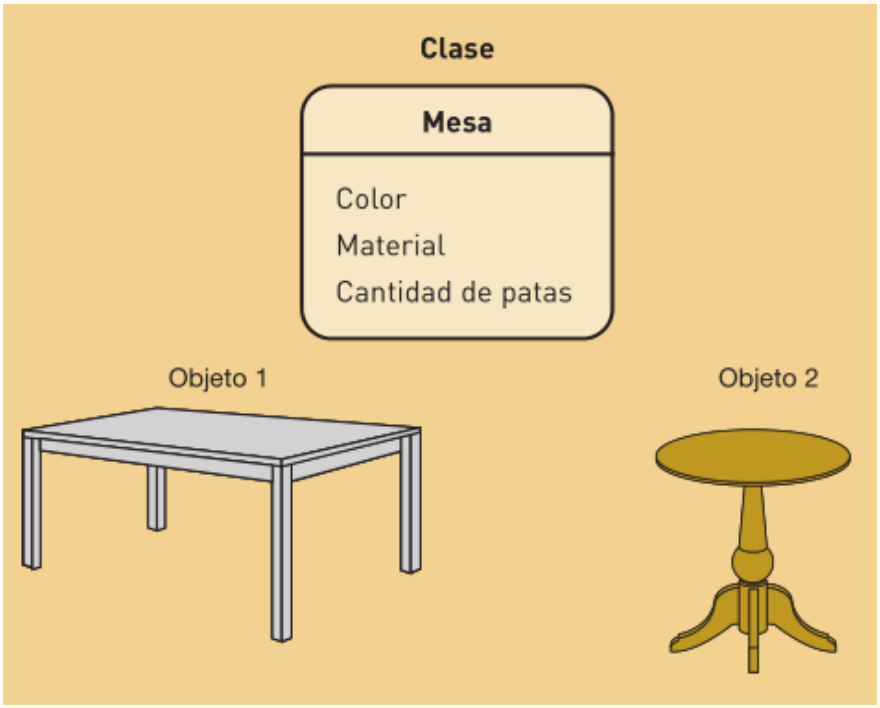
Un objeto es una encapsulación genérica de datos y de los procedimientos para manipularlos.

Al igual que los objetos del mundo real, los objetos de software tienen un estado y un comportamiento. El estado de los objetos se determina a partir de una o más variables y el comportamiento con la implementación de métodos.

La siguiente figura muestra la representación común de los objetos de software





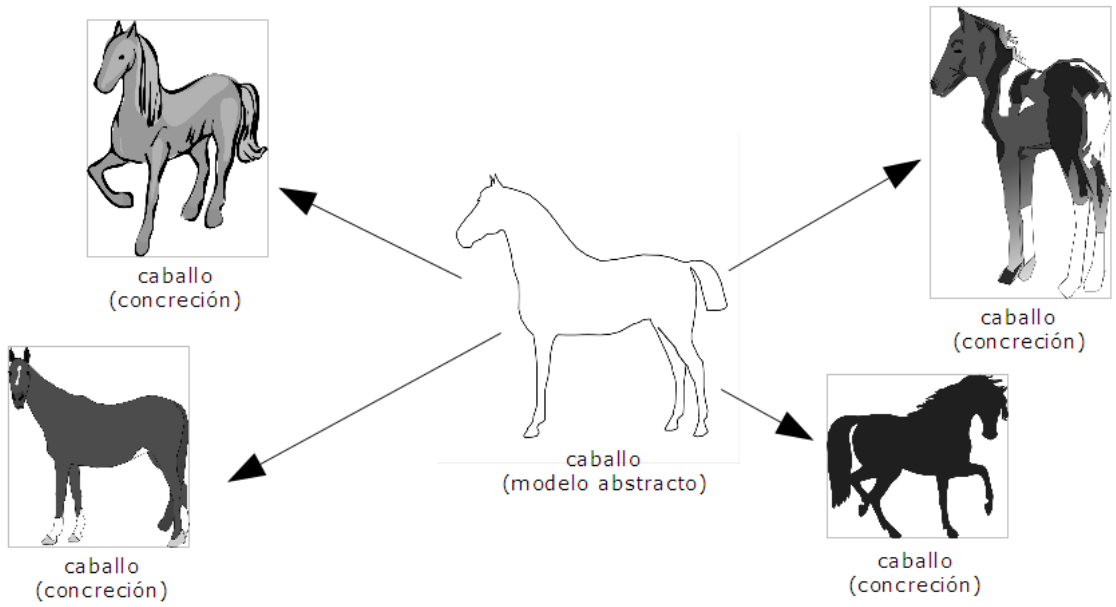


- **Atributos:** estos son los datos que caracterizan al objeto. Son variables que almacenan datos relacionados al estado de un objeto.
- **Métodos** (usualmente llamados **funciones de miembro**): Los métodos de un objeto caracterizan su comportamiento, es decir, son todas las acciones (denominadas operaciones) que el objeto puede realizar por sí mismo. Estas operaciones hacen posible que el objeto responda a las solicitudes externas (o que actúe sobre otros objetos). Además, las operaciones están estrechamente ligadas a los atributos, ya que sus acciones pueden depender de, o modificar, los valores de un atributo.
- **Identidad:** El objeto tiene una identidad, que lo distingue de otros objetos, sin considerar su estado. Por lo general, esta identidad se crea mediante un identificador que deriva naturalmente de un problema (por ejemplo: un producto puede estar representado por un código, un automóvil, por un número de modelo, etc.).

CLASE

Una clase está formada por los métodos y las variables que definen las características comunes a todos los objetos de esa clase. Precisamente la clave de la OOP está en abstraer los métodos y los datos comunes a un conjunto de objetos y almacenarlos en una clase.

Una clase equivale a la generalización de un tipo específico de objetos. Una instancia es la concreción de una clase.



En la figura anterior, el objeto A y el objeto B son instancias de la clase X.

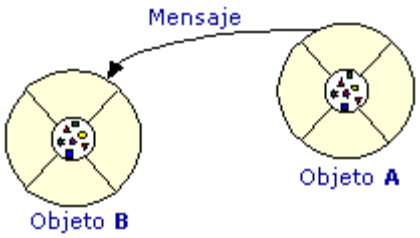
Cada uno de los objetos tiene su propia copia de las variables definidas en la clase de la cual son instanciados y comparten la misma implementación de los métodos.

MENSAJES Y MÉTODOS

El modelado de objetos no sólo tiene en consideración los objetos de un sistema, sino también sus interrelaciones.

Mensaje
Los objetos interactúan enviándose mensajes unos a otros. Tras la recepción de un mensaje el objeto actuará. La acción puede ser el envío de otros mensajes, el cambio de su estado, o la ejecución de cualquier otra tarea que se requiera que haga el objeto.

Método
Un método se implementa en una clase, y determina cómo tiene que actuar el objeto cuando recibe un mensaje.
Cuando un objeto A necesita que el objeto B ejecute alguno de sus métodos, el objeto A le manda un mensaje al objeto B.



Al recibir el mensaje del objeto A, el objeto B ejecutará el método adecuado para el mensaje recibido.

Declaración de una clase y creación de objetos

La programación orientada a objetos se basa en la definición de clases; a diferencia de la programación estructurada, que está centrada en las funciones.
Una clase es un molde del que luego se pueden crear múltiples objetos, con similares características.
Un poco más abajo se define una clase Persona y luego se crean dos objetos de dicha clase.
Una clase es una plantilla (molde), que define atributos (lo que conocemos como variables) y métodos (lo que conocemos como funciones).
La clase define los atributos y métodos comunes a los objetos de ese tipo, pero luego, cada objeto tendrá sus propios valores y compartirán las mismas funciones.
Debemos declarar una clase antes de poder crear objetos (instancias) de esa clase. Al crear un objeto de una clase, se dice que se crea una instancia de la clase o un objeto propiamente dicho.

Problema 1:

Implementaremos una clase llamada Persona que tendrá como atributo (variable) su nombre y dos métodos (funciones), uno de dichos métodos inicializará el atributo nombre y el siguiente método mostrará en la pantalla el contenido del mismo.
Definir dos objetos de la clase Persona.

Siempre conviene buscar un nombre de clase lo más próximo a lo que representa. La palabra clave para declarar la clase es class, seguidamente el nombre de la clase y luego dos puntos.
Los métodos de una clase se definen utilizando la misma sintaxis que para la definición de funciones.
Como veremos todo método tiene como primer parámetro el identificador self que tiene la referencia del objeto que llamó al método.
Luego dentro del método diferenciamos los atributos del objeto antecediendo el identificador **self**:

```
self.nombre="Pedro"
```

Con la asignación previa almacenamos en el atributo nombre el parámetro nom, los atributos siguen existiendo cuando finaliza la ejecución del método. Por ello cuando se ejecuta el método imprimir podemos mostrar el nombre que cargamos en el primer método.
Decíamos que una clase es un molde que nos permite definir objetos. Ahora veamos cual es la sintaxis para la creación de objetos de la clase Persona:

```
# bloque principal
persona1=Persona()
persona1.inicializar("Pedro")
persona1.imprimir()
persona2=Persona()
persona2.inicializar("Carla")
persona2.imprimir()
```

Definimos un objeto llamado **persona1** y lo creamos asignándole el nombre de la clase con paréntesis abierto y cerrado al final (como cuando llamamos a una función)
Luego para llamar a los métodos debemos disponer luego del nombre del objeto el operador **.** y por último el nombre del método (función)
En el caso que tenga parámetros se los enviamos (salvo el primer parámetro (self) que el mismo Python se encarga de enviar la referencia del objeto que se creó):
persona1.inicializar("Pedro")

También podemos definir tantos objetos de la clase Persona como sean necesarios para nuestro algoritmo:

```
persona2=Persona()
persona2.inicializar("Carla")
persona2.imprimir()
```

La declaración de clases es una de las ventajas fundamentales de la Programación Orientada a Objetos (POO), es decir reutilización de código (gracias a que está encapsulada en clases) es muy sencilla.

Problema 2:

Implementar una clase llamada Alumno que tenga como atributos su nombre y su nota.
Definir los métodos para inicializar sus atributos, imprimirlos y mostrar un mensaje si está regular (nota mayor o igual a 4)
Definir dos objetos de la clase Alumno.

Declaramos la clase Alumno y definimos sus tres métodos, en el método inicializar llegan como parámetros a parte del self el nombre y nota del alumno:

```
   def inicializar(self,nombre,nota):
        self.nombre=nombre
        self.nota=nota
```

No hay problema que los atributos se llamen iguales a los parámetros ya que siempre hay que anteceder la palabra "self" al nombre del atributo:

```
       self.nombre=nombre
```

Tener en cuenta que cuando se crean los atributos en el método inicializar luego podemos acceder a los mismos en los otros métodos de la clase, por ejemplo en el método **mostrar_estado** verificamos el valor almacenado en el atributo nota:

```
def mostrar_estado(self):
       if self.nota>=4:
           print("Regular")
       else:
           print("Libre")
```

Decimos que una clase es un molde que nos permite crear luego objetos de dicha clase, en este problema el molde Alumno lo utilizamos para crear dos objetos de dicha clase:

```
# bloque principal

alumno1=Alumno()
alumno1.inicializar("diego",2)
alumno1.imprimir()
alumno1.mostrar_estado()

alumno2=Alumno()
alumno2.inicializar("ana",10)
alumno2.imprimir()
alumno2.mostrar_estado()
```

Es fundamental la definición de objetos de una clase para que haya tenido sentido la declaración de dicha clase.

Método __init__ y __del__ de la clase

El método __init__ es un método especial de una clase en Python. El objetivo fundamental del método __init__ es **inicializar** los atributos del objeto que creamos.

El método __del__ también es un método especial que será ejecutado cuando termine la ejecución del programa y el objeto sea eliminado.

Básicamente el método __init__ reemplaza al método inicializar que habíamos hecho en el ejercicio anterior.

Las ventajas de implementar el método __init__ en lugar del método inicializar son:

1. El método __init__ es el primer método que se ejecuta cuando se crea un objeto.
2. El método __init__ se llama automáticamente. Es decir es imposible de olvidarse de llamarlo ya que se llamará automáticamente.
3. Quien utiliza POO en Python (Programación Orientada a Objetos) conoce el objetivo de este método.

Otras características del método __init__ son:

- Se ejecuta inmediatamente luego de crear un objeto.
- El método __init__ no puede retornar dato.
- el método __init__ puede recibir parámetros que se utilizan normalmente para inicializar atributos.
- El método __init__ es un método opcional, de todos modos es muy común declararlo.

Veamos la sintaxis del constructor:

```
def __init__(self):
        print('Método init llamado')
```

```
def __del__(self):
        print('Método delete llamado')
```

Debemos definir un método llamado __init__ (es decir utilizamos dos caracteres de subrayado, la palabra init y seguidamente otros dos caracteres de subrayado).

Problema 3:

Confeccionar una clase que represente un empleado. Definir como atributos su nombre y su sueldo. En el método __init__ cargar los atributos por teclado y luego en otro método imprimir sus datos y por último uno que imprima un mensaje si debe pagar impuestos (si el sueldo supera a 3000)

Este método se ejecuta inmediatamente luego que se crea un objeto de la clase Empleado:

```
empleado1=Empleado()
```

Como vemos no llamamos directamente al método __init__ sino que se llama automáticamente.
Los otros dos métodos tienen por objeto mostrar los datos del empleado y mostrar una leyenda si paga impuestos o no:

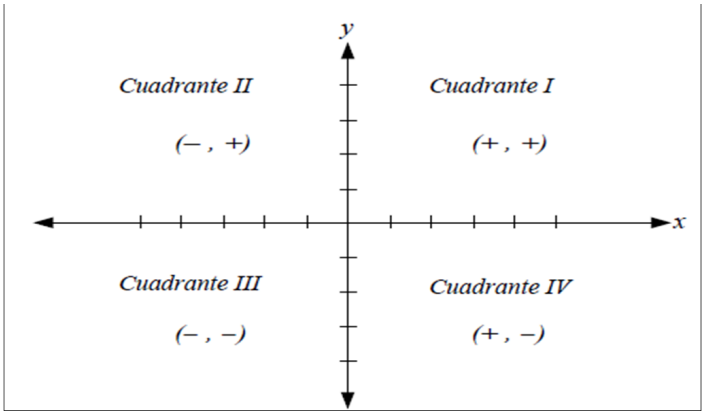
```
    def imprimir(self):
        print("Nombre:",self.nombre)
        print("Sueldo:",self.sueldo)

    def paga_impuestos(self):
        if self.sueldo>3000:
            print("Debe pagar impuestos")
        else:
            print("No paga impuestos")
```

Desde el bloque principal donde creamos un objeto de la clase Empleado debemos llamar explícitamente a estos dos métodos:

```
empleado1.imprimir()
empleado1.paga_impuestos()
```

Problema 4:

Desarrollar una clase que represente un punto en el plano y tenga los siguientes métodos: inicializar los valores de x e y que llegan como parámetros, imprimir en que cuadrante se encuentra dicho punto (concepto matemático, primer cuadrante si x e y son positivas, si x<0 e y>0 segundo cuadrante, etc.)



En este problema el método __init__ aparte del parámetro self que siempre va tenemos otros dos parámetros:

```
   def __init__(self,x,y):
        self.x=x
        self.y=y
```

Desde el bloque principal donde creamos un objeto de la clase **Punto** pasamos los datos a los parámetros:

```
punto1=Punto(10,-2)
```

Recordemos que pasamos dos parámetros aunque el método __init__ recibe 3. El parámetro self recibe la referencia de la variable punto1 (es decir el objeto propiamente dicho)

Llamada de métodos desde otro método de la misma clase

Hasta ahora todos los problemas planteados hemos llamado a los métodos desde donde definimos un objeto de dicha clase, por ejemplo:

```
empleado1=Empleado("Diego",2000)
empleado1.paga_impuestos()
```

Utilizamos la sintaxis:

```
[nombre del objeto].[nombre del método]
```

Es decir antecedemos al nombre del método el nombre del objeto y el operador punto
Ahora bien, qué pasa si queremos llamar dentro de la clase a otro método que pertenece a la misma clase, la sintaxis es la siguiente:

```
self.[nombre del método]
```

Es importante tener en cuenta que esto solo se puede hacer cuando estamos dentro de la misma clase.

Problema 5:

Plantear una clase Operaciones que solicite en el método __init__ la carga de dos enteros e inmediatamente muestre su suma, resta, multiplicación y división. Hacer cada operación en otro método de la clase Operación y llamarlos desde el mismo método __init__

Nuestro método __init__ además de cargar los dos enteros procede a llamar a los métodos que calculan la suma, resta, multiplicación y división de los dos valores ingresados.
La llamada de los métodos de la misma clase se hace antecediendo al nombre del método la palabra self:

```
  def __init__(self):
       self.valor1=int(input("Ingrese primer valor:"))
       self.valor2=int(input("Ingrese segundo valor:"))
       self.sumar()
       self.restar()
       self.multiplicar()
       self.dividir()
```

El método que calcula la suma de los dos atributos cargados en el método __init__ define una variable local llamada suma y guarda la suma de los dos atributos. Posteriormente muestra la suma por pantalla:

```
def sumar(self):
       suma=self.valor1+self.valor2
       print("La suma es",suma)
```

De forma similar los otros métodos calculan la resta, multiplicación y división de los dos valores ingresados:

En el bloque principal de nuestro programa solo requerimos crear un objeto de la clase Operación ya que el resto de los métodos se llama en el método __init__:

Problema 6:

Plantear una clase que administre dos listas de 5 nombres de alumnos y sus notas. Mostrar un menú de opciones que permita:
1- Cargar alumnos.
2- Listar alumnos.
3- Mostrar alumnos con notas mayores o iguales a 7.
4- Finalizar programa.

Colaboración de clases

Normalmente un problema resuelto con la metodología de programación orientada a objetos no interviene una sola clase, sino que hay muchas clases que interactúan y se comunican. Plantearemos un problema separando las actividades en dos clases.

Problema 7:

Un banco tiene 3 clientes que pueden hacer depósitos y extracciones. También el banco requiere que al final del día calcule la cantidad de dinero que hay depositado.
Lo primero que hacemos es identificar las clases:
Podemos identificar la clase Cliente y la clase Banco.
Luego debemos definir los atributos y los métodos de cada clase:
Cliente
- atributos
  - nombre
  - monto
- métodos
  - __init__
  - depositar
  - extraer
  - retornar_monto

Banco
- atributos
  - 3 Cliente (3 objetos de la clase Cliente)
- métodos
  - __init__
  - operar
  - depositos_totales

Primero hacemos la declaración de la clase Cliente, en el método __init__ inicializamos los atributos nombre con el valor que llega como parámetro y el atributo monto con el valor cero.

Recordemos que en Python para diferenciar un atributo de una variable local o un parámetro le antecedemos la palabra clave self (es decir nombre es el parámetro y self.nombre es el atributo):

En el bloque principal no se requiere crear objetos de la clase Cliente, esto debido a que los clientes son atributos del Banco.

Problema 8:

Plantear un programa que permita jugar a los dados. Las reglas de juego son:
se tiran tres dados si los tres salen con el mismo valor mostrar un mensaje que "gano", sino "perdió".
Lo primero que hacemos es identificar las clases:
Podemos identificar la clase Dado y la clase JuegoDeDados.
Luego los atributos y los métodos de cada clase:
Dado
- atributos
  - valor
- métodos

tirar
imprimir
retornar_valor

JuegoDeDados
atributos
3 Dado (3 objetos de la clase Dado)
métodos
__init__
jugar

Importamos el módulo "random" de la biblioteca estándar de Python ya que requerimos utilizar la función randint:

```
import random
```

La clase Dado define un método tirar que almacena en el atributo valor un número aleatorio comprendido entre 1 y 6.
Los otros dos métodos de la clase Dado tienen por objetivo mostrar el valor del dado y retornar dicho valor a otra clase que lo requiera.

La clase JuegoDeDados define tres atributos de la clase Dado, en el método __init__ crea dichos objetos.

Acotación

Para cortar una línea en varias líneas en Python podemos encerrar entre paréntesis la condición:

```
if (self.dado1.retornar_valor()==self.dado2.retornar_valor()
    and self.dado1.retornar_valor()==self.dado3.retornar_valor()):
```

O agregar una barra al final:

```
if self.dado1.retornar_valor()==self.dado2.retornar_valor() and \
   self.dado1.retornar_valor()==self.dado3.retornar_valor():
```

Variables de clase

Hemos visto como definimos atributos en una clase anteponiendo la palabra clave self:
class Persona:

```
    def __init__(self,nombre):
        self.nombre=nombre
```

Los atributos son independientes por cada objeto o instancia de la clase, es decir si definimos tres objetos de la clase Persona, todas las personas tienen un atributo nombre pero cada uno tiene un valor independiente.

En algunas situaciones necesitamos almacenar datos que sean compartidos por todos los objetos de dicha clase, en esas situaciones debemos emplear variables de clase.
Para definir una variable de clase lo hacemos dentro de la clase pero fuera de sus métodos:

```
class Persona:

    variable=20

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


# bloque principal

persona1=Persona("Juan")
persona2=Persona("Ana")
persona3=Persona("Luis")

print(persona1.nombre) # Juan
print(persona2.nombre) # Ana
print(persona3.nombre) # Luis

print(persona1.variable) # 20
Persona.variable=5
print(persona2.variable) # 5
```

Se reserva solo un espacio para la variable "variable", independientemente que se definan muchos objetos de la clase Persona. La variable "variable" es compartida por todos los objetos persona1,persona2 y persona3.
Para modificar la variable de clase hacemos referencia al nombre de la clase y seguidamente el nombre de la variable:

```
Persona.variable=5
```

Problema 9:

Definir una clase Cliente que almacene un código de cliente y un nombre.
En la clase Cliente definir una variable de clase de tipo lista que almacene todos los clientes que tienen suspendidas sus cuentas corrientes.
Imprimir por pantalla todos los datos de clientes y el estado que se encuentra su cuenta corriente.

La clase Cliente define una variable de clase llamada suspendidos que es de tipo lista y por ser variable de clase es compartida por todos los objetos que definamos de dicha clase.

En el método imprimir mostramos el código, nombre del cliente y si se encuentra suspendida su cuenta corriente.

El método suspender lo que hace es agregar el código de dicho cliente a la lista de clientes suspendidos.

El método que analiza si está suspendido el cliente verifica si su codigo se encuentra almacenado en la variable de clase suspendidos.

Podemos imprimir la variable de clase suspendidos de la clase Cliente:

```
print(Cliente.suspendidos)
```

Es importante remarcar que todos los objetos acceden a una única lista llamada **suspendidos** gracias a que se definió como variable de clase.

Método especial __str__

Podemos hacer que se ejecute un método definido por nosotros cuando pasamos un objeto a la función print o cuando llamamos a la función str (convertir a string)
¿Qué sucede cuando llamamos a la función print y le pasamos como parámetro un objeto?

```
class Persona:
    def __init__(self,nom,ape):
        self.nombre=nom
        self.apellido=ape

persona1=Persona("Jose","Rodriguez")
print(persona1)
```

Nos muestra algo parecido a esto:

```
<__main__.Persona object at 0x03E99C90>
```

Python nos permite redefinir el método que se debe ejecutar. Esto se hace definiendo en la clase el método especial **__str__**
En el ejemplo anterior si queremos que se muestre el nombre y apellido separados por coma cuando llamemos a la función print el código que debemos implementar es el siguiente:

```
class Persona:
    def __init__(self,nom,ape):
        self.nombre=nom
        self.apellido=ape

    def __str__(self):
        cadena=self.nombre+","+self.apellido
        return cadena
```

```
persona1=Persona("Jose","Rodriguez")
print(persona1)
```

Como vemos debemos implementar el método **__str__** y retornar un **string**, este luego será el que imprime la función print:

```
def __str__(self):
        cadena=self.nombre+","+self.apellido
        return cadena
```

```
class Persona:
    def __init__(self,nom,ape):
        self.nombre=nom
        self.apellido=ape

    def __str__(self):
        cadena=self.nombre+","+self.apellido
        return cadena


# bloque principal

persona1=Persona("Jose","Rodriguez")
persona2=Persona("Ana","Martinez")
print("{} - {}".format(persona1,persona2))
  # Jose,Rodriguez-Ana,Martinez
```

Problema 10:

Definir una clase llamada Punto con dos atributos x e y.
Crearle el método especial __str__ para retornar un string con el formato (x,y).

La clase Punto define dos métodos especiales. El método __init__ donde inicializamos los atributos x e y.
Y el segundo método especial que definimos es el __str__ que debe retornar un string.

Luego en el bloque principal después de definir dos objetos de la clase Punto procedemos a llamar a la función print y le pasamos cada uno de los objetos.

Hay que tener en cuenta que cuando pasamos a la función print el objeto punto1 en ese momento se llama el método especial __str__ que tiene por objetivo retornar un string que nos haga más legible lo que representa dicho objeto.

Problema 11:

Declarar una clase llamada Familia. Definir como atributos el nombre del padre, madre y una lista con los nombres de los hijos.
Definir el método especial __str__ que retorne un string con el nombre del padre, la madre y de todos sus hijos.

Para resolver este problema el método __init__ recibe en forma obligatoria el nombre del padre, madre y en forma opcional una lista con los nombres de los hijos.

Si no tiene hijos la familia, el atributo hijos almacena una lista vacía.
El método especial __str__ genera un string con los nombres del padre, madre y todos los hijos.

Comparación entre objetos

Leer desde el siguiente link: http://elclubdelautodidacta.es/wp/2015/06/comparando-objetos-en-python/

Encapsulación: atributos privados

La encapsulación consiste en denegar el acceso a los atributos y métodos internos de la clase desde el exterior. En Python no existe, pero se puede simular precediendo atributos y métodos con dos barras bajas __ como indicando que son "especiales".

En el caso de los atributos quedarían así:

Código

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

e = Ejemplo()
print(e.__atributo_privado)

Resultado
Y en los métodos...
Código
class Ejemplo:
    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

e = Ejemplo()
e.__metodo_privado()
```

¿Qué sentido tiene esto en Python? Ninguno, porque se pierde toda la gracia de lo que en esencia es el lenguaje: **flexibilidad** y **polimorfismo** sin control (hablaremos de esto más adelante).

Sea como sea para acceder a esos datos se deberían crear métodos públicos que hagan de interfaz. En otros lenguajes les llamaríamos ***getters*** y ***setters*** y es lo que da lugar a las *propiedades*, que no son más que atributos protegidos con interfaces de acceso:

Código:

```
class Ejemplo:
    __atributo_privado = "Soy un atributo inalcanzable desde fuera."

    def __metodo_privado(self):
        print("Soy un método inalcanzable desde fuera.")

    def atributo_publico(self):
        return self.__atributo_privado

    def metodo_publico(self):
        return self.__metodo_privado()

e = Ejemplo()
print(e.atributo_publico())
e.metodo_publico()
```

Getters y Setters en Python

Los **getters** serían las funciones que nos permiten acceder a una variable privada. En Python se declaran creando una función con el **decorador** **@property**.

Los setters serían las funciones que usamos para sobreescribir la información de una variable y se generan definiendo un método con el nombre de la variable sin guiones y utilizando como decorador el nombre de la variable sin guiones más ".setter".

Y para dejarlo claro del todo, vamos a verlo con el siguiente ejemplo:

```
class ListadoBebidas:

    def __init__(self):
        self.__bebida = 'Naranja'
        self.__bebidas_validas = ['Naranja', 'Manzana']

    @property
    def bebida(self):
        return "La bebida oficial es: {}".format(self.__bebida)

    @bebida.setter
    def bebida(self, bebida):
        self.__bebida = bebida

bebidas = ListadoBebidas()
```

```
print(bebidas.bebida)
bebidas.bebida = 'Limonada'
print(bebidas.bebida)
```

En este ejemplo declaramos dos variables, una llamada **_bebida** y una lista llamada **_bebidas_validas**. Para recuperar la información de la variable **_bebida** tendremos que hacerlo con el objeto y el nombre de la función bebida.

Objetos dentro de objetos

Al ser las clases un nuevo tipo de dato se pueden poner en colecciones e incluso utilizarse dentro de otras clases.

```
class Pelicula:

    # Constructor de clase
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento
        print('Se ha creado la película:', self.titulo)

    def __str__(self):
        return '{} ({})'.format(self.titulo, self.lanzamiento)


class Catalogo:

    peliculas = []  # Esta lista contendrá objetos de la clase Pelicula

    def __init__(self, peliculas=[]):
        Catalogo.peliculas = peliculas

    def agregar(self, p):  # p será un objeto Pelicula
        Catalogo.peliculas.append(p)

    def mostrar(self):
        for p in Catalogo.peliculas:
            print(p)  # Print toma por defecto str(p)


p = Pelicula("El Padrino", 175, 1972)
c = Catalogo([p])  # Añado una lista con una película desde el principio
c.mostrar()
c.agregar(Pelicula("El Padrino: Parte 2", 202, 1974))  # Añadimos otra
c.mostrar()
```