Revisión de conceptos
Unidad 1

1. Concepto de POO:

La programación orientada a objetos (POO) es un paradigma de la programación que permite realizar programación modular e incentiva la reutilización de código. Estos son útiles en aplicaciones complejas.

Según la Real Academia Española (2016), paradigma son las "teorías cuyo núcleo central (...) suministra la base y modelo para resolver problemas (…)". En ese sentido, un paradigma de programación es un **esquema** con reglas y conceptos mediante el cual se desarrollan algoritmos y puede facilitar el desarrollo e implementación de estos dependiendo del problema que se quiera abordar.

En programación un objeto se refiere a la manera de abstraer conceptos del mundo real al plano de un programa informático. Los objetos se componen de ciertos atributos que los caracterizan y de acciones (llamados métodos) que pueden realizar. Por ejemplo:

El objeto **caballo**:

Puede tener los siguientes **atributos**:

Color	**Raza**
Peso	**Edad**

Y realizar las siguientes **acciones**:

Trotar	**Alimentarse**
Saltar	

Una característica fundamental sobre POO es el hecho de que permite la interacción entre 2 objetos. Una de las aplicaciones más importantes para esto es el mundo de los videojuegos. Piense, por ejemplo, en un combate entre un *ORCO* y un *ELFO* en un videojuego. En el software dichos personajes se construyen como objetos y el combate entre ellos se modela a partir de una interacción entre objetos.

Otro aspecto importante, es el hecho de que los objetos no necesariamente tienen que estar ligados al concepto cotidiano que tenemos de la palabra **"objeto".** En Python todos los tipos de datos son objetos, por ejemplo: las listas, tuplas, diccionarios, archivos, etc., y estos tienen asociados métodos que producen modificaciones o permiten obtener información. En ese sentido, el concepto objeto se puede extender a cualquier ámbito de la programación.

Algunas de las ventajas de POO son:

- Programación por módulos: permite, por ejemplo, que una aplicación sea desarrollada por diferentes programadores simultáneamente. Esto hace que sea más fácil entender el código.
- Código reutilizable: cuando se requieren instrucciones muy parecidas para ejecutar cierta tarea se pueden usar líneas de código preexistentes mediante la herencia.
- Encapsulamiento: el funcionamiento interno de cada módulo es totalmente independiente de otro.

2. POO en Python:

Para iniciar es necesario definir algunos conceptos:

Clase: es un modelo en el que se definen las características **comunes** de un grupo de objetos. Por ejemplo, si se quiere representar la construcción de carros en un código, la clase estaría definida por el chasis, 4 ruedas, etc., es decir, todas las características que comparten los carros.

Instancia de clase (Objeto): se define como un objeto perteneciente a la clase. Por ejemplo, el auto BMW comparte con los demás carros el chasis, 4 ruedas, etc., pero su marca, color, entre otros aspectos, difiere.

A continuación, se muestra cómo implementar estos conceptos en Python:

```
class Caballo():        # Clase caballo
        Raza="Mustang" # Atributos
        Edad=1
        Color="Marron"  # 4 propiedades.
        Trotar=False

        def Trotar(self):      # Metodos: son funciones que
pertenecen
            Trotar=True        # a una clase, las funciones simples
no                          pertenecen
                               # a ninguna clase, el self hace
referencia al mismo
                               # objeto de la clase.
        def get_Trotar(self):
            if (self.Trotar):
                return("Trotando")   # 2 Comportamientos.
            else:
                return("Quieto")

Caballo1=Caballo()# Se crea el objeto(Se instancia una clase)
print(Caballo1.Raza)
print(Caballo1.Color) # Se accede a la propiedades de objeto
Caballo1.Trotar()     # Se accede a los comportamientos del objeto
print(Caballo1.get_Trotar())
```

```
>>>
Mustang
Marron  # Lo que se obtiene en el Shell
Trotando
```

Ahora, se presenta cómo definir un constructor. Para determinar en cada objeto características específicas de estado inicial, es habitual que se requiera dar diferentes propiedades a cada objeto. Además, se muestra cómo encapsular variables, es decir, estas variables solo podrán ser manipuladas dentro de la clase, no en otra parte del código.

```
class Caballo(): # Clase caballo.
        def __init__(self,raza): # Constructor, va a dar un estado
inicial.

            self.Raza=raza # Atributos, se deben preceder por la
palabra self.
            self.Edad=1
            self.Color="Marron"  # 5 propiedades.
            self.Trotar=False
            self.__numero_de_patas=4 # Esta propiedad deberia
estar encapsulada
                                     # al utilizar 2 guiones bajos,
esta variable
                                     # no se puede utilizar por
fuera de la clase.

Caballo1=Caballo("Mustang")# Se crea el objeto(Se instancia una
clase).
Caballo2=Caballo("Percheron")
print(Caballo1.Raza)
print(Caballo1.Color)
>>>
Mustang
Percheron
```

También se pueden encapsular los métodos, de la siguiente manera:

```
class Ejemplo():        # Clase caballo
        def __init__(self,raza): # Constructor, va a dar un estado
iicial.
          pass # La palabra reservada pass indica que no hay
ninguna instrucción
              # en el método, se usa en estos ejemplos, para no
escribir
```

```
                # código adicional

        def __Encapsulado(self):      # Para encapsularlo, se
precede el nombre
                                      # del método con 2 guiones
bajos
            pass
```

3. Herencia

Para comprender este concepto es necesario pensar en objetos que tengan características en común. Por ejemplo, un guerrero *ORCO* y un guerrero *ELFO* tienen en común atributos como fuerza y armas, al igual que acciones como atacar y defender. La herencia permite crear una clase padre o súper clase que cobija a estos 2 objetos y se construye con todas las características compartidas entre las instancias. Además, con la herencia es posible reutilizar códigos y evitar redundancias en los atributos y métodos utilizados en varios objetos. Es importante resaltar que las características particulares de cada objeto se otorgan en cada clase en específico.

Para crear la clase padre se sigue la siguiente estructura:

```
class Guerrero(): # Se crea padre guerrero
    def __init__(self,fuerza,arma): # constructor de
inicializacion
        self.fuerza=fuerza
        self.arma=arma
        self.atacar=False    # Atributos
        self.defender=False
    def atacar(self):
        self.atacar=True
    def defender(self):      # Metodos
        self.defender=True
    def atributos(self):
        print("fuerza:"+str(self.fuerza))
        print("arma:"+self.arma)

class Orco(Guerrero): # Para que la clase Orco herede
                      # las caracteristicas de guerrro
                      # se debe escribir como parametro
                      # de entrada a la clase guerrero
        def __init__(self,fuerza,arma):# contructor de la clase
orco
            Guerrero.__init__(self,fuerza,arma)# esta linea sirve
para
                                  # inicializar la clase guerrero
            self.bestia="rinocenronte" # se otorga una
caracteristica adicional
        def __str__(self): # el objeto string sire para realizar
casting
```

```
            return('ORCO') # se le asigna al metodo str el nombre
del objeto

class Elfo(Guerrero): # GUERRERO ELFO
    pass # al escribir pass el elfo hereda todo del guerrero pero
         # sin caracteristicas aicionales


Orco1=Orco(5, "hacha")
Orco1.atributos()

Elfo1=Elfo(10,"lanza")
Elfo1.atributos()
>>>
fuerza:5
arma:hacha
fuerza:10
arma:lanza
```

Si en el ejemplo anterior no se hubiera utilizado herencia, el código sería redundante ya que los atributos y métodos en común tendrían que ser especificados en la clase Orco y en la clase Elfo.

Consultar los conceptos: Sobrescribir métodos y Herencia múltiple en Python.

Para iniciar los atributos y métodos de la clase padre en la clase hija, se utiliza el comando super(), de la siguiente manera:

```
class Ciudadano(): # Clase Ciudadano
    def __init__(self,nombre,ID):
        self.nombre=nombre # Atributos del ciudadano
        self.ID=ID
    def informacion(self):
        print('nombre: '+self.nombre)# Imprimir informacion
                                     # del ciudadano
        print('ID: '+str(self.ID))

class Obrero(Ciudadano): # Clase Obrero
    def __init__(self,nombre,ID,cargo,salario):

        super().__init__(nombre,ID) # se inicializa el constructor
                                    # de la superclase ciudadano
        # Utilizar el comando super() es equivalente a escribir:
              # Ciudadano.__init__(self,nombre,ID)
        self.cargo=cargo
        self.salario=salario
    def informacion(self): # para añadir instruciones al metodo
                           # informacion, tambien se usa el
comando
```

```
                                # super()
        super().informacion()
        # Utilizar el comando super() es equivalente a escribir:
            #Ciudadano.informacion(self)
        print("Cargo: "+self.cargo)
        print("Salario: "+str(self.salario))

Paula= Obrero("Paula",123458,"Enfermera",1500)
Paula.informacion()

>>>
nombre: Paula
ID: 123458
Cargo: Enfermera
Salario: 1500
```

Para evitar confusiones al trabajar con herencia, se tiene como guía el principio de sustitución. Este enuncia que el objeto de la subclase es siempre perteneciente a la clase, por ejemplo, en el código anterior se tiene que un obrero es siempre un ciudadano, pero un ciudadano no siempre es un obrero.

Para conocer si un objeto es perteneciente a una clase determinada, se utiliza el comando is instance(), de la siguiente manera:

```
==============================

print(isinstance(Paula, Obrero))
>>>>
True

print(isinstance(Paula, Ciudadano))
>>>>
True

==============================
```

4. Polimorfismo

Su significado es: muchas formas, e indica que un objeto puede cambiar de forma dependiendo de su contexto. Esto implica que las acciones que ejecuta también cambian, por ejemplo, en el código del ciudadano, el obrero podría cambiar a ser estudiante lo cual cambiaría su comportamiento.

Veamos esto, con un ejemplo:

```
class Obrero(): # Clase Obrero
    def Tiempo(Self):
        print("Gasto mi tiempo en la Empresa")

class Estudiante(): # Clase Estudiante
    def Tiempo(Self):
        print("Gasto mi tiempo en la Universidad")
```

```
class Nino(): # Clase Estudiante
    def Tiempo(Self):
        print("Gasto mi tiempo jugando")

Sebastian=Nino()
Sebastian.Tiempo()
# Si el polimorfismo no existira
# se deberia programar de esta manera

Juan=Estudiante()
Juan.Tiempo()

Carolina=Obrero()
Carolina.Tiempo()
>>>
Gasto mi tiempo jugando
Gasto mi tiempo en la Universidad
Gasto mi tiempo en la Empresa

def TiempoPersonas(Persona):
    Persona.Tiempo()# python identifica el objeto y utiliza
                    # su metodo correspondiente

for i in [Sebastian,Juan,Carolina]:
    TiempoPersonas(i) # El polimorfismo transforma la variable
                      # de la funcion en el objeto indicado

>>>
Gasto mi tiempo jugando
Gasto mi tiempo en la Universidad
Gasto mi tiempo en la Empresa
```