Universidad Nacional Mayor de San Marcos
Universidad del Perú, Decana de América
FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

PROGRAMA: ESPECIALIZACION EN PYTHON
MÓDULO BÁSICO

Clase 03

1.
Programación Orientada a Objetos

P.O.O. en Python

Python es un lenguaje de programación orientado a objetos.

La programación orientada a objetos (POO) es un paradigma de programación en el que podemos pensar en problemas complejos como objetos.

Un paradigma nos va a proporcionar la base para resolver problemas.

Al hablar de POO, nos referimos a un conjunto de conceptos y patrones que utilizamos para resolver problemas con objetos.

Cuando creemos una variable se le asigna un valor entero, este valor es un **objeto**, una función es un **objeto**, una cadena de caracteres es un **objeto**, listas, tuplas, diccionarios, etc.

Este tipo de programación introduce un nuevo paradigma que nos va a permitir **encapsular**, **aislar datos** y **operaciones** que se pueden realizar sobres estos mismos datos.

La POO **nos disminuye errores** y promueve la reutilización de código.

2.
Clases y objetos en Python

Definición

Principalmente una clase es una **instancia** donde define elementos que determinan un estado (en este caso datos) y ciertos **comportamientos** (operaciones que podamos hacer sobre los datos que modifican su estado)

Un objeto termina siendo una instancia de una clase.

Para crear una clase anteponemos la palabra reservada *class*

En el presente esquema veremos los elementos principales que van a componer las clases.

En el esquema anterior vimos la definición de la clase *carro*

La cual establece una serie de datos , como el *color, ruedas, aceleración y velocidad* conjuntamente con las operaciones *acelera( )* y *frena ( )*.

En el siguiente ejemplo crearemos un variable de tipo *carro.* A esto se le domina instaciar un objeto de dicha clase.

```
carro1 = Carro("azul", 40)

print("El color de mi primer carro es: ", carro1.color)
```

```
El color de mi primer carro es:  azul
```

Puntos importantes a tener en cuenta:

- En los ejemplos visto **carro1** y **carro2** son objetos cuya clase es *Carro*.

- Ambos objetos creados tienen la propiedad de acelerar y frenar, ya que sus clases definen estas operaciones y tiene un color ya que la clase Carro también está definiendo este dato.

- Nota: es una convención utiliar **CamelCase** para la asignación de nombres de las clases. Esto significa que cada palabra está en mayúsuclas y el resto se mantendrán en minúsculas.

Constructor de una clase

Constructor de una clase

Para poder crear un objeto de una clase específica, o sea, **instanciar una clase**, se escribe el nombre de la clase y luego se añaden paréntisis. (como llamar una función).

```
obj = MiClase( )
```

Así creamos una instancia de la clase **MiClase** y esto crea un objeto vacío, sin estado.

A pesar de esto existen clases (como nuestra clase **Carro**) que necesitan crear instancias de objetos iniciales con un estado inicial.

Esta propiedad se obtiene implementado el método especial ***__init__()***. Este método es nuestro constructor de la clase y siempre se invoca cuando se instancia un nuevo objeto.

Este método establece un parámetro especial ***self*** *(el cual veremos a continuación con más detalle). Se podrá especificar otros parámetros siguiendo las mismas reglas de cualquier función.*

Para nuestro ejemplo el constructor de clase ***Carro*** *es el siguiente:*

```
def __init__(self, color, aceleracion):
    self.color = color
    self.aceleracion = aceleracion
    self.velocidad = 0
```

Se puede observar aparte del parámetro self, que se tienen que definir el ***color*** y ***aceleración***, que determinar el estado inicial de tipo carro.

Deberemos pasar necesariamente los valores para los argumentos de ***color*** y ***aceleración***.

```
carro1 = Carro("azul", 40)
```

IMPORTANTE: En el caso de otros lenguajes, está permitido implentar más de un costructor, en Python sólo está permitido definir un método **__init__()**

Atributos, atributos de datos y métodos

Definición

Ya tenemos entendido qué es un objeto, ahora la única operación que pueden llegar a hacer los objetos es referenciar a sus atributos por medio del operador .

- Tiene objetos de datos y de métodos.
- Los atributos de datos definen el estado del objeto instanciado.
- Los métodos son las funciones que se han definido dentro de la clase.

Atributos de datos

Definición

Los atributos de datos no tienen la necesidad de ser declarados de manera previa. Entonces será posible declarar de la misma manera que creamos las variables en Python, o sea cuando se les declara por primera vez.

Los vemos en el siguiente ejemplo:

```
carro1 = Carro("Negro", 40)
carro2 = Carro("Blanco", 60)

carro2.marchas = 8
print("La candtidad de marchas del segundo carro 2 es: ", carro2.marchas)

print("La candtidad de marchas del segundo carro 2 es: ", carro1.marchas)
```

```
La candtidad de marchas del segundo carro 2 es:  8
Traceback (most recent call last):
  File "/Users/macanthony/Desktop/Dev/Python/projects/basic_module/sesión_06-05/test_10.py", line 27, in <module>
    print("La candtidad de marchas del segundo carro 2 es: ", carro1.marchas)
AttributeError: 'Carro' object has no attribute 'marchas'
```

Herencia en Python

Definición

Cuando hablamos de Herencia en Python, referenciamos a poder reutilizar una clase extendiendo su funcionalidad. Una clase que nosotros podamos crear puede añadir nuevos atributos, añadir nuevos métodos o redefinirlos.

Lo vemos en el siguiente ejemplo:

```python
class CarroVolador(Carro):

    ruedas = 6

    def __init__(self, color, aceleracion, estado_volando=False):
        super().__init__(color, aceleracion)
        self.estado_volando = estado_volando

    def vuela(self):
        self.estado_volando = True

    def aterriza(self):
        self.estado_volando = False
```

Cuando hablamos de Herencia en Python, referenciamos a poder reutilizar una clase extendiendo su funcionalidad. Una clase que nosotros podamos crear puede añadir nuevos atributos, añadir nuevos métodos o redefinirlos.

```
carro1 = Carro("Azul", 40)
carroVolador1 = CarroVolador("Rojo", 60)

print(carroVolador1.color)
print(carroVolador1.estado_volando)

carroVolador1.acelera()
print("La velocidad de mi primer carro volador es: ", carroVolador1.velocidad)

print("La cantidad de ruedas de mi carro volador es: ", carroVolador1.ruedas)

print("Estado de vuelo: ", carro1.estado_volando)
```

```
Traceback (most recent call last):
  File "D:\Developer\Python\Clase Python\sesion 03\sesion 03 05\test_4.py", line 53, in <module>
    print("Estado de vuelo: ", carro1.estado_volando)
AttributeError: 'Carro' object has no attribute 'estado_volando'
Rojo
False
La velocidad de mi primer carro volador es:  60
La cantidad de ruedas de mi carro volador es:  6
```

Encapsulamiento

Definición

En POO la encapsulación se refiere a la capacidad que tiene un objeto para ocultar su estado, la única manera que sus datos se puedan modificar pero sólo por medio de sus operaciones, **métodos**.

```
class A:
    def __init__(self):
        self._contador = 0  # Este atributo es privado
    def incrementa(self):
        self._contador += 1
    def cuenta(self):
        return self._contador
class B(object):
    def __init__(self):
        self.__contador = 0  # Este atributo es privado
    def incrementa(self):
        self.__contador += 1
    def cuenta(self):
        return self.__contador
```

Polimorfismo

Definición

Podemos entender por Polimorfismo a la capacidad de poder referenciar en tiempo de ejecución a instancias de diferentes Clases

```python
class Perro:
    def sonido(self):
        print("Guauu!!!")

class Gato:
    def sonido(self):
        print("Miauuuu!!!")

class Vaca:
    def sonido(self):
        print("Muuuu!!")


def cantar(animales):
    for animal in animales:
        animal.sonido()
```

3.
Manejo de excepciones

1. Error de sintaxis

Los errores de sintaxis, también conocidos como errores de interpretación, son quizás el tipo de queja más común que tenemos cuando todavía estamos aprendiendo Python.

```
>>> while True print('Hola Pythonistas')
  File "<stdin>", line 1
    while True print('Hola Pythonistas')
                     ^
SyntaxError: invalid syntax
```

2. Excepciones

Los errores que se detectan durante la ejecución se llaman ***excepciones***, los cuales no son incondicionalmente fatales y las cuales aprenderemos a gestionarlas en Python. La mayoría de errores no puede ser gestionados por el código.

```
>>> 5 / (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

La última línea de error nos indicará qué ha sucedido exactamente, como las que se ven en el ejemplo como: ZeroDivisionError y TypeError. La cadena mostrada como tipo de la excepción es el nombre de la excepción predefinida que ha ocurrido.

El resto de la línea provee información basado en el tipo de la excepción y qué la causó.

3. Gestión de excepciones

Python nos posibilita manejar o gestionar excepciones. En el siguiente ejemplo se le pedirá al usuario escribir un número entero hasta que introduzca solamente un dato de tipo entero.

```
while True:
    try:
        x = int(input("Por favor ingresar un número: "))
        break
    except ValueError:
        print("Oops!  Este es un número, intentar de nuevo...")
```

Una *cláusula except* puede nombrar múltiples excepciones como una tupla entre paréntesis, por ejemplo:

```
try:
    pass
except (RuntimeError, TypeError, NameError):
    pass
```

Dado que dentro de un mismo bloque **try** pueden producirse excepciones de distintos tipos, es posible utilizar varios bloques ***except***, cada uno para capturar un tipo distinto de excepción.

```
1
2   try:
3       """aquí ponemos el código que puede lanzar excepciones"""
4   except IOError:
5       """entrará aquí en caso que se haya producido"""
6       """una excepción IOError"""
7   except ZeroDivisionError:
8       """entrará aquí en caso que se haya producido"""
9       """una excepción ZeroDivisionError"""
10  except:
11      """entrará aquí en caso que se haya producido"""
12      """una excepción que no corresponda a ninguno"""
13      """de los tipos especificados en los except previos"""
14
```

4.
Módulos y librerías

Cuando creamos un programa en Python tendremos que recurrir a módulos, librerías o paquetes para no tener que repetir código o tener que reinventar nuevamente soluciones ya existentes.

También nos ayuda a poder organizar nuestro programa cuando es demasiado extenso o contiene demasiadas líneas de código, esta estructura puede ser separada en **módulos**.

- **Módulo:** Porción de programa.

- Codificadas en un lenguaje de programación.

- Ofrece un interfaz bien hecha y definida para alguna funcionalidad que se invoca.

➢ **Librería (o biblioteca o dependencia):**

- Conjunto de implementaciones funcionales que serán rehusadas.
- Conjunto de módulos, esta agrupación tendrá una afinidad específica.

Ventajas de trabajar con módulos y librerías:

- Evita realizar miles de líneas de código para tu programa en un solo archivo.
- Encontrarás los errores de tu programa de una manera más legible.
- Existen librerías que nos evita escribir código innecesario para ciertos problemas para nuestros programas.

Importando módulos

Para importar un módulo en Python usamos la palabra reservada **import.**

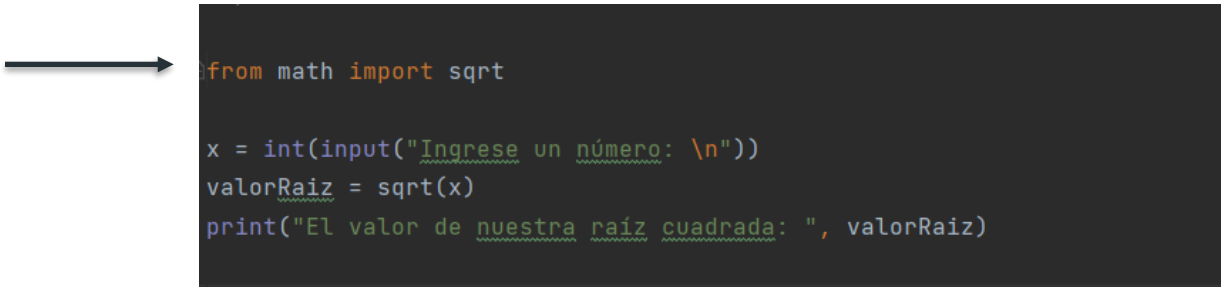
Como podemos ver en el siguiente ejemplo:

```python
import math

x = int(input("Ingrese un número: \n"))
valorRaiz = math.sqrt(x)
print("El valor de nuestra raíz cuadrada: ", valorRaiz)
```

Pero existe un problema al importar todo un módulo y es la sobre carga innecesaria de memoria ya que traemos todas los métodos del módulo al mismo tiempo.

Sólo sería necesario utilizar la función **sqrt**



```
from math import sqrt

x = int(input("Ingrese un número: \n"))
valorRaiz = sqrt(x)
print("El valor de nuestra raíz cuadrada: ", valorRaiz)
```

De este modo sólo importará el **método** que queremos usar en nuestro programa.

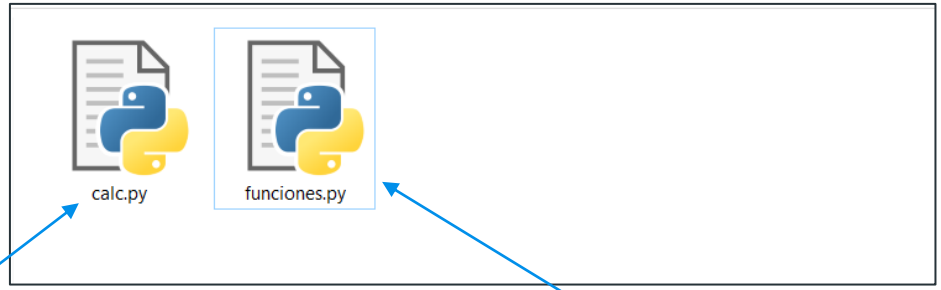
Ya que resulta riesgoso traer varios módulos, ya que no podremos saber si estos módulos puedan tener nombres de clases y métodos exactamente iguales.

* Para saber cuáles son todas las funciones que tiene nuestro módulo usamos la función de Python **dir()**

Creando un módulo propio

Lo primero que debemos es analizar como vamos a dividir nuestro **proyecto**.

Para el caso de una calculadora, las funciones de las operaciones matemáticas estarán en un file llamado ***funciones.py*** *y el ejecutable principal estará dentro de un file llamado* **calc.py**



Programa principal.

Módulo a ser importado en el programa principal.

Uso de Date en python

➢ Librería Datatime

Para manejar fechas en Python se suele utilizar la librería datetime que incorpora los tipos de datos ***date***, ***time*** y ***datetime*** para representar **fechas y funciones** para manejarlas.

Algunas de las operaciones más habituales que permite son:

- Acceder a los distintos componentes de una fecha (año, mes, día, hora, minutos, segundos y microsegundos).
- Convertir cadenas con formato de fecha en los tipos **date**, **time** o **datetime**.
- Convertir fechas de los **tipos date, time o datetime** en cadenas formateadas de acuerdo a diferentes formatos de fechas.
- Hacer aritmética de fechas (sumar o restar fechas).
- Comparar fechas.

Uso de la librería JSON en python

➢ Librería JSON

Para trabajar con datos JSON en Python definiremos brevemente lo que se entiende por JSON respecto a su página oficial:

” *JSON, es el acrónimo de* ***JavaScript Object Notation****, es un formato de texto ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript aunque hoy, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente.*

Es utilizado para proyectos de lenguajes de programación diferentes como C, C++, Java, Javascript, Perl, Python y muchos más. Estas propiedades hacen que JSON sea el lenguaje ideal para el intercambio de datos.”

Como conclusión, JSON es una forma de almacenar e intercambiar datos.

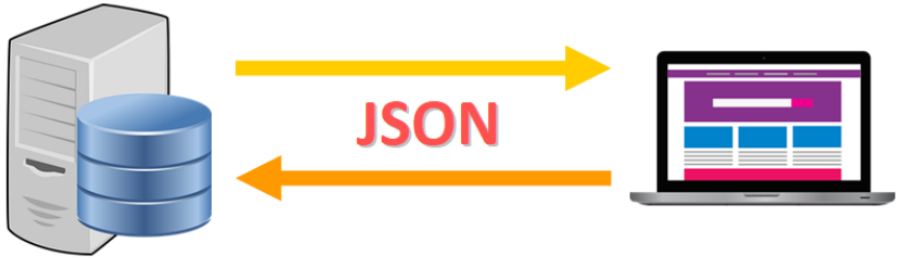
Lo bueno de JSON es que tiene un formato muy legible para el humano, y es una de las grandes razones de su popularidad, además de su eficacia podemos trabajar con **APIs** (abreviatura de **Application Programming Interfaces**, en español significa interfaz de programación de aplicaciones. Se basa en un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos de ellas.

Un ejemplo de datos en formato JSON lo vemos a continuación:

Python hace que sea sencillo trabajar con archivos JSON. El módulo utilizado para este propósito es el módulo **json**. Este módulo debe ser incluido (built-in) dentro de tu instalación con Python y, por lo tanto, no será necesario instalar módulos externos.

Lo único que necesitas hacer con el fin de utilizar este módulo es importarlo.

• **Desarrollo software:** JSON se usa comúnmente en aplicaciones web o móvil para enviar información desde el servidor al cliente o desde el cliente al servidor.



Ejemplo:

Ahora ya sabiendo como se usa el formato JSON, veremos un poco su estructura

```
{
   "empresa":"starbucks",
   "pais":"Peru",
   "inicio":2005,
   "empleados":[
      {
         "nombre":"Ariana M.",
         "edad":24,
         "habilidades":[
            "responsable",
            "comunicacion efectiva"
         ]
      },
      {
         "nombre":"Angie H.",
         "edad":26,
         "habilidades":[
            "responsable",
            “puntualidad"
         ]
      } ]
}
```

JSON vs. diccionarios de Python

JSON y los diccionarios parecen muy similares inicialmente (visualmente) pero en realidad son muy diferentes. Veamos cómo están relacionados y cómo se complementan para lograr que Python sea una herramienta poderosa para trabajar con archivos JSON.

JSON es un formato de archivo usado para representar y almacenar información mientras que un diccionario de Python es una estructura de datos (**objeto**) que se almacena en la memoria del dispositivo mientras se ejecuta el programa de Python.

IMPORTANTE:

Al momento de trabajar archivos JSON en Python, no es simplemente leerlos y usar la información en nuestro programa creado directamente.

Esto se debe a que el archivo completo estaría representado como una sola cadena de caracteres y no podríamos acceder a los pares clave-valor de forma individual.

A menos que suceda lo siguiente:

Usemos los pares **clave-valor** del archivo JSON **para crear un diccionario de Python** que podamos usar en nuestro programa para leer, usar y modificar la información si es necesario.

Esta es la relación principal entre **JSON** y los **diccionarios de Python**.
JSON es la representación en forma de cadena de caracteres de la información y los diccionarios son las estructuras de datos **en memoria** que se crean cuando se ejecuta el programa.

➢ De JSON a Python

Poder leer JSON significa convertir el valor tipo JSON en un valor de Python (o sea un objeto).

Como hemos mencionado anteriormente, la librería **json** parsea el dato JSON en un **diccionario** o en una **lista** en Python.

Para realizar este procedimiento, utilizamos el método **loads(),** de la siguiente manera:

```
import json

jsonData = '{"nombre":'Python', "tipo":'backend' , "paradigma":'Orientado a Objetos'}'

dictionaryToJson = json.loads(jsonData)
```

➢ De Python a JSON

Hemos visto como convertir un valor JSON a Python (o sea, en un diccionario), gracias a la librería JSON, ahora también podemos convertir (codificar) de un valor Python a JSON.

El método de la librería que nos ayuda es **dumps()**

```
import json


pythonDictionary = {‘nombre’:’Python', ‘tipo’:’backend’ , ‘paradigma’:’Orientado a Objetos’}

dictionaryToJson = json.dumps(pythonDictionary)
```

Importante: JSON no puede almacenar todo tipo de datos de Python, solamente los siguiente tipos: **Lists, Dictionaries, Booleanas, Numbers, Character Strings y None**.

Universidad Nacional Mayor de San Marcos