**Resumen Unidad 3**

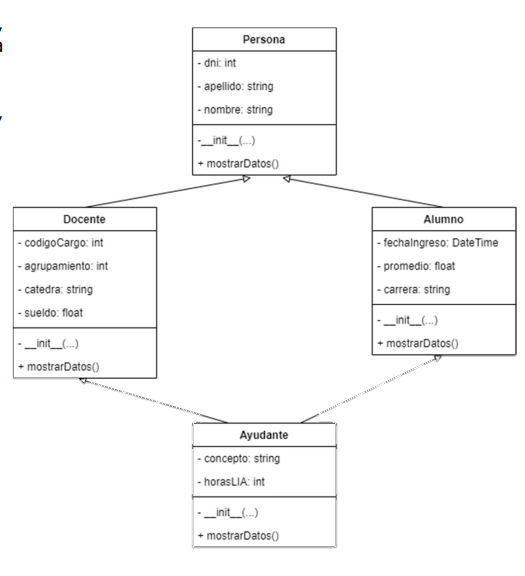
**Herencia**

La herencia es el mecanismo que permite compartir automáticamente métodos y datos entre clases y subclases. Este mecanismo potente, permite crear nuevas clases a partir de clases existentes programando solamente diferencias

Todas las clases en Python, derivan de **object**, por lo que disponen de los atributos y métodos de dicha clase, es decir **object**, es la clase base o superclase de todas las clases de Python, y las definidas por el programador

**Herencia Múltiple**

Una subclase deriva de más de una clase base.



Un problema que se presenta es cuando la subclase que hereda de más de una clase, recibe de las clases bases un mismo nombre de método.

¿Cuándo desde la subclase se invoca el método, en qué orden se ejecutan ambos? Python provee un mecanismo de resolución de conflictos, denominado MRO (Method Resolution Order).

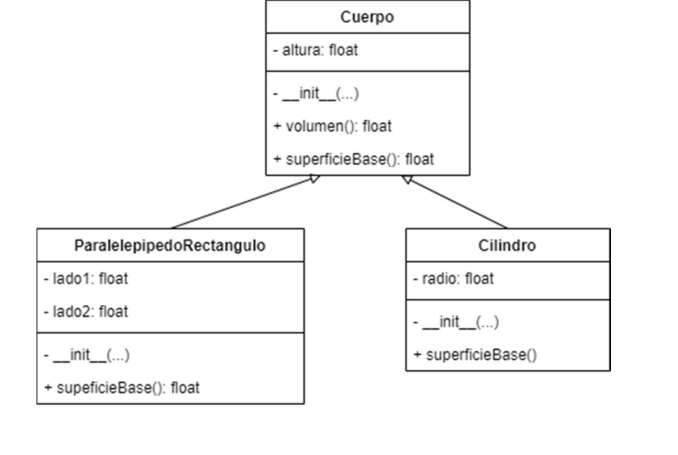
El MRO aplica también al orden de inicialización de las clases bases cuando se hace usando el método \_\_init\_\_ desde la función super(). MRO para la ejecución de métodos recorre el árbol de herencia de arriba hacia abajo, y al mismo nivel, de derecha a izquierda (denominada regla del diamante)

**Polimorfismo**

El polimorfismo es la capacidad que tienen objetos de clases diferentes, a responder de forma distinta a una misma llamada de un método.

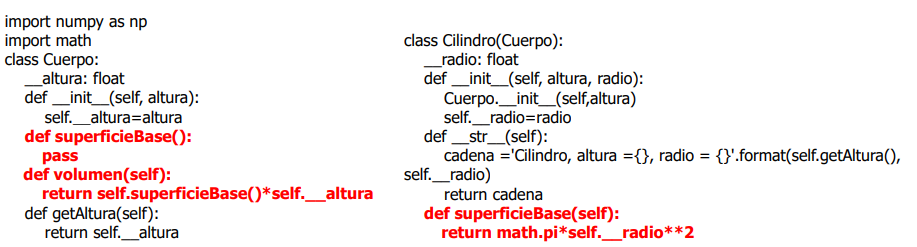
El polimorfismo hace que el código sea flexible y por lo tanto reusable.

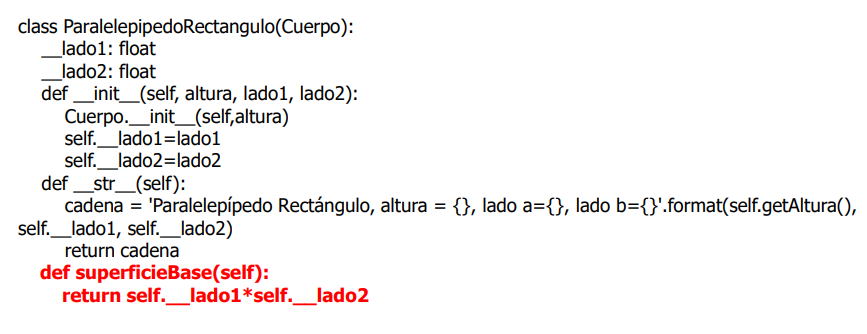
El polimorfismo de subtipo se basa en la ligadura dinámica y la herencia.



La clase Cuerpo, posee datos insuficientes para el cálculo de la superficie de la base, por lo que la función superficieBase(), no tendrá cuerpo (en Python se utiliza la palabra reservada pass, para indicarlo)

La función superficieBase () sin cuerpo, es un método candidato a ser abstracto, lo que haría que la clase Cuerpo sea una Clase Abstracta





**La clase de los objetos referenciados**

Para determinar a qué clase pertenece un objeto pueden utilizarse las funciones:

* **isinstance(x, Clase)**, donde x es una referencia a un objeto, Clase es el nombre de la clase de la que se quiere averiguar si un objeto es instancia o no, la función devuelve True o False, dependiendo si x es un objeto perteneciente a la clase o no.
* **type(x)**, donde x es una referencia a un objeto, devuelve la clase a la que pertenece dicho objeto.

**Errores en un programa**

Tales errores se clasifican en tres tipos básicos:

* **Errores de sintaxis:** son errores donde el código no es válido para el compilador o intérprete, generalmente son fáciles de corregir.
* **Errores en tiempo de ejecución:** son errores donde un código sintácticamente válido falla, quizá debido a una entrada no válida, generalmente son fáciles de corregir.
* **Errores semánticos:** errores en la lógica del programa, el código se ejecuta sin problemas, pero el resultado no es el esperado, a veces muy difíciles de seguir y corregir

**Excepciones**

Las excepciones son errores que ocurren cuando se ejecuta un programa, errores en tiempo de ejecución.

Cuando se produce este error, el programa se detendrá y generará una excepción que luego se manejará para evitar que el programa se detenga por completo.

Python provee un manejo muy completo de las excepciones, que incluye las instrucciones:

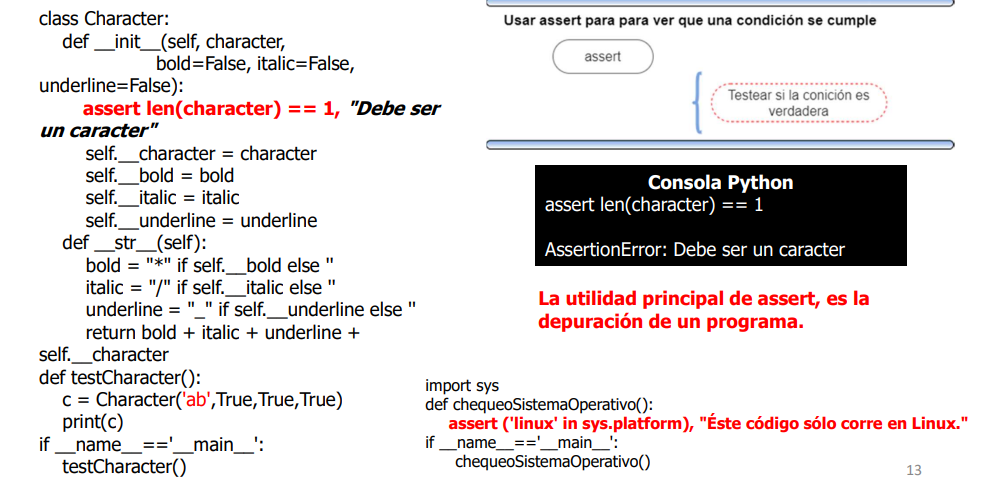
* **assert:** para probar si una afirmación es verdadera o falsa.
* **raise:** para forzar el lanzamiento de una excepción.
* **try-except-else-finally:** bloque para manejar y capturar excepciones, y determinar qué hace cuando sea capturada una excepción.

**Assert**

En lugar de esperar a que el programa se detenga por un error, el programador puede iniciar una afirmación de excepción con la sentencia assert.

Si lo que se afirma se cumple, es verdadero, la ejecución continúa sin problemas.

Si lo que se afirma no se cumple, es falso, la ejecución se interrumpe y se la lanza la excepción AssertionError

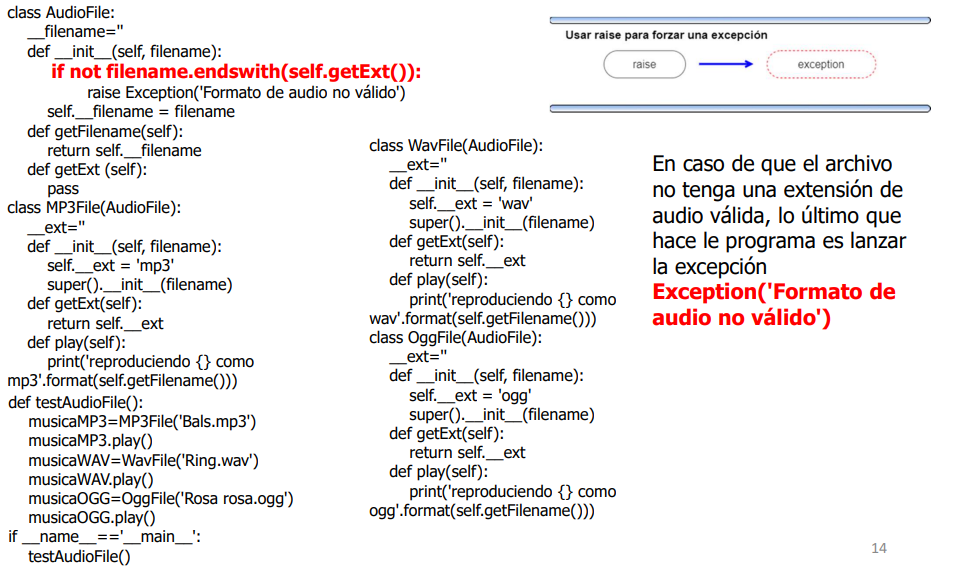


**Raise**

Python lanza excepciones en forma automática.

El programador puede necesitar lanzar una excepción.

Lanzar en forma manual una excepción se hace con la instrucción raise

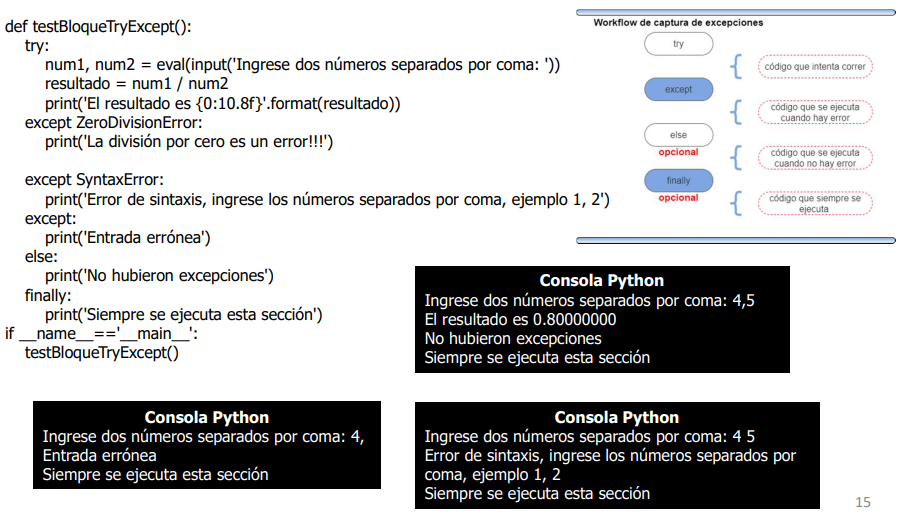


**Bloque try-except-else-finally**

El bloque try-except-else-finally, se utiliza para capturar y manejar excepciones.

Los bloques else y finally son opcionales.

El bloque try-except puede controlar más de una excepción, por lo que el sub bloque except puede repetirse tantas veces como excepciones se quieran capturar y manejar



**Listas definidas por el programador**

**Iterador sobre la clase lista**

Un iterador es un objeto adherido al iterator protocol de Python

Esto significa que tiene una función next(), es decir, cuando se le llama, devuelve el siguiente elemento en la secuencia, cuando no queda nada para ser devuelto, lanza la excepción StopIteration, lo que causa que la iteración se detenga.

Para que la clase Lista definida anteriormente, o cualquier otra clase definida por el programador, sea iterable, la clase debe proveer los métodos:

**\_\_iter\_\_(self)**, que devuelve un iterador de Python, en general se deja que devuelva el iterador de object

**\_\_next\_\_(self)**, que devuelve el siguiente elemento de la secuencia, y cuando no hay más elementos lanza la excepción StopIteration

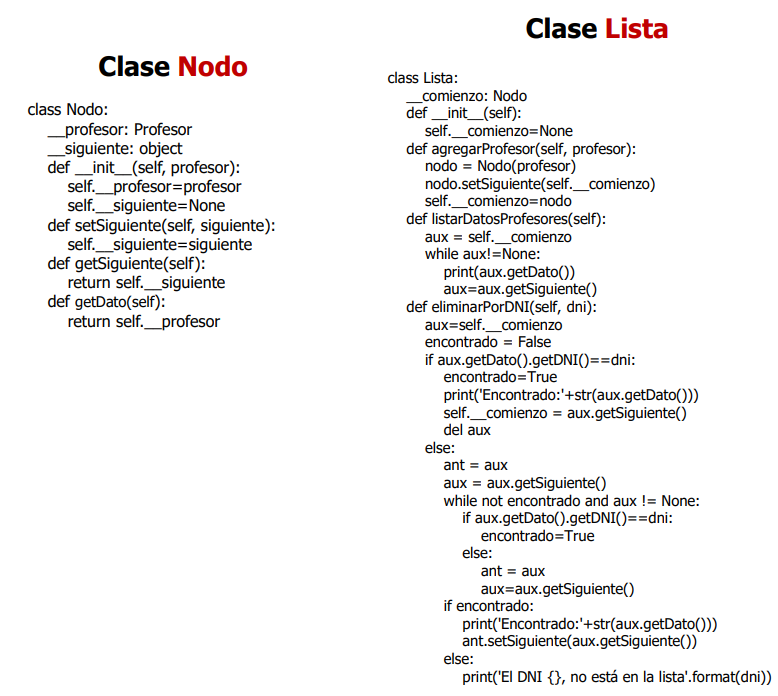
En el caso de la clase Lista, se deberán agregar tres nuevos atributos:

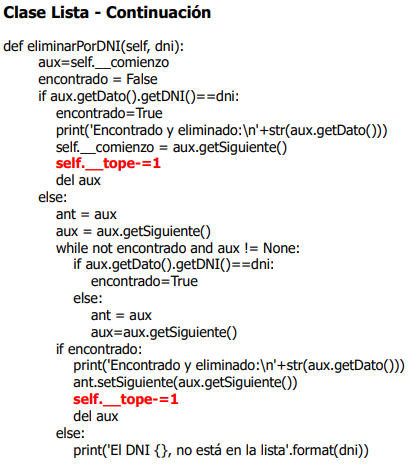
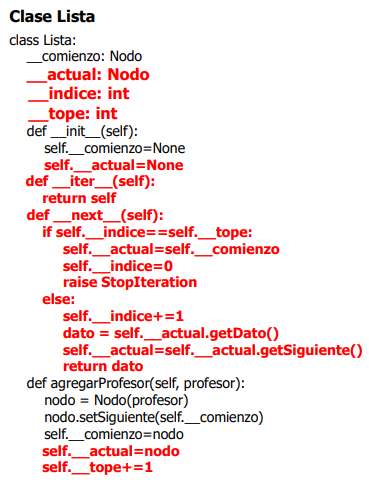
**\_\_actual:** para saber cuál es el elemento actual, y poder devolverlo al iterar.

**\_\_index**: lleva la cuenta de los pasos de iteración, se actualiza en uno para pasar el siguiente elemento.

**\_\_tope:** lleva la cuenta total de elementos de la lista, se actualiza sumando uno cuando se agregan

elementos a la lista, y se decrementa cuando se eliminan elementos de la lista





**Testing**

Probar el código es asegurarse de que funciona.

A medida que crece un programa, los diversos caminos que el intérprete puede tomar, crecen con el código, y rápidamente se vuelve imposible probarlos a todos manualmente.

Para manejar esto, se escriben pruebas automatizadas. Estos son programas que automáticamente ejecutan ciertas entradas (pruebas). Se pueden correr estos programas de prueba en segundos y cubren más situaciones de entrada posibles que las que un programador pensaría en probar cada vez que cambia algo.

**Cuatro razones por las que es necesario hacer pruebas de software:**

* Para asegurarse de que el código funcione de la manera que el desarrollador cree que debería.
* Para garantizar que el código siga funcionando cuando se realizan cambios.
* Asegurarse de que el desarrollador entendió los requisitos.
* Para asegurarse que el código que se escribió tenga una interfaz que se pueda mantener.

**Desarrollo basado en pruebas**

La metodología basada en pruebas tiene dos objetivos:

* El primero es asegurarse de que las pruebas realmente se escriben. Si la prueba ya está escrita antes de escribir el código, se sabrá exactamente cuándo funciona (porque la prueba pasará), y se sabrá en el futuro si alguna vez se rompe por un cambio que se haya hecho sobre el código.
* En segundo lugar, escribir pruebas primero obliga al programador a considerar exactamente cómo será el código. Ayuda a dividir el problema inicial en problemas más pequeños y comprobables, y luego recombinar las soluciones probadas en soluciones más grandes, también probadas

**Test de unidad**

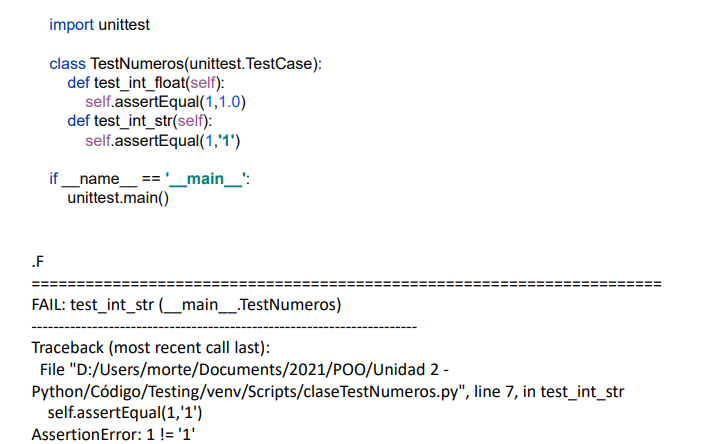
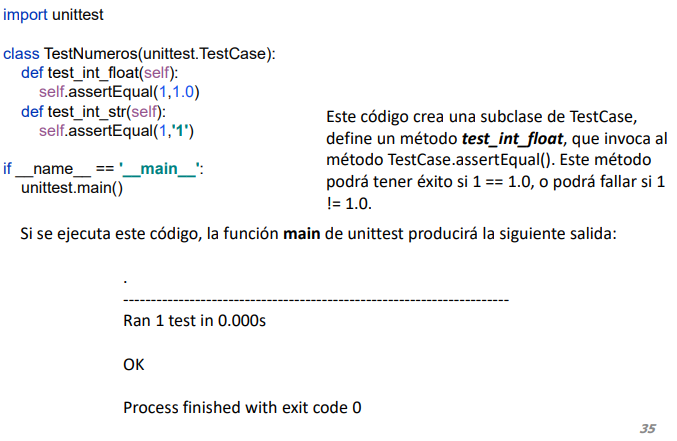
Python provee la biblioteca **unittest** incorporada al core. Esta biblioteca proporciona una interfaz común para pruebas unitarias.

Las pruebas unitarias se enfocan en probar la menor cantidad de código posible en cualquier prueba.

Esta biblioteca proporciona varias herramientas para crear y ejecutar pruebas unitarias, siendo la más importante la clase TestCase. TestCase proporciona un conjunto de métodos que permiten comparar valores, configurar pruebas y limpiar la memoria cuando ya se hayan terminado.

Cuando se escribe un conjunto de pruebas unitarias para una tarea específica, se crea una subclase de TestCase y se escriben métodos individuales para realizar la prueba real.

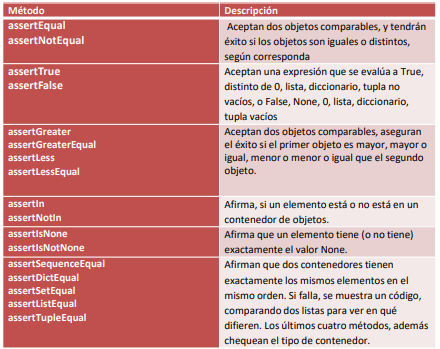
Estos métodos todos DEBEN comenzar con el nombre de test (se ejecutarán automáticamente).



**Métodos Assert**

Las afirmaciones son declaraciones que se pueden hacer dentro del código, mientras se está desarrollando, las afirmaciones, se pueden usar para probar la validez del código, si la declaración no resulta ser cierta, se genera un AssertionError y el programa se detendrá.

El diseño general de un caso de prueba es establecer ciertas variables o atributos, en valores conocidos, y ejecutar una o más funciones, métodos o procesos, y luego "probar" el valor esperado con los resultados que se devolvieron o calcularon



**setUp y tearDown**

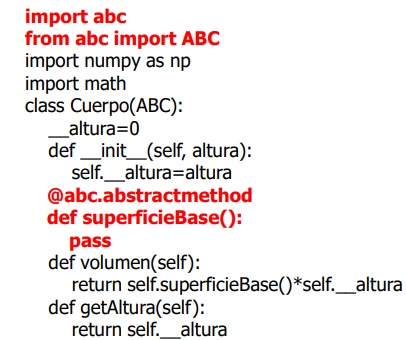
el método **setUp()** de la clase TestCase sirve para realizar la inicialización de cada prueba. Este método no acepta argumentos y permite realizar una configuración arbitraria antes de ejecutar cada prueba.

El método **tearDown()**, se utiliza para liberar recursos que se ocuparon durante las pruebas.

**Clase base abstracta**

Una clase abstracta:

* Permite construir una interfaz común a un conjunto de subclases. Se construye para reunir un conjunto de atributos comunes al conjunto de subclases.
* Es aquella que define una interfaz, pero no su implementación, de tal forma que sus subclases sobrescriban los métodos con las implementaciones correspondientes.
* Una clase abstracta no puede ser instanciada.



**Interfaces**

Una interfaz es un conjunto de firmas de métodos, atributos o propiedades eventos agrupados bajo un nombre común. Funcionan como un conjunto de funcionalidades que luego implementan las clases.

Las clases que implementan las funcionalidades quedan vinculadas por la o las interfaces que implementan.

Son parecidas a las clases abstractas, en el sentido en que no proveen implementación de los métodos que declaran.

Se diferencian en que las clases que derivan de clases abstractas pueden no implementarlos métodos abstractos (subclase abstracta), mientras que las clases que implementen una interfaz deben proveer implementación de todos los métodos de la interfaz.

Al igual que las clases abstractas, son tipo referencia, pero no pueden crearse objetos directamente de ellas (solo de las clases que las implementen)

**¿Cuándo definir interfaces?**

* Siempre que se prevea que dos o más clases no vinculadas por la herencia, harán lo mismo, es conveniente definir una interfaz con los comportamientos comunes.
* Pensar en la interfaz antes que, en la clase en sí, es pensar en lo qué debe hacerse en lugar de pensar en cómo debe hacerse.
* Usar interfaces permite a posteriori cambiar una clase por otra que implemente la misma interfaz y poder integrar la nueva clase de forma mucho más fácil, sólo se debe modificar donde se instancian las clases, el resto del código queda igual, fomentando la reusabilidad del código

**Diccionarios**

Los diccionarios en Python, son estructuras de datos utilizadas para mapear claves a valores. Los valores pueden ser de cualquier tipo, inclusive otro diccionario, una lista, un arreglo.

Por ejemplo, dados los valores (Arnold, 55, Masculino, Si, No, 4, 33876), se podrán mapear a través delas claves (nombre, edad, genero, hijos, casado, cantidad de hijos, sueldo).

Para crearlo, se declara el identificador, que será el nombre del diccionario, seguido de un signo igual yente llaves los pares Clave:Valor separados por comas:

diccionario={'nombre':'Arnold', 'edad':55,'genero':'masculino','hijos':'Si', 'casado':'No', 'cantidad de hijos':4, 'sueldo':33876}

O su equivalente usando la función **dict()**:

diccionario=**dict**({'nombre':'Arnold', 'edad':55,'genero':'masculino','hijos':'Si','casado':'No', 'cantidad de hijos':4, 'sueldo':33876})

print('Nombre: ', diccionario['nombre'])

print('Edad: ‘, diccionario['edad'])

**Archivos JSON**

JSON (JavaScript Object Notation) es un formato de intercambio de datos basado en texto, se usa para el intercambio de datos en la web.

JSON puede ser codificado y decodificado con Python, se utiliza la librería JSON de Python.

Para escribir los datos en un archivo JSON se utiliza la función dump() o dumps()

Para leer los datos se utiliza la función load() o loads()

Como se debe proveer un método para codificar los valores de las clases, dicha representación será un diccionario, donde la clave será el nombre del parámetro formal que de la función \_\_init\_\_, y el valor, representará el estado del objeto al momento de ejecutar la función dump().

Los datos en formato JSON pueden almacenarse en archivos, para luego recuperarlos y almacenar los datos en los atributos de las clases.

De lo expuesto, los archivos JSON, constituyen un mecanismo de persistencia del estado de los objetos.

