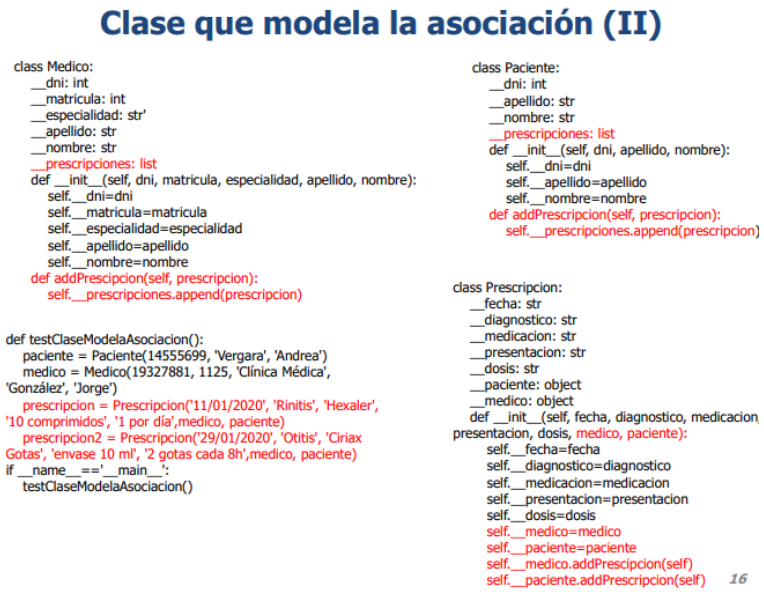
Unidad 3 Poo - Relaciones entre clases en el lenguaje Python

Algunas clases pueden existir aisladas pero la mayoría no pueden y deben cooperar entre sí. Estas relaciones se expresan de una forma de acoplamiento entre ellas

Según el tipo podemos distinguir entre:

- Asociación: la asociación en un diagrama de clases implica transitividad y bidirección de clases. La cardinalidad de la asociación indicará si hace falta una colección (un manejador, un gestor) para almacenar los objetos,podría ser una lista Python, un arreglo Numpy, una lista definida por el programador.

  Una instancia de clase asociación siempre se relaciona a una única instancia de la clase en un extremo y a una única instancia de la clase en el otro extremo. No importa la multiplicidad en ambos extremos. Una instancia de la clase de asociación representa una relación uno a uno.
  - Referencia circular:Una referencia circular en Python ocurre cuando dos o más objetos se refieren entre sí en una forma que crea un ciclo.
  - Clase que modela la asociación:



- Agregación:
  - Un objeto de una clase contiene como partes a objetos de otras clases
  - La destrucción del objeto continente no implica la destrucción de sus partes.
  - Los tiempos de vida de los objetos continente y contenido no están acoplados, de modo que se pueden crear y destruir instancias de cada clase independientemente.
- Composición:
  - Un objeto de una clase contiene como partes a objetos de otras clases y estas partes están físicamente contenidas por el agregado.
  - Los tiempos de vida de los objetos continente y contenido están estrechamente acoplados
  - La destrucción del objeto continente implica la destrucción de sus partes.

- 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.
    - Con la función **super()**, se accede a **atributos**, y métodos de la clase base,también es posible realizar la llamada al constructor invocando: Circulo.__init__(self, radio)
    - 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.
      Python provee 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 de subtipo se basa en la ligadura dinámica y la herencia.
- El polimorfismo hace que el código sea flexible y por lo tanto reusable
- Python es un lenguaje con tipado dinámico, la vinculación de un objeto con un método también es dinámica.
- Todos los métodos se vinculan a las instancias en tiempo de ejecución.
- Para llevar este tipo de vinculación, se utiliza una tabla de métodos virtuales por cada clase

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 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 el tiempo de ejecución.

Cuando se produce este error, el programa se detendrá y generará una excepción que luego se maneja 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.
    - 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 lanza la excepción **AssertionError**.
- **raise:** para forzar el lanzamiento de una excepción.
- **try-except-else-finally:** bloque para manejar y capturar excepciones, y determinar qué hacer cuando sea capturada una excepción.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.

Consejos:

- No todas las excepciones se crean de la misma manera: si sabe con qué clase de excepción está tratando, sea específico sobre lo que captura.

- No atrape nada con lo que no pueda tratar.

- Si necesita tratar con múltiples tipos de excepciones, entonces tenga múltiples bloques except en el orden correcto

- Las excepciones personalizadas pueden ser muy útiles si se necesita almacenar información compleja o específica en instancias de excepción

- No cree nuevas clases de excepción cuando las integradas tengan toda la funcionalidad que necesita.

- No necesita lanzar una excepción tan pronto como se construye, esto facilita la predefinición de un conjunto de errores permitidos en un solo lugar.

LISTAS DEFINIDAS POR EL PROGRAMADOR:

Clase Nodo

```
class Nodo:
    __profesor: Profesor
    __siguiente: object
    def __init__(self, profesor):
        self.__profesor=profesor
        self.__siguiente=None
    def setSiguiente(self, siguiente):
        self.__siguiente=siguiente
    def getSiguiente(self):
        return self.__siguiente
    def getDato(self):
        return self.__profesor
```

Clase Lista

```
class Lista:
    __comienzo: Nodo
    def __init__(self):
        self.__comienzo=None
    def agregarProfesor(self, profesor):
        nodo = Nodo(profesor)
        nodo.setSiguiente(self.__comienzo)
        self.__comienzo=nodo
    def listarDatosProfesores(self):
        aux = self.__comienzo
        while aux!=None:
            print(aux.getDato())
            aux=aux.getSiguiente()
    def eliminarPorDNI(self, dni):
        aux=self.__comienzo
        encontrado = False
        if aux.getDato().getDNI()==dni:
            encontrado=True
            print('Encontrado:'+str(aux.getDato()))
            self.__comienzo = aux.getSiguiente()
            del aux
        else:
            ant = aux
            aux = aux.getSiguiente()
            while not encontrado and aux != None:
                if aux.getDato().getDNI()==dni:
                    encontrado=True
                else:
                    ant = aux
                    aux=aux.getSiguiente()
            if encontrado:
                print('Encontrado:'+str(aux.getDato()))
                ant.setSiguiente(aux.getSiguiente())
            else:
                print('El DNI {}, no está en la lista'.format(dni))
```

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 cual 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 :

Un testing efectivo puede ayudar a identificar problemas en etapas tempranas del desarrollo, lo que conduce a la reducción de costos relacionados con correcciones tardías y retrasos en el lanzamiento de productos.

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.

La metodología basada en pruebas tiene dos objetivos:

• El primero es asegurarse de que las pruebas realmente se escriben. Es muy fácil, después de haber escrito el código. 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. Dice qué métodos deben tener los objetos y cómo los atributos serán accedidos. Ayuda a dividir el problema inicial en problemas más pequeños y comprobables, y luego recombinar las soluciones probadas en soluciones más grandes, también probadas.

Python provee la biblioteca unittest, proporciona una interfaz común para pruebas unitarias. Las pruebas unitarias se enfocan en probar la menor cantidad de código posible en cualquier prueba

- TestCase(herramienta de unittest):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).

Assert(testing):

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.

Método	Descripción
assertEqual assertNotEqual	Aceptan dos objetos comparables, y tendrán éxito si los objetos son iguales o distintos, según corresponda
assertTrue assertFalse	Aceptan una expresión que se evalúa a True, distinto de 0, lista, diccionario, tupla no vacíos, o False, None, 0, lista, diccionario, tupla vacíos
assertGreater assertGreaterEqual assertLess assertLessEqual	Aceptan dos objetos comparables, aseguran el éxito si el primer objeto es mayor, mayor o igual, menor o menor o igual que el segundo objeto.
assertIn assertNotIn	Afirma, si un elemento está o no está en un contenedor de objetos.
assertIsNone assertIsNotNone	Afirma que un elemento tiene (o no tiene) exactamente el valor None.
assertSequenceEqual assertDictEqual assertSetEqual assertListEqual assertTupleEqual	Afirman que dos contenedores tienen exactamente los mismos elementos en el mismo orden. Si falla, se muestra un código, comparando dos listas para ver en qué difieren. Los últimos cuatro métodos, además chequean el tipo de contenedor.

setUP y tearDown(testing):

Para probar situaciones en las cuales se querrá ver que pasa con listas vacías o con lista que contienen valores no números se puede utilizar el método **setUp()** de la clase TestCase para realizar la inicialización de cada prueba.

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

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 sobrescribir los métodos con las implementaciones correspondientes.
- Una clase abstracta no puede ser instanciada.
- Una clase en Python, es abstracta si al menos tiene un método abstracto.

- Las subclases que hereden de una clase base abstracta, que pretendan ser concretas, deben implementar los métodos abstractos, si no lo hacen se convierten automáticamente en clase abstracta.
- El a través del módulo abc (Abstract Base Class), Python define e implementa la abstracción de clases, mediante meta clases y decoradores.
- El decorador para establecer que un método es abstracto, es **@abc.abstractmethod.**
- Una **meta clase**, es una clase que sirve de instancia a otras clases.
- Se importa:

  import abc
  from abc import ABC

Interfaces

- Una interfaz es un conjunto de firmas de métodos, atributos o propiedades eventos agrupados bajo un nombre común (los miembros de una interfaz no poseen modificador de acceso, **por defecto son públicos**)

- 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.

- Las clases que implementan las interfaces son intercambiables con las interfaces, esto hace que a una referencia a la interface se le pueda asignar una referencia a un objeto de la clase que la implementa, la referencia a la interfaz tendrá acceso sólos los métodos, atributos y propiedades declarados por la interface e implementados por la clase.

- 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.

Características:

- Al igual que las clases abstractas, son tipo referencia, pero no pueden crearse objetos directamente de ellas (solo de las clases que las implementen).
- Las clases pueden implementar cualquier número de interfaces (no confundir con herencia múltiple) Una clase que implementa una interfaz, provee implementación a todos los métodos de la interfaz.
- En la POO se dice que las interfaces son funcionalidades (o comportamientos) y las clases representan implementaciones.

¿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 que debe hacerse en lugar de pensar en cómo debe hacerse.

Diccionarios en Python:

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.

- 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 se ha hecho fuerte como alternativa a XML
- Flujo de trabajo:
    a. Importar la librería json
    b. Proveer a las clases un método para codificar la representación tipo diccionario necesaria para JSON
    c. Leer los datos usando las funciones load() o loads()
    d. . Procesar los datos
    e. Escribir los datos usando las funciones dump() o dumps()

- 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().**

Clase Manejador

```
class Manejador(object):
  __puntos: list
  def __init__(self):
     self.__puntos=[]
  def agregarPunto(self, unPunto):
     self.__puntos.append(unPunto)
  def mostrarDatos(self):
     for i in range(len(self.__puntos)):
        print(self.__puntos[i])
  def toJSON(self):
     d = dict(
          __class__=self.__class__.__name__,
          puntos=[punto.toJSON() for punto in self.__puntos]
          )
     return d
```

Genera una lista de Puntos, como valores correspondientes a la clave puntos.
Recordar que punto.toJSON(), genera una representación diccionario del punto que recibe el mensaje.