Unidad 3
Programación Orientada a Objetos

Asignatura: Programación Orientada a Objetos

Licenciatura en Ciencias de la Computación

Licenciatura en Sistemas de Información

Tecnicatura en Programación WEB

Departamento de Informática

FCEFN – UNSJ

Año 2024

Objetivos

- Que el estudiante implemente las relaciones entre clases en el lenguaje Python.
- Que el estudiante adquiera habilidades para definir y capturar excepciones de software.
- Que el estudiante adquiera habilidades para el testeo de software, aplicando testing de unidad.

Bibliografía y Sitios WEB

Lott, Steven F. (2019) Mastering Object-Oriented Python Second Edition – Packt Publishing.
Dusty Phillips - Python 3 Object-oriented Programming, 2nd Edition-Packt Publishing (2015)
Mark Lutz - Learning Python_ powerful object-oriented programming-O'Reilly Media (2013)
Tim Hall, J-P Stacey - Python 3 for Absolute Beginners – Apress (2009)
Mark Summerfield - Programming in Python 3-Addison Wesley (2009)

https://docs.python.org/3/reference/datamodel.html

https://pypi.org/project/zope.interface/

https://zopeinterface.readthedocs.io/en/latest/README.html

Relaciones entre Clases en Python (I)

En la primera etapa del DOO, se identifican las clases.

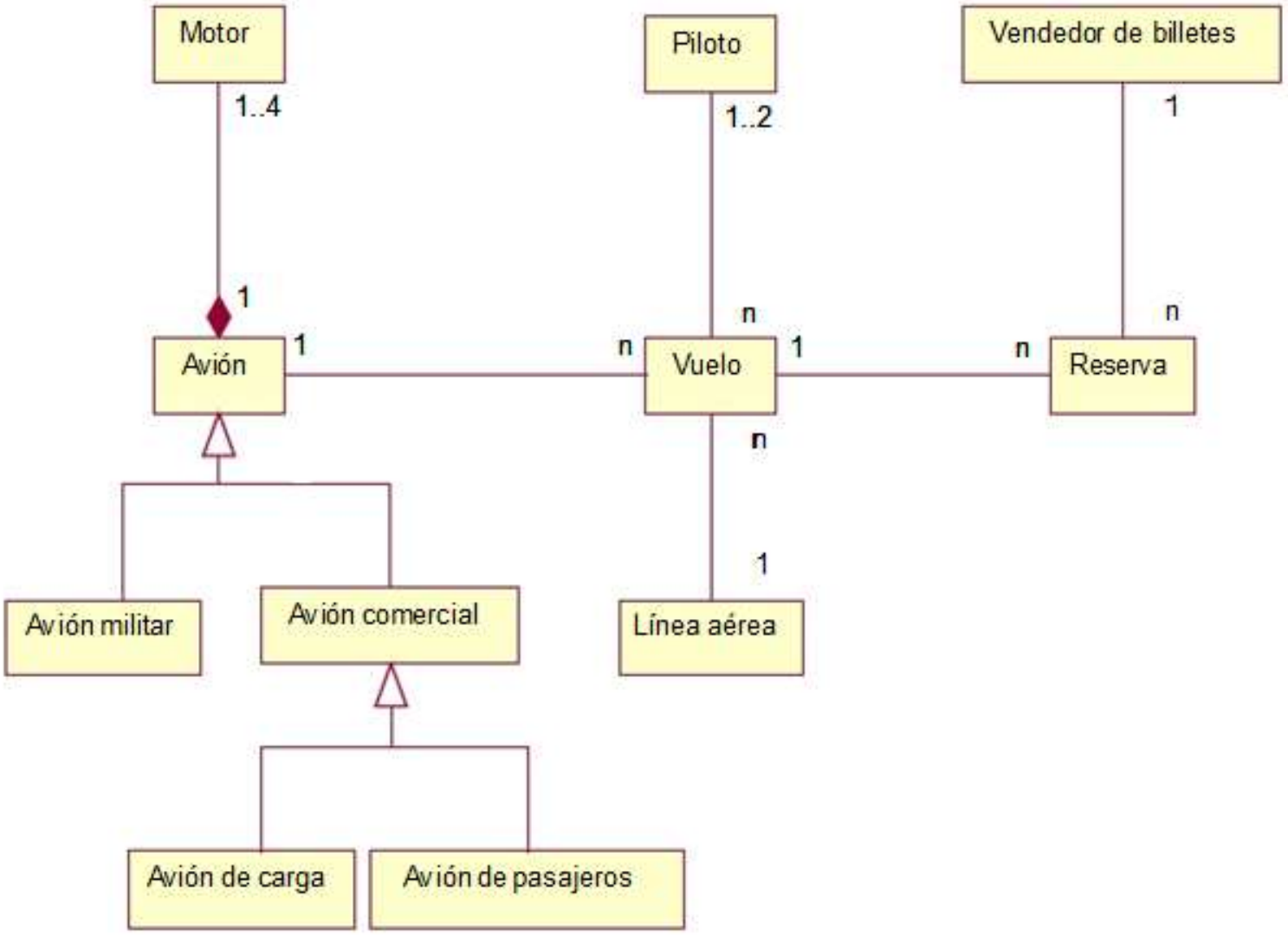
Aunque algunas clases pueden existir aisladas, la mayoría no puede, y deben cooperar unas con otras.

Las relaciones entre las clases expresan una forma de **acoplamiento** entre ellas.

Según el tipo de acoplamiento que presentan las clases podemos distinguir distintos tipos de relaciones.

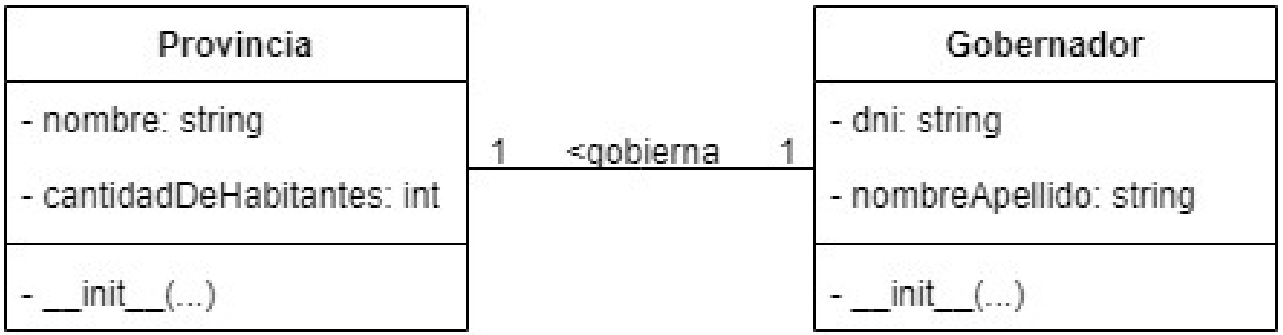
Asociación - Agregación – Composición - Herencia

Relaciones entre Clases en Python (II)



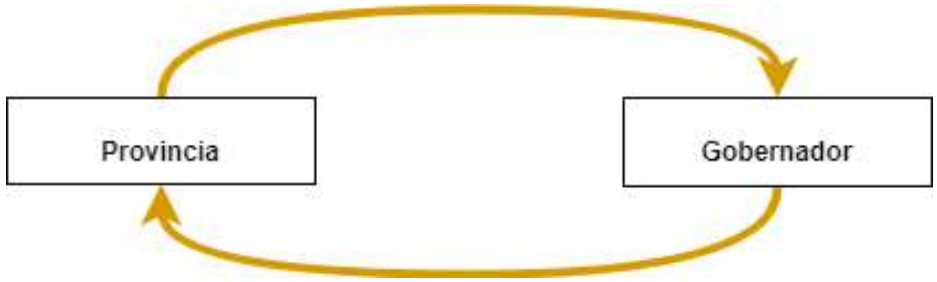
Identificar y nombrar las distintas relaciones

Asociación (I)




```
class Provincia:
    __nombre: str
    __cantidadDeHabitantes: int
    __gobernador: object
    def __init__(self, nombre, cantidadDeHabitantes, gobernador):
        self.__nombre=nombre
        self.__cantidadDeHabitantes=cantidadDeHabitantes
        self.__gobernador=gobernador

class Gobernador:
    __dni: int
    __nombreApellido: str
    __provincia: object
    def __init__(self, dni, nombreApellido):
        self.__dni=dni
        self.__nombreApellido=nombreApellido
        self.__provincia=provincia
```

Conceptualmente la asociación en un diagrama de clases implica transitividad y bidirección de clases. En la implementación de la relación, una Provincia tendrá un atributo que será instancia de la clase Gobernador y Gobernador, tendrá un atributo que será instancia de la clase Provincia. 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.



Cómo se soluciona la referencia circular entre las clases Provincia y Gobernador, ya que para crear una Provincia se necesita y Gobernador, y viceversa???

Lott, Steven – Chapter 2 – Páginas 85 a 93

Solución a la referencia circular

```
class Provincia:
    __nombre: str
    __cantidadDeHabitantes: int
    __gobernador: object
    def __init__(self, nombre, cantidadDeHabitantes,
gobernador=None):
        self.__nombre=nombre
        self.__cantidadDeHabitantes=cantidadDeHabitantes
        self.__gobernador=gobernador
    def __str__(self):
        return 'Provincia: %s, habitantes %d, gobernada por: %s'
%(self.__nombre, self.__cantidadDeHabitantes, self.__gobernador)
    def setGobernador(self, gobernador):
        self.__gobernador=gobernador
```

```
class Gobernador:
    __dni: int
    __nombreApellido: str
    __provincia: object
    def __init__(self, dni, nombreApellido, provincia=None):
        self.__dni=dni
        self.__nombreApellido=nombreApellido
        self.__provincia=provincia
    def __str__(self):
        cadena = 'DNI: %d, Nombre y Apellido: %s' % (self.__dni,
self.__nombreApellido)
        return cadena
    def setProvincia(self, provincia):
        self.__provincia=provincia
```

```
def testAsociacion():
    provincia = Provincia('San Juan',681055)
    gobernador = Gobernador(27888111, 'Sergui Uñac', provincia)
    provincia.setGobernador(gobernador)
    print(provincia)
if __name__=='__main__':
    testAsociacion()
```

Una vez creada la instancia provincia, que no inicializa el gobernador, se puede resolver en la clase Gobernador, de alguna otra forma de modo de evitar la sentencia provincia.setProvincia(gobernador), que está en el programa principal???

Otro posible error:

Cuando se programa modularmente, cada clase se programa en un módulo distinto, y se pretende validar el tipo de datos previo a la asignación, en cada módulo se importa el otro módulo

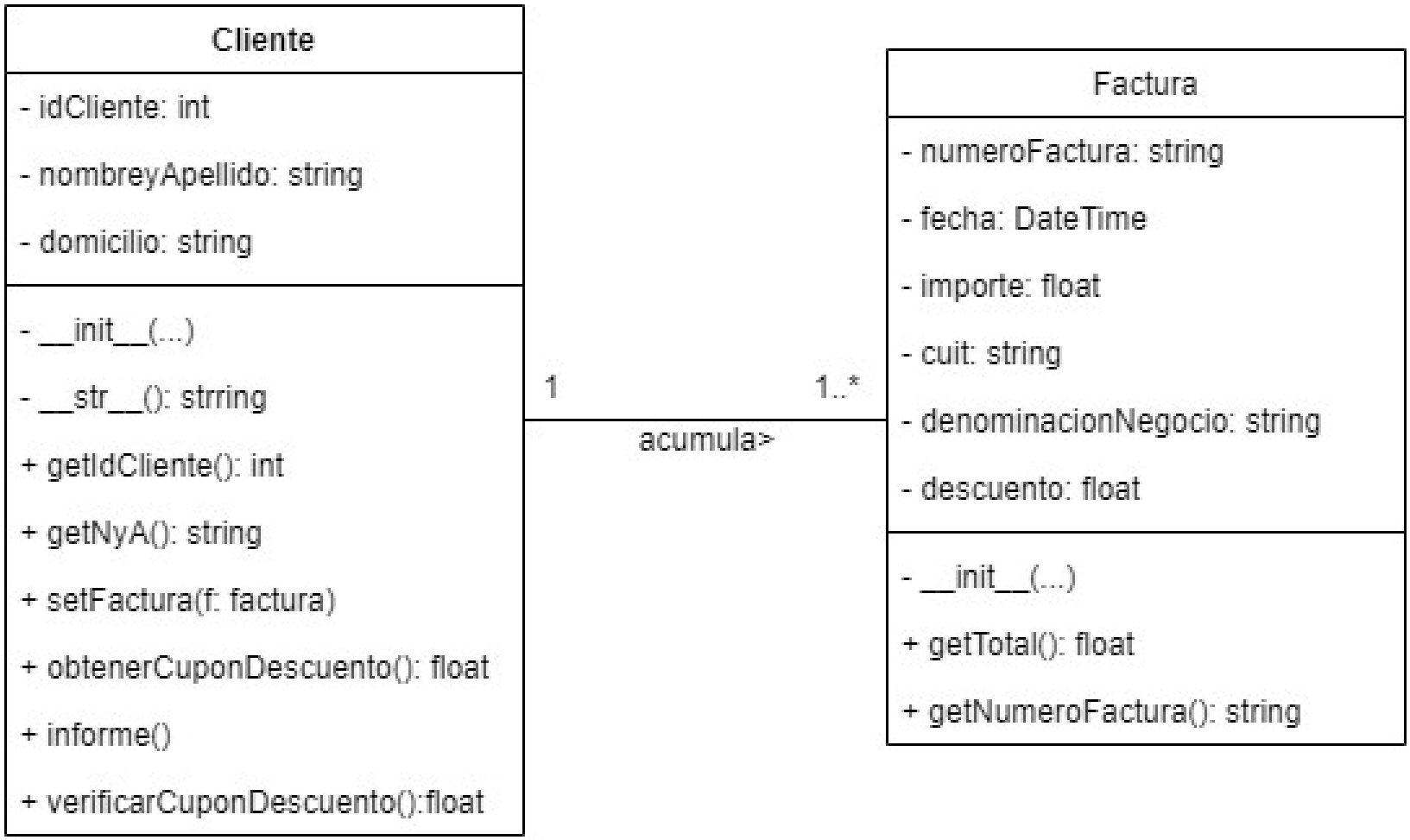
ImportError: cannot import name 'Gobernador' from partially initialized module 'claseGobernador' (most likely due to a circular import)

Asociación (II)

Contexto: Un shopping en el día de la madre presenta una promoción para sus clientes. El cliente puede comprar en cualquier local del shopping y en cada local se le realizará la factura por el total consumido. Por cada compra que supera los $2500 se les descuenta $250. Por cada compra menor a $2500 se le reintegra $100.
Al término de sus compras, al cliente se le restituye un porcentaje de dinero en concepto de descuento si compró en más de 3 negocios (más de 3 facturas). En este caso se emite un cupón de descuento para futuras compras que asciende al 12% del importe total gastado.

Se le solicita a usted que construya una aplicación, que registre las facturas de compra que se emiten en todos los locales del shopping, a medida que los clientes hacen sus compras.
El programa deberá:
1. Informar a un cliente cada una de las compras realizadas incluyendo: número e importe de cada factura, e importe total acumulado.
2. Informar a un cliente si posee cupón de descuento e importe del mismo.

Asociación (III)



Asociación (IV)

```
class Factura:
    __numeroFactura: str
    __fecha: object
    __importe: float
    __cuit: str
    __denominacionNegocio: str
    __descuento: float
    def __init__(self, numeroFactura,fecha, importe, cuit,
denominacionNegocio):
        self.__numeroFactura=numeroFactura
        self.__fecha=fecha
        self.__importe=importe
        self.__cuit=cuit
        self.__denominacionNegocio=denominacionNegocio
        if self.__importe>2500:
            self.__descuento=250
        else:
            self.__descuento=100
    def getTotal(self):
        return self.__importe-self.__descuento
    def getNumeroFactura(self):
        return self.__numeroFactura
```

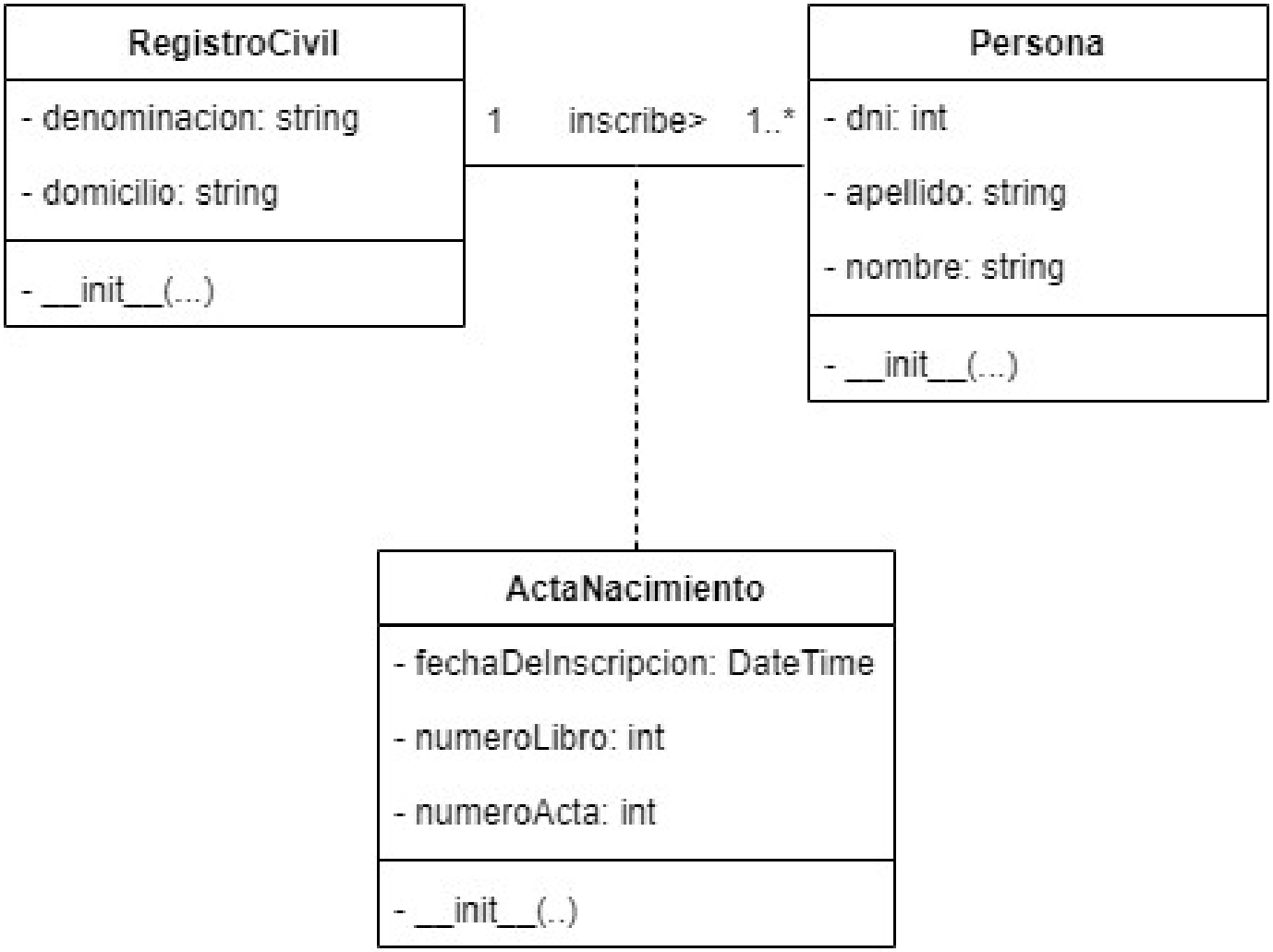
Asociación (V)

```
class Cliente:
    __idCliente: int
    __nombreyApellido: str
    __direccion: str
    __facturas: list
    def __init__(self,idCliente, nombreyApellido, direccion):
        self.__idCliente=idCliente
        self.__nombreyApellido=nombreyApellido
        self.__direccion=direccion
    def setFactura(self, unaFactura):
        self.__facturas.append(unaFactura)
    def getIdCliente(self):
        return self.__idCliente
    def getNyA(self):
        return self.__nombreyApellido
    def obtenerCuponDescuento(self):
        descuento = 0.0
        total = 0.0
        if len(self.__facturas)>2:
            for factura in self.__facturas:
                total+=factura.getTotal()
        descuento = total * 12/100
        return descuento
    def informe(self):
        total = 0.0
        print('Cliente: ',self.getNyA())
        for factura in self.__facturas:
            print('Factura {}, importe, {}'.format(factura.getNumeroFactura(), factura.getTotal()))
            total+=factura.getTotal()
        print('Total acumulado {0:8.2f}'.format(total))
    def verificarCuponDescuento(self):
        descuento=self.obtenerCuponDescuento()
        print('Cliente: {}, descuento: {}'.format(self.getNyA(),descuento))
```

```
def testClienteFactura():
    unCliente = Cliente(123, 'Luis Ventura', 'Mitre 156 (O)')
    factura1 = Factura('223','27/01/2020',4500, '30-33333323-1', 'Calzados la Zapa')
    unCliente.setFactura(factura1)
    factura2 = Factura('441','27/01/2020',6500, '27-12334333-1', 'Deportes ABC')
    unCliente.setFactura(factura2)
    factura3 = Factura('223','27/01/2020',8500, '20-31333333-1', 'Bicicletería la
BICI')
    unCliente.setFactura(factura3)
    unCliente.informe()
    unCliente.verificarCuponDescuento()
if __name__=='__main__':
    testClienteFactura()
```

Qué pasa si para cada factura se quiere saber el cliente???

Clase Asociación (I)



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 asociación representa una relación uno a uno.

Clase Asociación (II)

```
class RegistroCivil:
        __denominacion: str
        __domicilio: str
        __actas: list
        # Variables de clase
        __actaActual = 100
        __libroActual = 5
        def __init__(self, denominacion, domicilio):
            self.__denominacion = denominacion
            self.__domicilio = domicilio
            self.__actas=[]
        # Métodos de clase
        @classmethod
        def getActaActual(cls):
            cls.__actaActual+=1
            return cls.__actaActual
        @classmethod
        def getLibroActual(cls):
            return cls.__libroActual
        def inscribirPersona(self, persona, fecha):
            numeroActa = self.getActaActual()
            libro = self.getLibroActual()
            acta = ActaNacimiento(numeroActa, libro, fecha, persona, self)
            self.__actas.append(acta)
        def mostrarActas(self):
            for acta in self.__actas:
                print(acta)
```

```
class Persona:
        __dni: int
        __apellido: str
        __nombre: str
        def __init__(self, dni, nombre, apellido):
            self.__dni=dni
            self.__nombre=nombre
            self.__apellido=apellido
        def __str__(self):
            cadena = 'DNI: '+str(self.__dni)+'\n'
            cadena += 'Apellido: '+self.__apellido+', Nombre:
'+self.__nombre+'\n'
            return cadena
```

Clase Asociación (III)

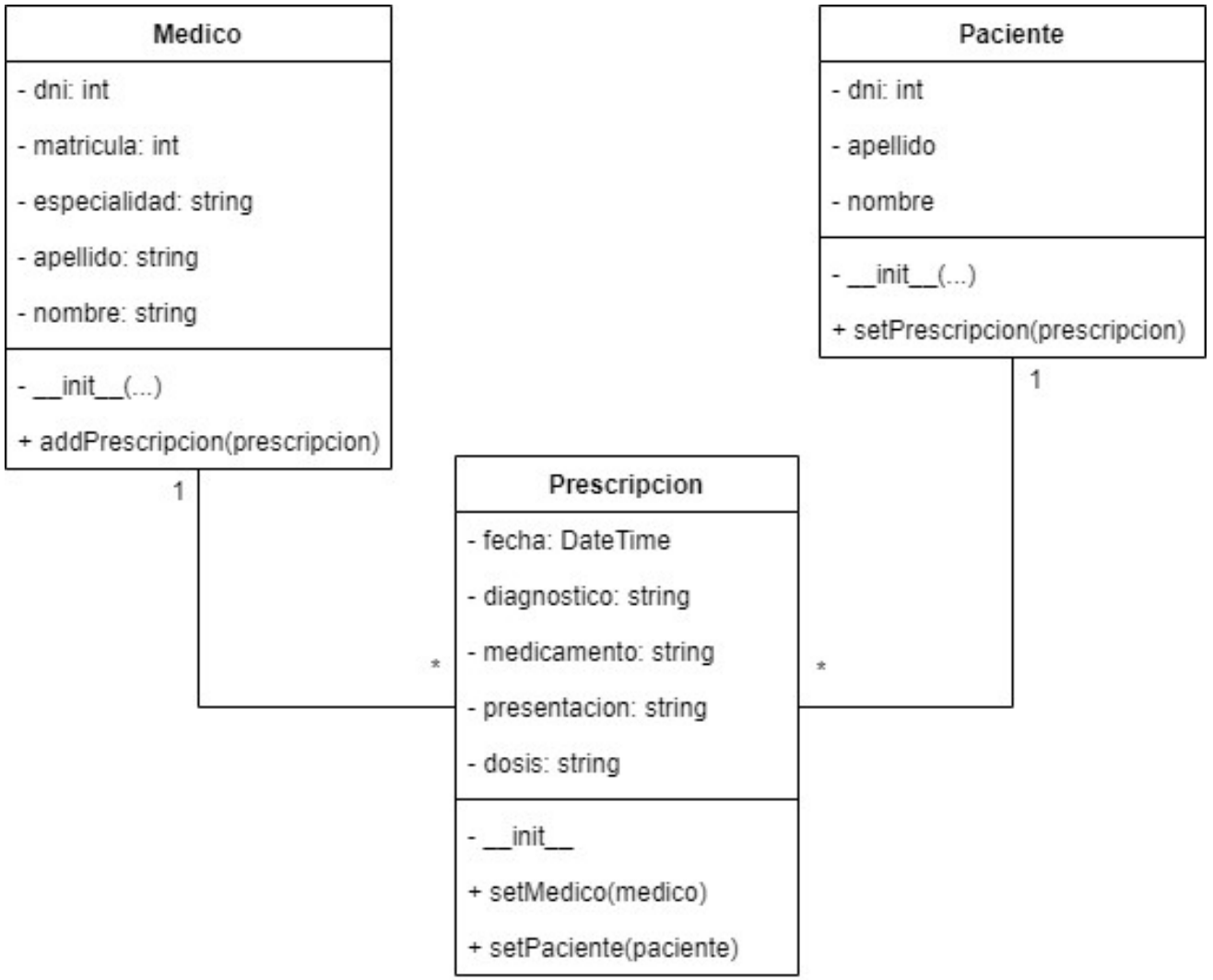
```
class ActaNacimiento:
        __fechaInscripcion: str
        __numeroroLibro: int
        __numeroActa: int
        __persona: object
        __registrocivil: object
        def __init__(self, nroActa, nroLibro, fechaInscripcion, persona, registroCivil):
            self.__numeroActa = nroActa
            self.__numeroLibro = nroLibro
            self.__fechaInscripcion = fechaInscripcion
            self.__persona = persona
            self.__registrocivil = registroCivil
        def __str__(self):
            cadena = 'Fecha de Inscripcion '+self.__fechaInscripcion+'\n'
            cadena +='Libro: '+str(self.__numeroLibro) + ' Acta: '+str(self.__numeroActa)+'\n'
            cadena+= str(self.__persona)
            return cadena
```

```
def testClaseAsociacion():
    registro = RegistroCivil('Registro Cuarta Zona', 'Av. Córdoba y
Urquiza')
    persona = Persona(20112113, 'Carlos', 'Vargas')
    persona1 = Persona(3444222, 'Anastacia', 'Arboleda')
    registro.inscribirPersona(persona, '28/01/2019')
    registro.inscribirPersona(persona1, '28/01/2019')
    registro.mostrarActas()
if __name__=='__main__':
    testClaseAsociacion()
```

Consola Python

```
Fecha de Inscripcion 28/01/2019
Libro: 5 Acta: 101
DNI: 20112113
Apellido: Vargas, Nombre: Carlos

Fecha de Inscripcion 28/01/2019
Libro: 5 Acta: 102
DNI: 3444222
Apellido: Arboleda, Nombre: Anastacia
```

Clase que modela la asociación (I)



En este caso existe una relación de asociación entre Médico y Paciente que se deriva o modela a través de la clase Prescripción. El mismo Médico y el mismo Paciente, se relacionan más de una vez, cada vez que el Médico atiende al mismo Paciente, y produce nuevas Prescripciones.

Clase que modela la asociación (II)

```
class Medico:
    __dni: int
    __matricula: int
    __especialidad: str'
    __apellido: str
    __nombre: str
    __prescripciones: list
    def __init__(self, dni, matricula, especialidad, apellido, nombre):
        self.__dni=dni
        self.__matricula=matricula
        self.__especialidad=especialidad
        self.__apellido=apellido
        self.__nombre=nombre
    def addPrescipcion(self, prescripcion):
        self.__prescripciones.append(prescripcion)
```

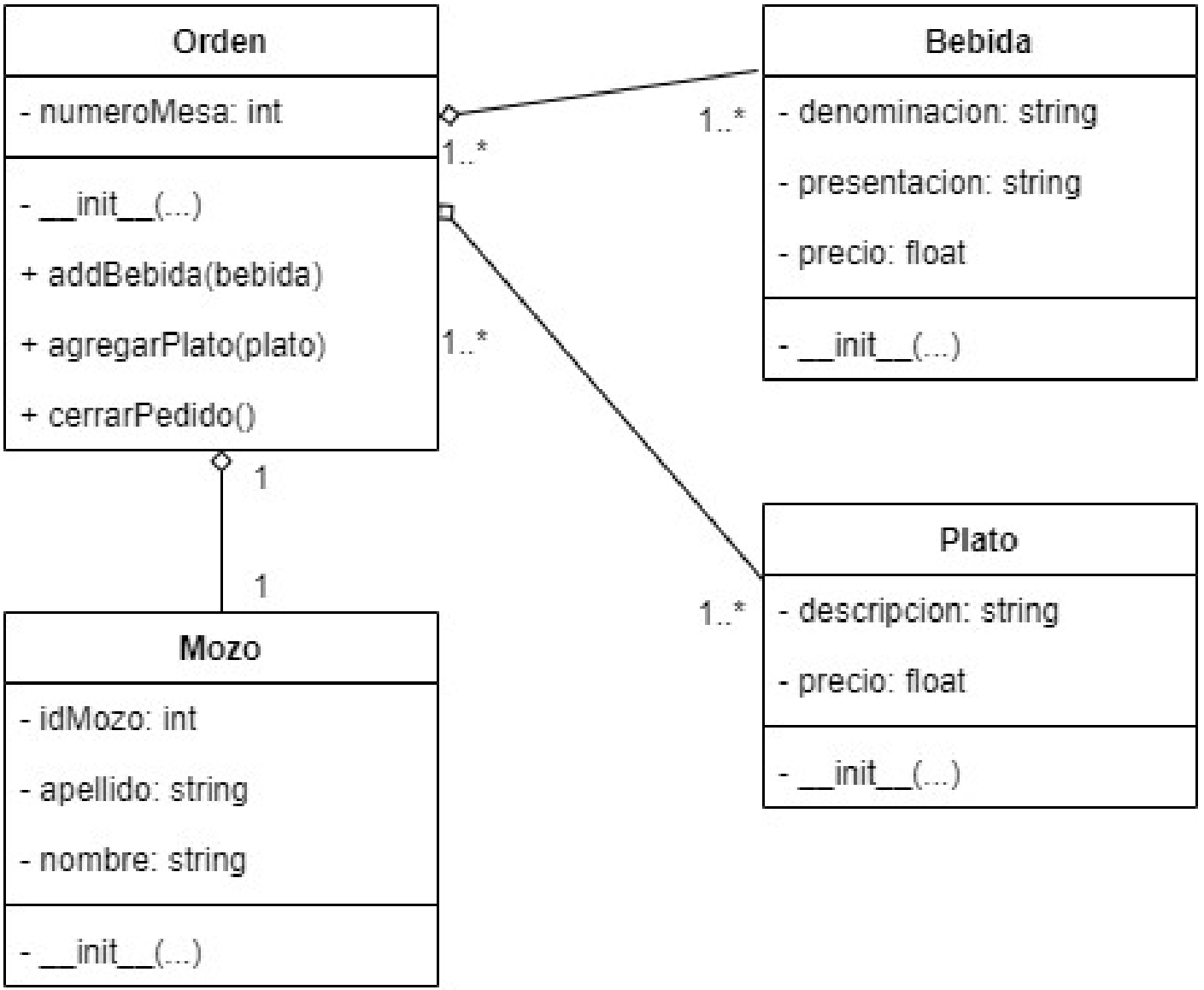
```
def testClaseModelaAsociacion():
    paciente = Paciente(14555699, 'Vergara', 'Andrea')
    medico = Medico(19327881, 1125, 'Clínica Médica',
'González', 'Jorge')
    prescripcion = Prescripcion('11/01/2020', 'Rinitis', 'Hexaler',
'10 comprimidos', '1 por día',medico, paciente)
    prescripcion2 = Prescripcion('29/01/2020', 'Otitis', 'Ciriax
Gotas', 'envase 10 ml', '2 gotas cada 8h',medico, paciente)
if __name__=='__main__':
    testClaseModelaAsociacion()
```

```
class Paciente:
    __dni: int
    __apellido: str
    __nombre: str
    __prescripciones: list
    def __init__(self, dni, apellido, nombre):
        self.__dni=dni
        self.__apellido=apellido
        self.__nombre=nombre
    def addPrescripcion(self, prescripcion):
        self.__prescripciones.append(prescripcion)
```

```
class Prescripcion:
    __fecha: str
    __diagnostico: str
    __medicacion: str
    __presentacion: str
    __dosis: str
    __paciente: object
    __medico: object
    def __init__(self, fecha, diagnostico, medicacion,
presentacion, dosis, medico, paciente):
        self.__fecha=fecha
        self.__diagnostico=diagnostico
        self.__medicacion=medicacion
        self.__presentacion=presentacion
        self.__dosis=dosis
        self.__medico=medico
        self.__paciente=paciente
        self.__medico.addPrescipcion(self)
        self.__paciente.addPrescripcion(self)
```

Agregación (I)

- 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.



Mark Lutz – Chapter 28 – Páginas 836 a 838

Agregación (II)

```
class Bebida:
    __denominacion: str
    __presentacion: str
    __precio: float
    def __init__(self, denominacion, presentacion, precio):
        self.__denomiancion=denominacion
        self.__presentacion=presentacion
        self.__precio=precio
    def getPrecio(self):
        return self.__precio
    def getDenominacion(self):
        return self.__denomiancion
```

```
class Mozo:
    __idMozo: int
    __apellido: str
    __nombre. str
    def __init__(self, idMozo, apellido, nombre):
        self.__idMozo=idMozo
        self.__apellido=apellido
        self.__nombre=nombre
```

```
class Plato:
    __descripcion: str
    __precio: float
    def __init__(self, descripcion, precio):
        self.__descripcion=descripcion
        self.__precio=precio
    def getPrecio(self):
        return self.__precio
    def getDescripcion(self):
        return self.__descripcion
```

Agregación (III)

```
class Pedido:
    __cantidadPedidos=0
    __idPedido: int
    __numeroMesa: int
    __mozo: object
    __bebidas: list
    __platos: list
    @classmethod
    def getIdPedido(cls):
        cls.__cantidadPedidos+=1
        return cls.__cantidadPedidos
    def __init__(self, numeroMesa, mozo, bebida=None, plato=None):
        self.__numeroMesa=numeroMesa
        self.__mozo=mozo
        self.__idPedido=self.getIdPedido()
        if bebida!=None:
            self.addBebida(bebida,1)
        if plato!=None:
            self.addPlato(plato,1)
    def addBebida(self, bebida, cantidad):
        for i in range(cantidad):
            self.__bebidas.append(bebida)
    def addPlato(self, plato, cantidad):
        for i in range(cantidad):
            self.__platos.append(plato)
```

```
    def cerrarPedido(self):
        print('Pedido número: ',self.__idPedido)
        total = 0
        print('Bebidas')
        for bebida in self.__bebidas:
            precio = bebida.getPrecio()
            print('{0:20s} {1:4.2f}'.format(bebida.getDenominacion(),  precio))
            total+=precio
        print('Platos')
        for plato in self.__platos:
            precio = plato.getPrecio()
            print('{0:20s} {1:4.2f}'.format(plato.getDescripcion(), precio))
            total+=precio
        print('Total a pagar:      {0:4.2f}'.format(total))
```

Agregación (IV)

```
def testAgregacion():
    bebida = Bebida('Coca cola','1/2 litro',750)
    bebida1 = Bebida('Aquarius', '1/2 litro',500)
    plato = Plato('Lomo especial',3250)
    papas = Plato('Papa frita chica',1250)
    pizza = Plato('Pizza especial', 3900)
    mozo1 = Mozo(1, 'López', 'Carlos')
    pedido1 = Pedido(1, mozo1, bebida, plato)
    pedido1.addBebida(bebida1,2)
    pedido1.addPlato(papas,1)
    pedido1.addPlato(plato,3)
    pedido1.cerrarPedido()
    del pedido1
    pedido2 = Pedido(2, mozo1, bebida1, papas)
    pedido2.addBebida(bebida,2)
    pedido2.addPlato(pizza,1)
    pedido2.cerrarPedido()

if __name__=='__main__':
    testAgregacion()
```

Consola Python

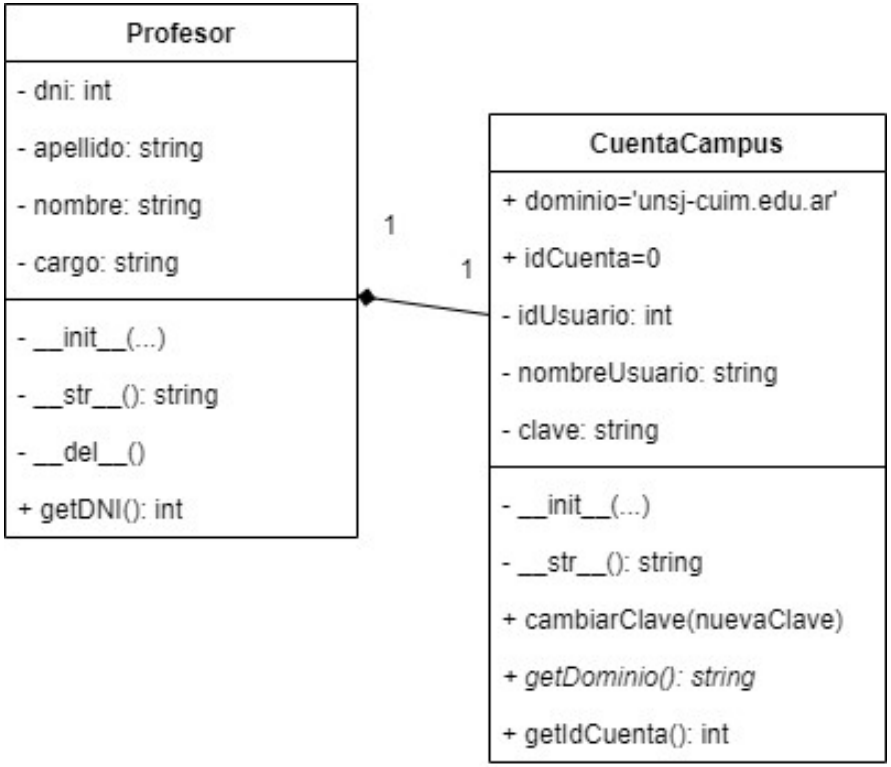
```
Pedido número:  1
Bebidas
Coca cola          150.00
Aquarius           100.00
Aquarius           100.00
Platos
Lomo especial       325.00
Papa frita chica    125.00
Lomo especial       325.00
Lomo especial       325.00
Lomo especial       325.00
Total a pagar:     1775.00
Pedido número:  2
Bebidas
Coca cola          150.00
Aquarius           100.00
Aquarius           100.00
Aquarius           100.00
Coca cola          150.00
Coca cola          150.00
Platos
Lomo especial       325.00
Papa frita chica    125.00
Lomo especial       325.00
Lomo especial       325.00
Lomo especial       325.00
Papa frita chica    125.00
Pizza especial      390.00
Total a pagar:     2690.00
```

Composición (I)

- 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.



Contexto: cuando se crea un nuevo Profesor, se crea automáticamente la cuenta en el Campus, que será su nombre y apellido en minúsculas, seguido por el dominio. La clave por defecto es el número de documento del profesor. El idUsuario es un número consecutivo que comienza desde 0, y se incrementa con cada nuevo usuario.

Composición (II)

```
class CuentaCampus:
    #Variables de clase
    __dominio='@unsj-cuim.edu.ar'
    __idCuenta=0
     # Variables de instancia
    __idUsuario: int
    __nombreUsuario: str
    __clave: str
    # Métodos de clase
    @classmethod
    def getDominio(cls):
        return cls.__dominio
    @classmethod
    def getIdCuenta(cls):
        cls.__idCuenta+=1
        return cls.__idCuenta
    # Métodos de instancia
    def __init__(self, idUsuario, nombreUsuario, clave):
        self.__idUsuario=idUsuario
        self.__nombreUsuario=nombreUsuario
        self.__clave=clave
    def __str__(self):
        cadena = 'Usuario: {}\nClave {}'.format(self.__nombreUsuario, self.__clave)
        return cadena
    def cambiarClave(self, nuevaClave):
        self.__clave=nuevaClave
```

Composición (III)

```
class Profesor:
    __dni: int
    __apellido: str
    __nombre: str
    __cuentaCampus: object
    def __init__(self, dni, apellido, nombre):
        self.__dni=dni
        self.__apellido=apellido
        self.__nombre=nombre
        idCuenta=CuentaCampus.getIdCuenta()
        dominio=CuentaCampus.getDominio()
        usuario=nombre.lower()+apellido.lower()+dominio
        self.__cuentaCampus=CuentaCampus(idCuenta,usuario,dni)
    def __del__(self):
        print('Borrando cuenta de usuario....')
        del self.__cuentaCampus
    def __str__(self):
        cadena ='Profesor: \n'
        cadena += 'Apellido y nombre: {}, {}\n'.format(self.__apellido, self.__nombre)
        cadena+=str(self.__cuentaCampus)
        return cadena
```

```
def testComposicion():
    profesor = Profesor(11334441, 'Rodríguez', 'Myriam')
    print(profesor)
    del profesor
if __name__=='__main__':
    testComposicion()
```

Consola Python

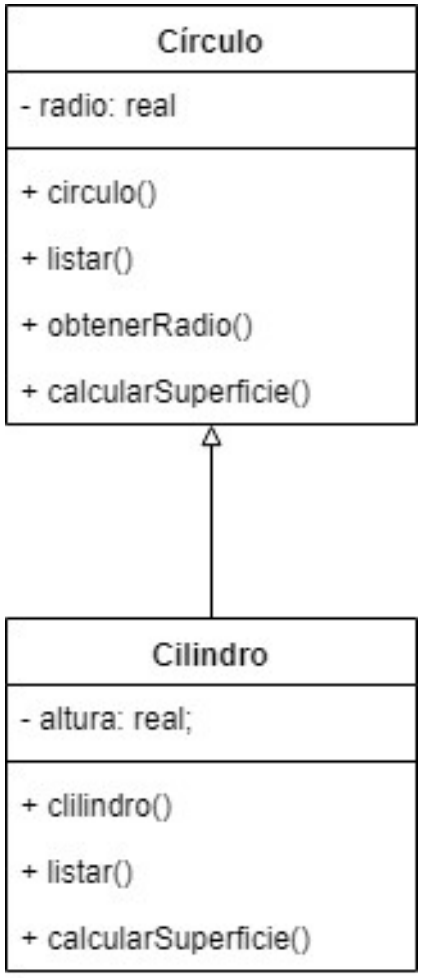
```
Profesor:
Apellido y nombre: Rodríguez, Myriam
Usuario: myriamrodríguez@unsj-cuim.edu.ar
Clave 11334441
Borrando cuenta de usuario....
```

Herencia (I)

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.

Si en la definición de una clase indicamos que ésta deriva de otra, entonces la primera -a la que se le suele llamar **clase hija**- será tratada por el intérprete automáticamente como si su definición incluyese la definición de la segunda –a la que se le suele llamar **clase padre** o **clase base**.

En este ejemplo, un objeto instancia de la clase cilindro además de los miembros de la clase círculo, tendrá un nuevo miembro (altura)



Clase padre o base

Clase derivada o hija

Herencia (II)

```
import math
class Circulo:
    __radio: float
    def __init__(self, radio):
        self.__radio=radio
    def superficie(self):
        return math.pi*self.__radio**2
    def getRadio(self):
        return self.__radio
    def listar(self):
        print('Circulo')
        print('Radio: {0:3.2f}, superficie {1:7.5f}'.format(self.__radio, self.superficie()))
```

```
class Cilindro(Circulo):
    __altura: float
    def __init__(self, radio, altura):
        super().__init__(radio)
        self.__altura=altura
    def superficie(self):
        superficieLateral=math.pi*2*self.getRadio()
        superficieCirculo = super().superficie()
        return superficieLateral+2*superficieCirculo
    def listar(self):
        print('Cilindro')
        print('Radio {0:3.2f}, Altura: {1:3.2f}, superficie {2:7.5f}'.format(self.getRadio(), self.__altura, self.superficie()))
```

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)

Qué ventajas tiene una forma respecto a la otra de invocar el constructor?

Dusty Phillips – Chapter 3 – Páginas 59 a 74

Herencia (III)

```
def testCirculoCilindro():
    circulo = Circulo(3)
    cilindro = Cilindro(5,7)
    circulo.listar()
    cilindro.listar()
if __name__=='__main__':
    testCirculoCilindro()
```

Consola Python

```
Circulo
Radio: 3.00, superficie 28.27433
Cilindro
Radio 5.00, Altura: 7.00, superficie 188.49556
```

Herencia (IV)

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.

```
class Punto(object):
    __x=0
    __y=0
    def __init__(self, x, y):
        self.__x=x
        self.__y=y
if __name__=='__main__':
    print(dir(Punto))
```

El mismo resultado se obtiene si se escribe

```
class Punto:
```

Consola Python

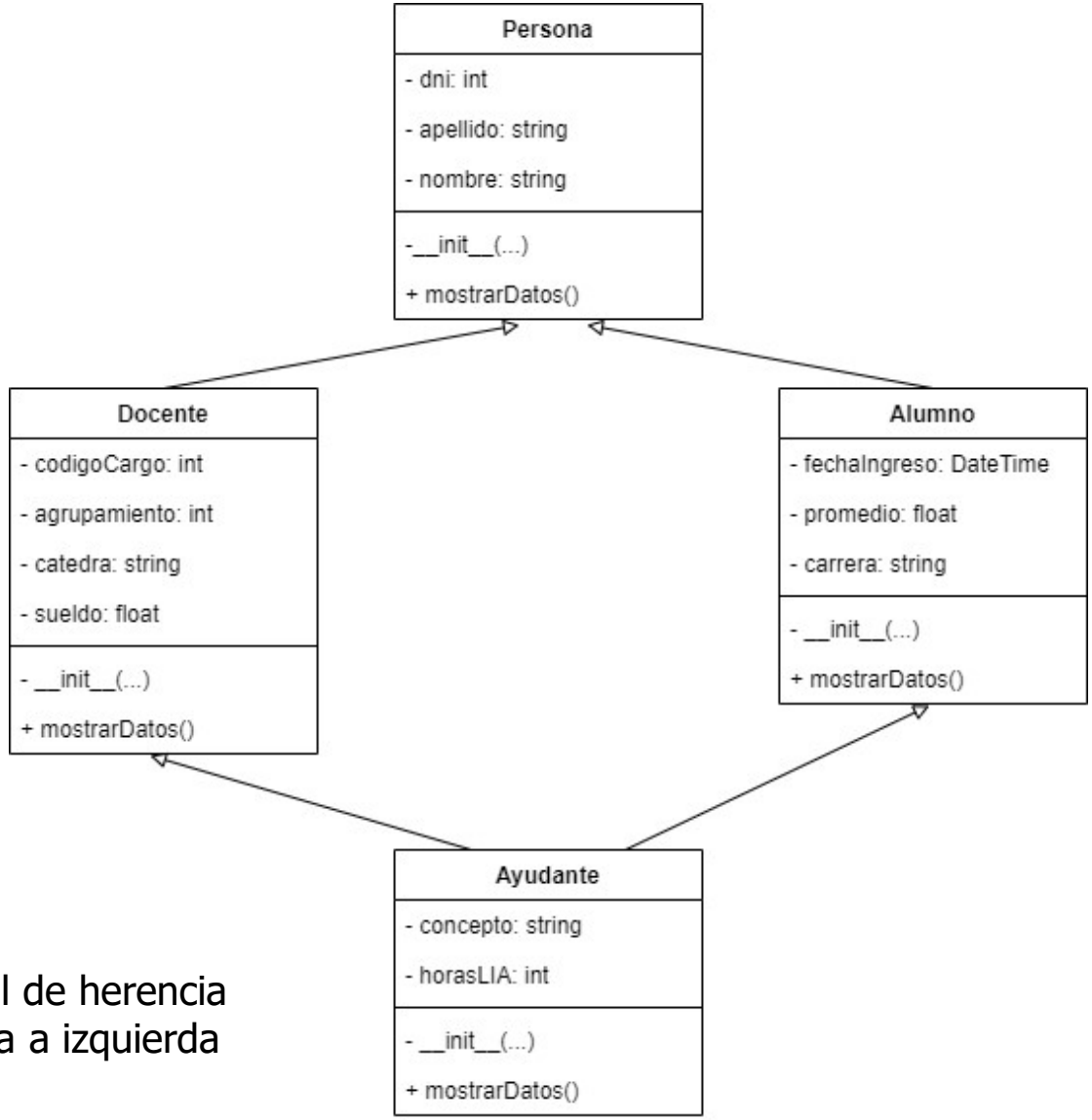
```
['_Punto__x', '_Punto__y', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

Como puede observarse, aparecen hererados métodos que permiten la comparación de objetos, como lo son los métdos __eq__, __ge__, __gt__, __le__, __lt__, __ne__ (vistos en la sobrecarga de operadores). Los métodos __init__, __new__, relacionados a la creación de objetos, es decir el constructor, __str__ para convertir el estado del objeto en una secuencia de caracteres. Los métodos __init__, __new__ y __str__, se han venido usando escribiendo en las clases, es decir sobrescribiendo a los métodos heredados y ocultándolos.

https://docs.python.org/3/reference/datamodel.html

Herencia Múltiple (I)

La herencia múltiple es un tema muy delicado, no todos los lenguajes de programación la implementan.
Python sí la implementa.
Una subclase deriva de más de una clase base, muy simple de enunciar.



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.
Cuando 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)**

Mark Lutz – Chapter 8 – Páginas 388 a 390 y Chapter 32 – páginas 997 a 1009

Herencia Múltiple (II)

```
class Persona(object):
    __dni: int
    __apellido: str
    __nombre: s
    def __init__(self, dni, apellido, nombre, codigoCargo=0, agrupamiento=0,
                 catedra='', sueldo=0.0, fechaIngreso='', promedio=0.0, carrera=''):
        self.__dni=dni
        self.__apellido=apellido
        self.__nombre=nombre
    def mostrarDatos(self):
        print('Datos Persona')
        print('DNI: {0:9d}'.format(self.__dni))
        print('Apellido: {}, Nombre: {}'.format(self.__apellido, self.__nombre))


class Docente(Persona):
    __codigoCargo: str
    __agrupamiento: int
    __catedra: str
    __sueldo: float
    def __init__(self, dni, apellido, nombre, fechaIngreso, promedio, carrera,
                 codigoCargo, agrupamiento, catedra, sueldo):
        super().__init__(dni, apellido, nombre, fechaIngreso, promedio,
            carrera, codigoCargo, agrupamiento,catedra, sueldo)
        self.__codigoCargo=codigoCargo
        self.__agrupamiento=agrupamiento
        self.__catedra=catedra
        self.__sueldo=sueldo
    def mostrarDatos(self):
        super().mostrarDatos()
        print('Datos del Docente')
        print('Codigo cargo {}/{}'.format(self.__codigoCargo, self.__agrupamiento))
        print('Cátedra: {0}, sueldo ${1:8.2f}'.format(self.__catedra, self.__sueldo))
```

Herencia Múltiple (III)

```
class Alumno(Persona):
    __fechaIngreso: str
    __promedio: float
    __carrera: str
    def __init__(self, dni, apellido, nombre, fechaIngreso, promedio, carrera,
                 codigoCargo, agrupamiento,catedra, sueldo):
        super().__init__(dni, apellido, nombre, fechaIngreso, promedio,
            carrera, codigoCargo, agrupamiento,catedra, sueldo)
        self.__fechaIngreso=fechaIngreso
        self.__promedio=promedio
        self.__carrera=carrera
    def mostrarDatos(self):
        super().mostrarDatos()
        print('Datos del Alumno')
        print('Carrera: {}, fecha de ingreso: {}'.format(self.__carrera, self.__fechaIngreso))
        print('Promedio {0}'.format(self.__promedio))


class Ayudante(Docente, Alumno):
    __concepto: str
    __horasLIA: int
    def __init__(self, dni, apellido, nombre, fechaIngreso, promedio, carrera, codigoCargo, agrupamiento, catedra, sueldo,
concepto, horasLIA=0):
#         Docente.__init__(self, dni, apellido, nombre, codigoCargo, agrupamiento, catedra, sueldo)
#         Alumno.__init__(self, dni, apellido, nombre, fechaIngreso, promedio, carrera)
        super().__init__(dni, apellido, nombre, fechaIngreso, promedio, carrera, codigoCargo, agrupamiento,
catedra, sueldo)
        self.__concepto=concepto
        self.__horasLIA=horasLIA
    def mostrarDatos(self):
        super().mostrarDatos()
        print('Datos Ayudante')
        print('Horas LIA {}'.format(self.__horasLIA))
        print('Concepto: {}'.format(self.__concepto))
```

Herencia Múltiple (IV)

Consola Python

```
MRO de la clase Ayudante:  [<class '__main__.Ayudante'>, <class '__main__.Docente'>, <class '__main__.Alumno'>, <class '__main__.Persona'>, <class 'object'>]
Datos Persona
DNI:  41223444
Apellido: Juarez, Nombre: Roberto
Datos del Alumno
Carrera: LSI, fecha de ingreso: 10/03/2020
Promedio 9.65
Datos del Docente
Codigo cargo 3211/90
Cátedra: POO, sueldo $ 2500.00
Datos Ayudante
Horas LIA 5
Concepto: Sobresaliente
```

```
def testHerenciaMultiple():
    print('MRO de la clase Ayudante: ',Ayudante.mro())
    ayudante = Ayudante(41223444, 'Juarez', 'Roberto',
'10/03/2020',9.65, 'LSI', 3211, 90,'POO',2500,'Sobresaliente',5)
    ayudante.mostrarDatos()
if __name__=='__main__':
    testHerenciaMultiple()
```

Consola Python

```
MRO de la clase Ayudante:  [<class '__main__.Ayudante'>, <class '__main__.Alumno'>, <class '__main__.Docente'>, <class '__main__.Persona'>, <class 'object'>]
Datos Persona
DNI:  41223444
Apellido: Juarez, Nombre: Roberto
Datos del Docente
Codigo cargo 3211/90
Cátedra: POO, sueldo $ 2500.00
Datos del Alumno
Carrera: LSI, fecha de ingreso: 10/03/2020
Promedio 9.65
Datos Ayudante
Horas LIA 5
Concepto: Sobresaliente
```

```
class Ayudante(Alumno, Docente):
```

MRO cambiado

Herencia Múltiple (V)

Como se vio en el ejemplo anterior, se hizo muy tedioso pasar argumentos a los constructores de las clases intermedias y clase base.

¿Existirá otra forma de resolverlo que oculte esa cantidad de parámetros?

```
class Circulo:
    __radio: float
    def __init__(self, radio):
        self.__radio=radio
    def getRadio(self):
        return self.__radio
```

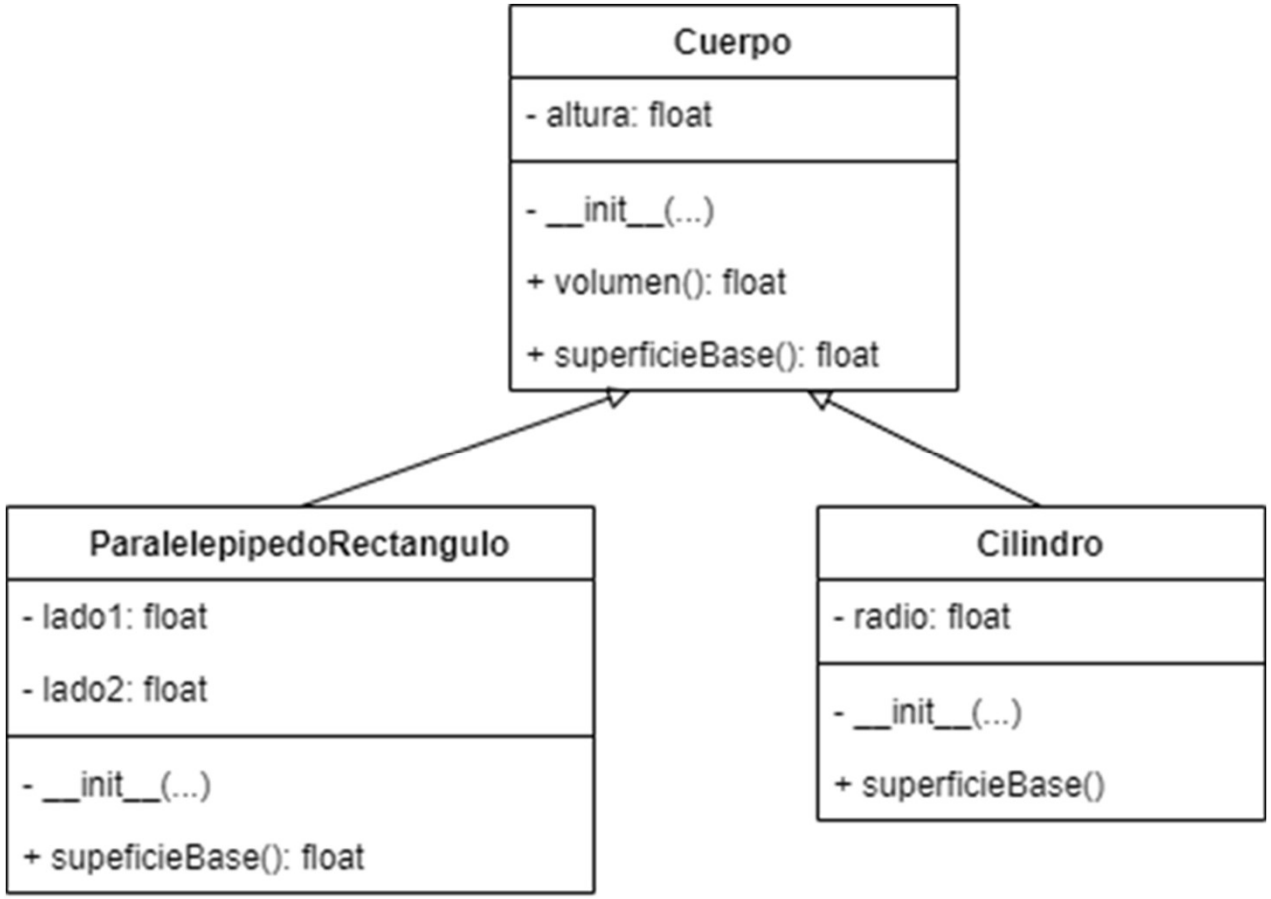
```
class Cilindro(Circulo):
    __altura: float
    def __init__(self, radio, **kwargs):
        super().__init__(radio)
        self.__altura=kwargs['altura']
    def __str__(self):
        return f'Radio: {self.getRadio()}, Altura: {self.__altura}'
```

```
if __name__=='__main__':
    cilindro=Cilindro(radio=4,altura=8)
    print(cilindro)
```

Polimorfismo: Vinculación Dinámica (I)

- 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.

Dada la siguiente jerarquía de clases:

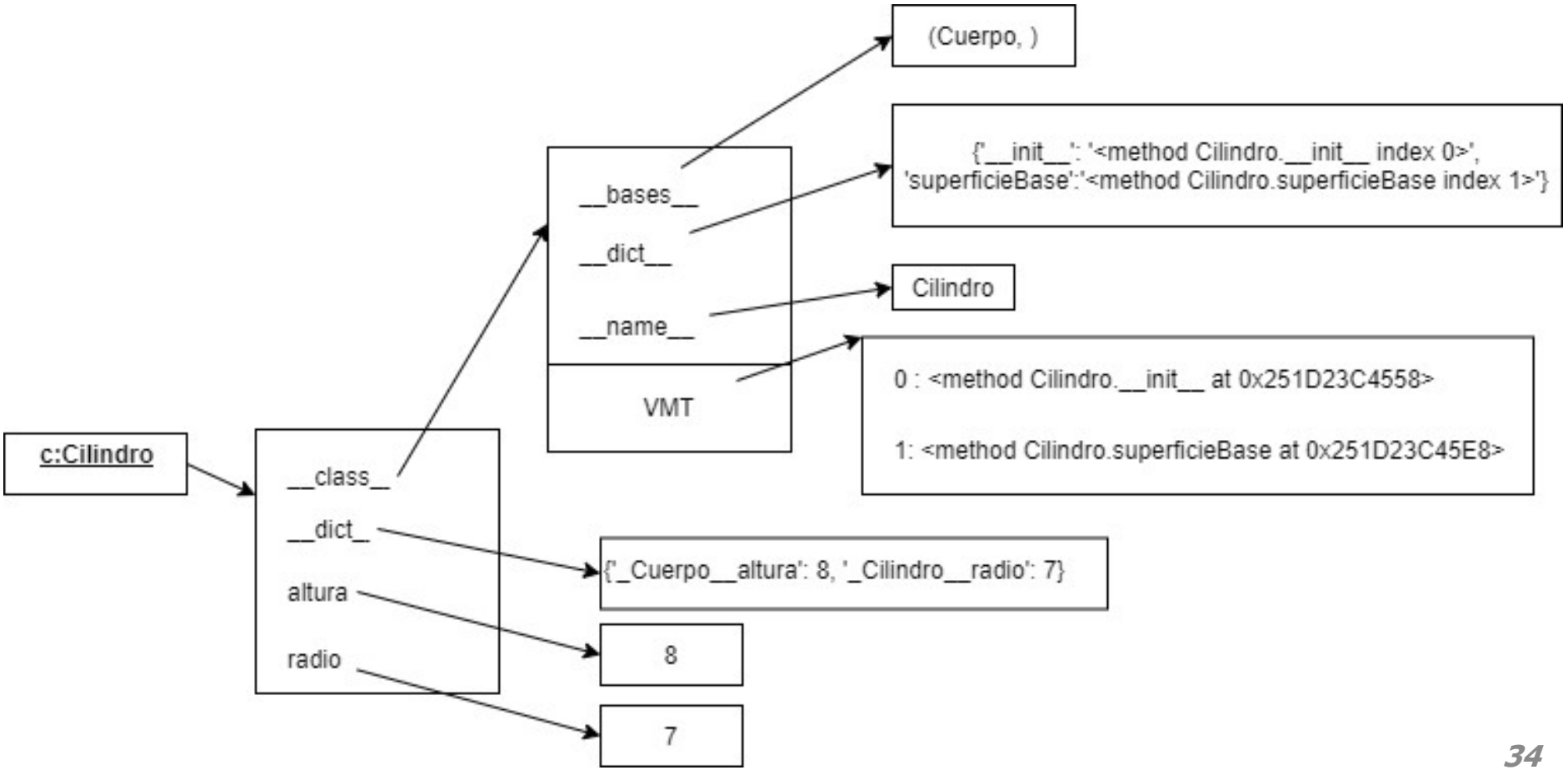


Polimorfismo: Vinculación Dinámica (II)

Si se crea un objeto de la clase Cilindro con la siguiente instrucción:

c = Cilindro(7,8)

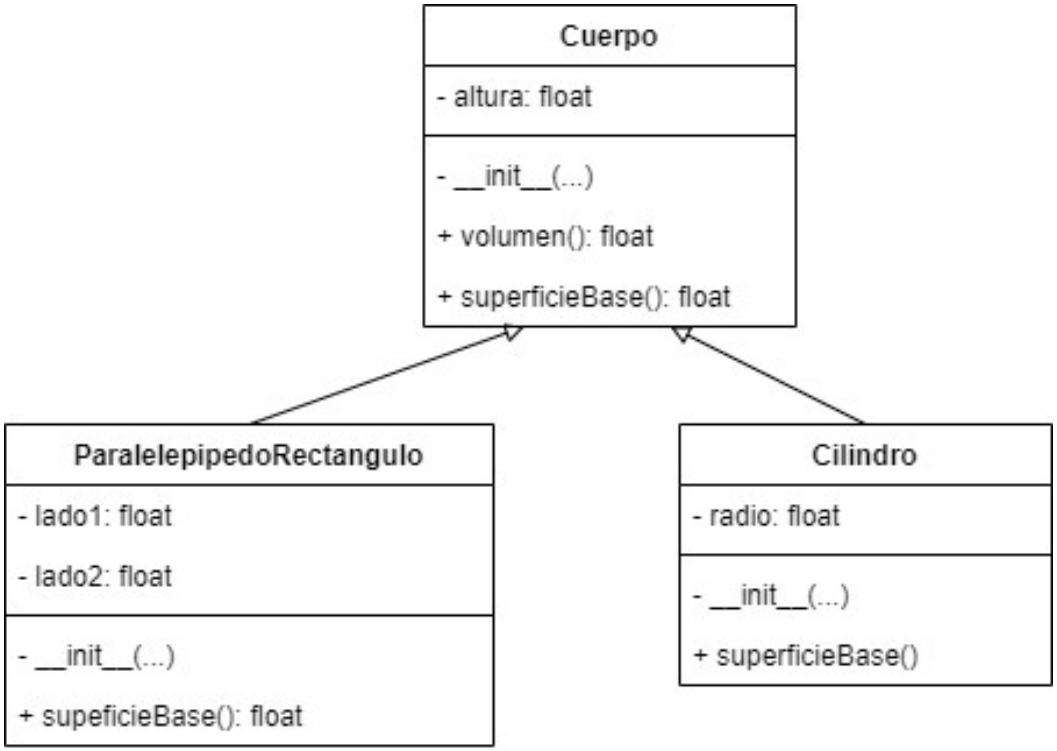
El diagrama de memoria que incluye la tabla de Métodos Virtuales, puede observarse en la figura



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.

Polimorfismo – Ejemplo (I)



Volumen es una función genérica.
Todo cuerpo tiene volumen, su cálculo se lleva a cabo a través de la fórmula
Volumen = Superficie de la Base x Altura

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

```
import numpy as np
import math
class Cuerpo:
    __altura: float
    def __init__(self, altura):
        self.__altura=altura
    def superficieBase():
        pass
    def volumen(self):
        return self.superficieBase()*self.__altura
    def getAltura(self):
        return self.__altura
```

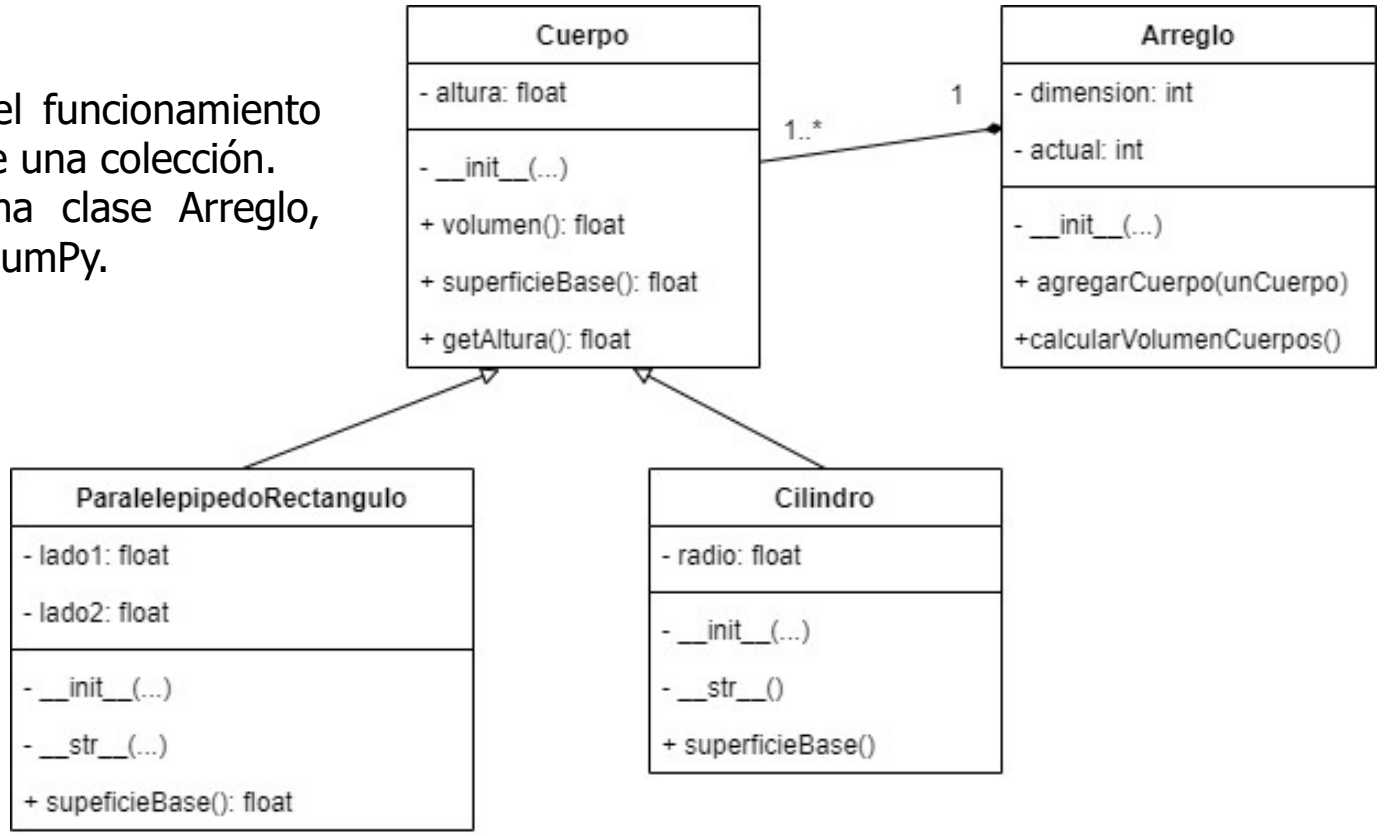
```
class Cilindro(Cuerpo):
    __radio: float
    def __init__(self, altura, radio):
        Cuerpo.__init__(self,altura)
        self.__radio=radio
    def __str__(self):
        cadena ='Cilindro, altura ={}, radio = {}'.format(self.getAltura(),
self.__radio)
        return cadena
    def superficieBase(self):
        return math.pi*self.__radio**2
```

Polimorfismo – Ejemplo (II)

```
class ParalelepipedoRectangulo(Cuerpo):
    __lado1: float
    __lado2: float
    def __init__(self, altura, lado1, lado2):
        Cuerpo.__init__(self,altura)
        self.__lado1=lado1
        self.__lado2=lado2
    def __str__(self):
        cadena = 'Paralelepípedo Rectángulo, altura = {}, lado a={}, lado b={}'.format(self.getAltura(),
self.__lado1, self.__lado2)
        return cadena
    def superficieBase(self):
        return self.__lado1*self.__lado2
```

Polimorfismo – Ejemplo (III)

La mejor forma para analizar el funcionamiento del polimorfismo, es a través de una colección. En el ejemplo, se utilizará una clase Arreglo, implementada con un arreglo NumPy.




```
class Arreglo:
    __dimension: int
    __actual: int
    __cuerpos: object
    def __init__(self, dimension=10):
        self.__cuerpos = np.empty(dimension, dtype=Cuerpo)
        self.__dimension=dimension
        self.__cantidad=0
    def agregarCuerpo(self, unCuerpo):
        self.__cuerpos[self.__actual]=unCuerpo
        self.__actual+=1
    def calcularVolumenCuerpos(self):
        for i in range(self.__actual):
            cuerpo=self.__cuerpos[i]
            print(str(cuerpo)+', Volumen = {0:7.2f}'.format(cuerpo.volumen()))
```

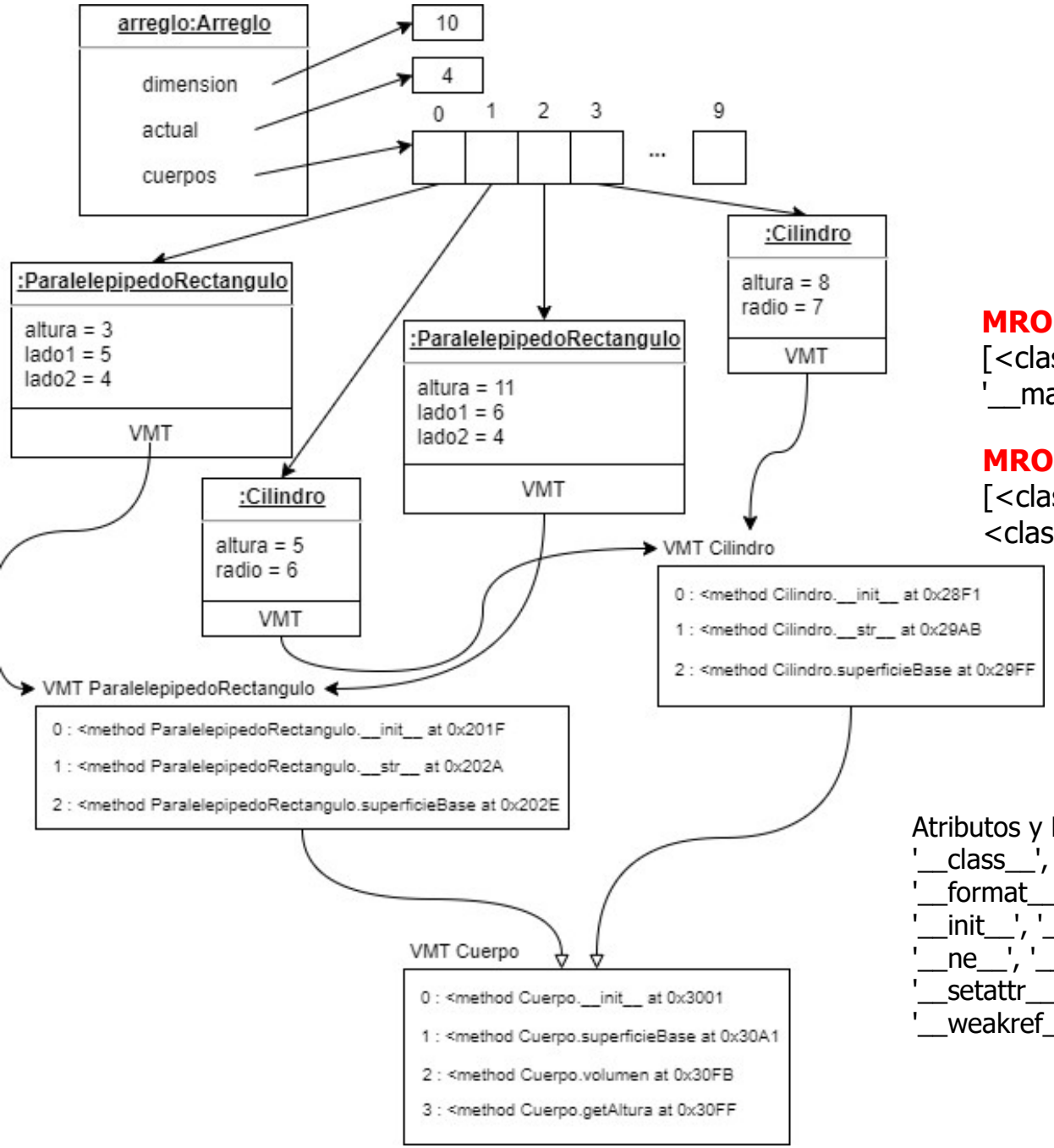
Polimorfismo – Ejemplo (IV)

```
def testPolimorfismo():
    arreglo = Arreglo()
    p = ParalelepipedoRectangulo(3,5,4)
    arreglo.agregarCuerpo(p)
    c = Cilindro(5,6)
    arreglo.agregarCuerpo(c)
    p = ParalelepipedoRectangulo(11,6,4)
    arreglo.agregarCuerpo(p)
    c = Cilindro(8,7)
    arreglo.agregarCuerpo(c)
    arreglo.calcularVolumenCuerpos()

if __name__=='__main__':
    testPolimorfismo()
```

Consola Python

```
Paralelepípedo Rectángulo, altura = 3, lado a=5, lado b=4, Volumen =   60.00
Cilindro, altura =5, radio = 6, Volumen =  565.49
Paralelepípedo Rectángulo, altura = 11, lado a=6, lado b=4, Volumen =  264.00
Cilindro, altura =8, radio = 7, Volumen = 1231.50
```

Polimorfismo – Ejemplo (V)



Cómo se ejecuta el mensaje:
__cuerpos[i].volumen()

Cómo se ejecuta el mensaje:
print(str(__cuerpos[i]))

MRO de la clase ParalelepipedoRectangulo
[<class '__main__.ParalelepipedoRectangulo'>, <class '__main__.Cuerpo'>, <class 'object'>]

MRO de la clase Cilindro
[<class '__main__.Cilindro'>, <class '__main__.Cuerpo'>, <class 'object'>]

Atributos y Métodos de Cilindro ['_Cilindro__radio', '_Cuerpo__altura', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'getAltura', 'superficieBase', 'volumen']

La Clase de los objetos referenciados (I)

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

La función correcta para saber si un objeto es instancia de una clase es la función isinstance(), ya que también funciona para subclases.

Retomando la clase arreglo, se necesita un método para saber cantidad de instancias de las clases ParalelepipedoRectangulo y de Cilindro, posee el arreglo.

```
def determinarClaseDeObjetos(self):
        cantidadP = 0
        cantidadC = 0
        for i in range(self.__actual):
            if isinstance(self.__cuerpos[i], ParalelepipedoRectangulo):
                cantidadP+=1
            else:
                if isinstance(self.__cuerpos[i], Cilindro):
                    cantidadC+=1
        print('Cantidad de Paralelepidedos Rectángulo: ', cantidadP)
        print('Cantidad de Cilindros: ', cantidadC)
```

La Clase de los objetos referenciados (II)

Qué resultado produce el siguiente código:

```
def determinarClaseDeObjetos(self):
        cuenta=0
        cantidadP = 0
        cantidadC = 0
        for i in range(self.__actual):
            if isinstance(self.__cuerpos[i], Cuerpo):
                cuenta+=1
            else:
                if isinstance(self.__cuerpos[i], ParalelepipedoRectangulo):
                    cantidadP+=1
                else:
                    if isinstance(self.__cuerpos[i], Cilindro):
                        cantidadC+=1
        print('Cuenta: ', cuenta)
        print('Cantidad de Paralelepidedos Rectángulo: ', cantidadP)
        print('Cantidad de Cilindros: ', cantidadC)
```

Errores en un programa

No importa la habilidad que se tenga como programador, eventualmente se cometerán errores de codificación.
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.

Errores en tiempo de ejecución

```
print(a)
```

NameError: name 'a' is not defined

```
print(1+'abc')
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

```
dividendo = 4
divisor = int(input('Ingrese un número: '))
print('El resultado de la división es: ',
dividendo/divisor)
```

ZeroDivisionError: division by zero

```
lista = [5, 7, 11, 21]
    for i in range(15):
        print(lista[i])
```

IndexError: list index out of range

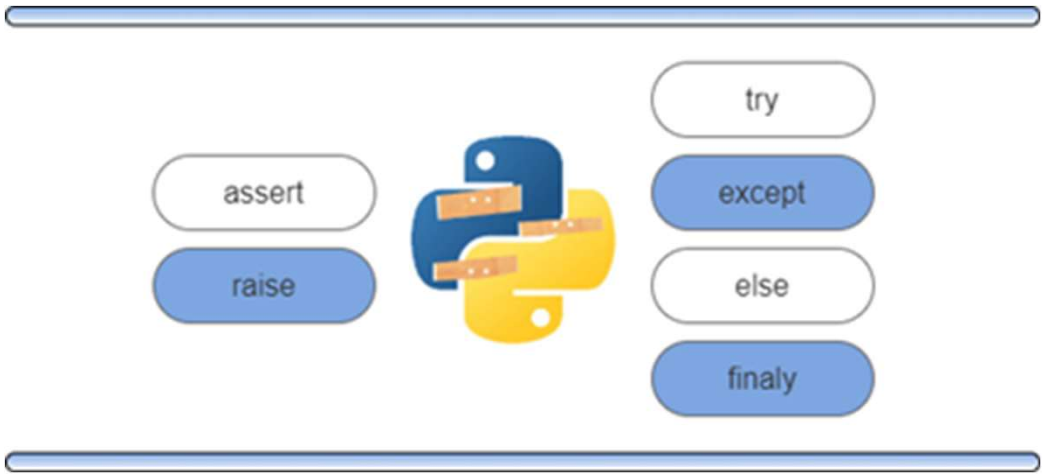
Python, no sólo indica que ocurrió un error, además indica la excepción que se lanzó por el error en tiempo de ejecución, a la hora de corregir el código, la excepción da una idea de lo que salió mal.

Mark - Chapter 34 - Páginas 1093 a 1115

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:

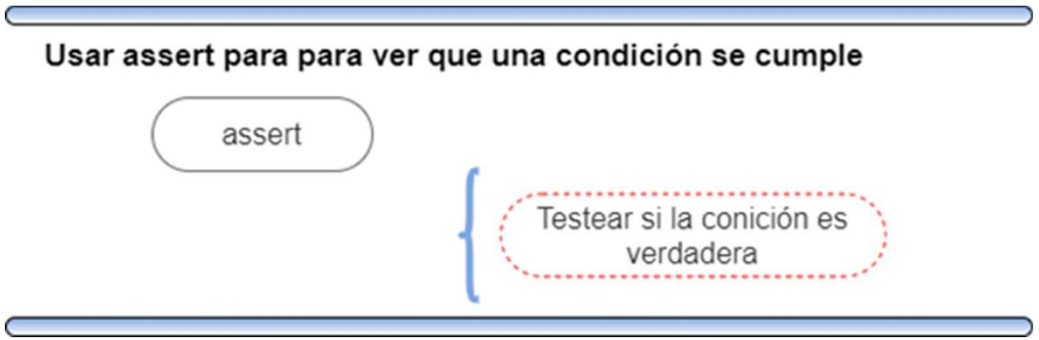
- assert: para probar si una afirmación es verdadera o falsa.
- raise: para forzar el lanzamiento de una excepción.
- try-except-else-finally: bloque para manejar y capturar excepciones, y determinar qué hacer cuando sea capturada una excepción.



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.

```
class Character:
    def __init__(self, character,
                 bold=False, italic=False,
underline=False):
        assert len(character) == 1, "Debe ser
un caracter"
        self.__character = character
        self.__bold = bold
        self.__italic = italic
        self.__underline = underline
    def __str__(self):
        bold = "*" if self.__bold else ''
        italic = "/" if self.__italic else ''
        underline = "_" if self.__underline else ''
        return bold + italic + underline +
self.__character
def testCharacter():
    c = Character('ab',True,True,True)
    print(c)
if __name__=='__main__':
    testCharacter()
```



Consola Python

```
assert len(character) == 1

AssertionError: Debe ser un caracter
```

La utilidad principal de assert, es la depuración de un programa.

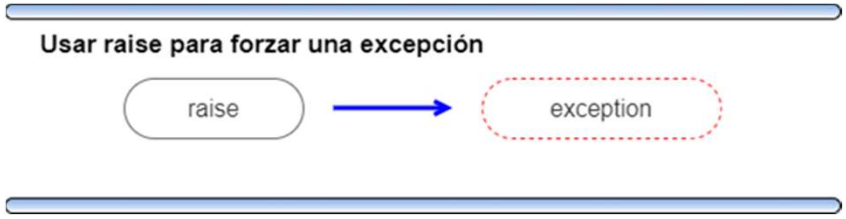
```
import sys
def chequeoSistemaOperativo():
    assert ('linux' in sys.platform), "Éste código sólo corre en Linux."
if __name__=='__main__':
    chequeoSistemaOperativo()
```

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.

```
class Cliente:
    __nombre: str
    __apellido: str
    __dni: int
    __numeroTarjeta: int
    __saldoAnterior: float
    def __init__(self, nomb, ape, dni, numtar, saldant):
        self.__nombre = nomb
        self.__apellido = ape
        self.__dni = dni
        self.__numeroTarjeta = numtar
        self.__saldoAnterior = saldant
    def getNombre(self):
        return self.__nombre
    def getApellido(self):
        return self.__apellido
    def getDni(self):
        return self.__dni
    def getNumeroTarjeta(self):
        return self.__numeroTarjeta
    def getSaldoAnterior(self):
        return self.__saldoAnterior
    def actualizar_saldo(self, nuevo_saldo):
        self.__saldoAnterior = nuevo_saldo
```

```
from claseCliente import Cliente
class GestorClientes:
    __listaClientes: list
    def __init__(self):
        self.__listaClientes=[]
    def agregarCliente(self, unCliente):
        if isinstance(unCliente, Cliente):
            self.__listaClientes.append(unCliente)
        else:
            raise TypeError
```

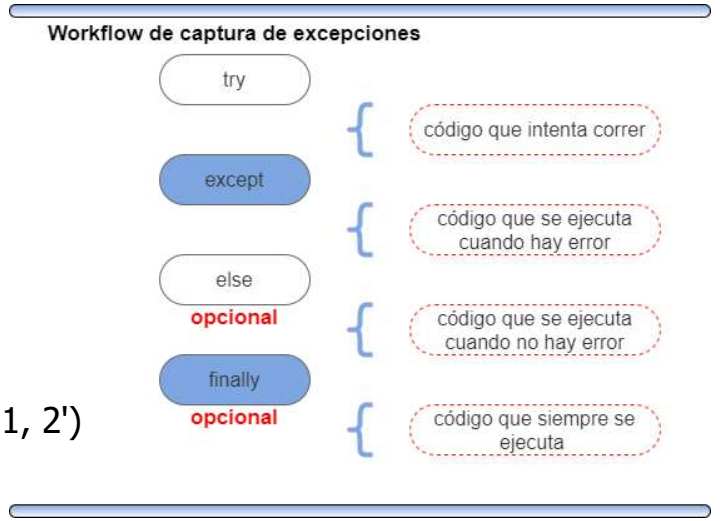



```
from claseGestorCliente import GestorClientes
from claseCliente import Cliente
def test1():
    gestorCliente=GestorClientes()
    unObjetoCliente = Cliente("Hernan", "Castro", 20333111,
5551, 28000)
    try:
        gestorCliente.agregarCliente(unObjetoCliente)
    except TypeError:
        print("Error de tipos")
def test2():
    gestorCliente=GestorClientes()
    try:
        gestorCliente.agregarCliente("agregando un string")
    except TypeError:
        print("Error de tipos")

if __name__=="__main__":
    test1()
    test2()
```

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.

```
def testBloqueTryExcept():
    try:
        num1, num2 = eval(input('Ingrese dos números separados por coma: '))
        resultado = num1 / num2
        print('El resultado es {0:10.8f}'.format(resultado))
    except ZeroDivisionError:
        print('La división por cero es un error!!!')

    except SyntaxError:
        print('Error de sintaxis, ingrese los números separados por coma, ejemplo 1, 2')
    except:
        print('Entrada errónea')
    else:
        print('No hubieron excepciones')
    finally:
        print('Siempre se ejecuta esta sección')
if __name__=='__main__':
    testBloqueTryExcept()
```



Consola Python

```
Ingrese dos números separados por coma: 4,5
El resultado es 0.80000000
No hubieron excepciones
Siempre se ejecuta esta sección
```

Consola Python

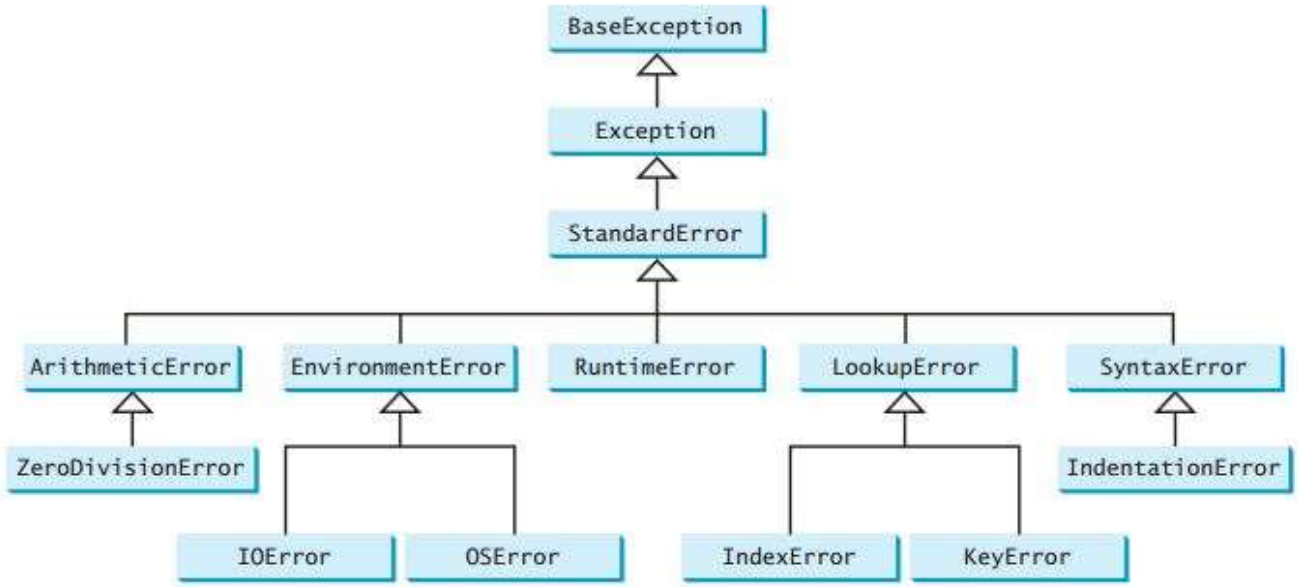
```
Ingrese dos números separados por coma: 4,
Entrada errónea
Siempre se ejecuta esta sección
```

Consola Python

```
Ingrese dos números separados por coma: 4 5
Error de sintaxis, ingrese los números separados por coma, ejemplo 1, 2
Siempre se ejecuta esta sección
```

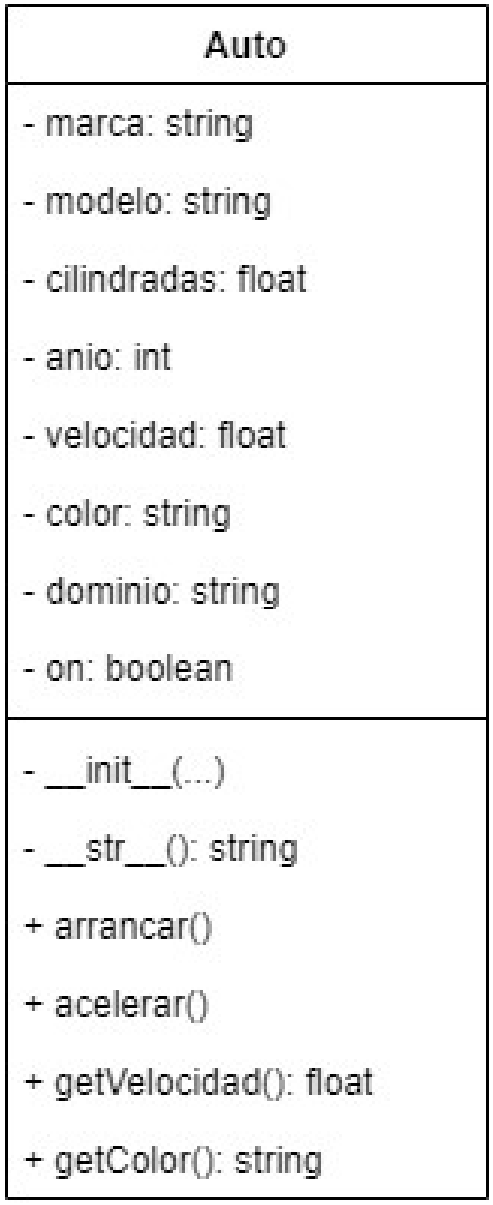
Definiendo nuevas Excepciones (I)

Se pueden definir nuevas excepciones, extendiendo la clase BaseException o alguna de sus subclases.
En la siguiente jerarquía puede observarse un subconjunto de clases de la jerarquía de excepciones pre definida en el lenguaje Python.



Definiendo nuevas Excepciones (II)

Dada la clase Auto, cuyo diagrama UML es el siguiente:

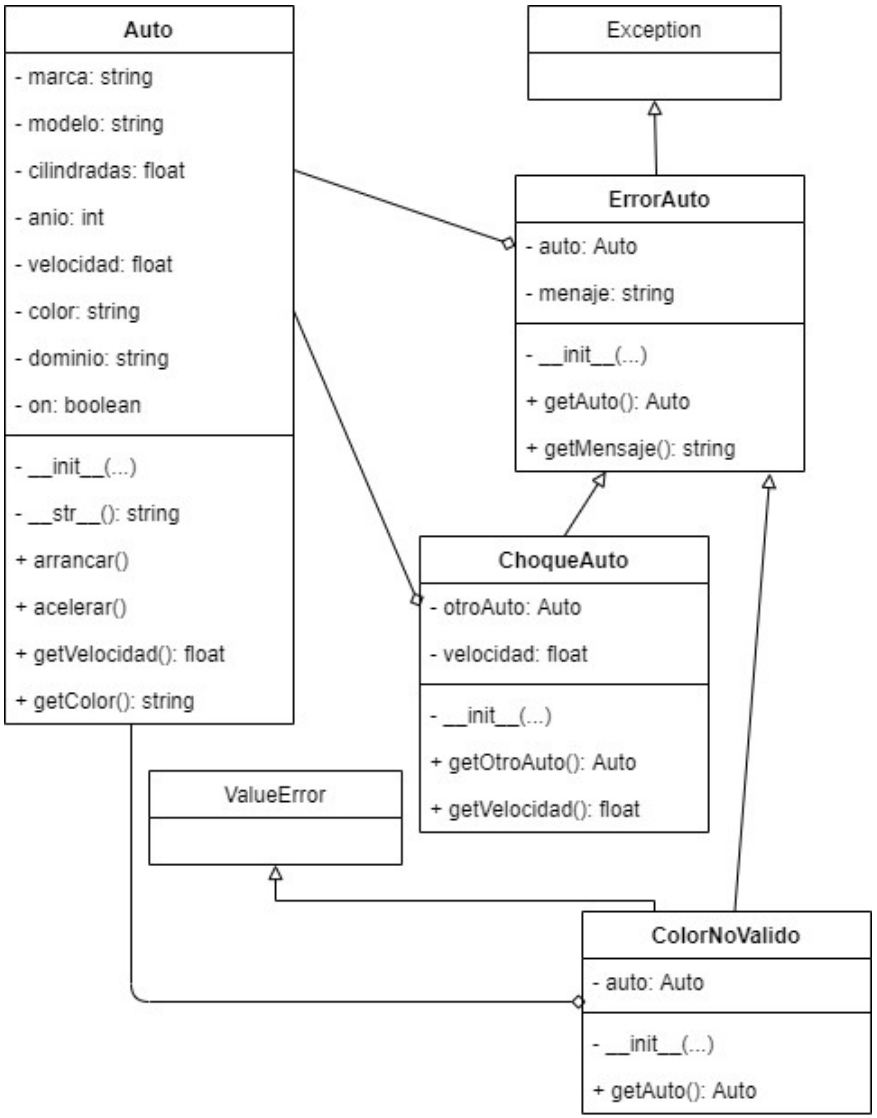


La clase tiene los métodos:
+ **arrancar()**, este método permite poner en marcha el vehículo, en caso de que el auto esté arrancado, on=True, y reciba el mensaje arrancar(), se debe lanzar la excepción **ErrorAuto** con el mensaje: **Auto ya arrancado**.
+ **acelerar()**: incrementa la velocidad del vehículo en 10 km/h.
+ **getVelocidad()**: devuelve la velocidad actual del auto.
+ **getColor()**: devuelve el color del auto.

Reglas de negocio:
Cuando se produce un accidente, se lanza la excepción **ChoqueAuto**, que verifica si la velocidad del vehículo es superior a 30 km/h, en caso afirmativo, la excepción muestra los autos involucrados en el accidente y aconseja llamar al 911.
Cuando se verifica que un auto no tiene un color válido, se lanza la excepción **ColorNoValido**, que muestra el mensaje: **El auto debe tener un color**.

Detalles de implementación de las excepciones creadas:
Todas las clases excepción creadas, deben almacenar una referencia al auto que recibió el mensaje y el mensaje propiamente dicho.
La clase **ChoqueAuto**, además, debe agregar el otro auto involucrado en el choque y la velocidad a la estaba circulando el auto principal del choque.

Definiendo nuevas Excepciones (III)




```
class Auto(object):
    __marca: str
    __modelo: str
    __cilindradas: float
    __anio: int
    __velocidad: int
    __color: str
    __dominio: str
    __on: bool
    def __init__(self, marca, modelo, cilindradas, anio, dominio, color=None):
        self.__marca=marca
        self.__modelo=modelo
        self.__cilindradas=cilindradas
        self.__anio=anio
        self.__velocidad=0
        self.__dominio=dominio
        self.__color=color
    def __str__(self):
        cadena='Auto\n'
        cadena+='Marca {}, modelo {}\n'.format(self.__marca, self.__modelo)
        cadena+='Cilindradas: {}\n'.format(self.__cilindradas)
        cadena+='Dominio: {}\n'.format(self.__dominio)
        cadena+='Año de fabricación: {}, color {}'.format(self.__anio,
self.__color)
        return cadena
    def arrancar(self):
        if self.__on==False:
            self.__velocidad = 10
            self.__on=True
        else:
            raise ErrorAuto(self,'Auto ya arrancado')
    def acelerar(self):
        self.__velocidad+=10
    def getVelocidad(self):
        return self.__velocidad
    def getColor(self):
        return self.__color
```

Definiendo nuevas Excepciones (IV)

```
class ErrorAuto(Exception):
    __auto=None
    __mensaje=None
    '''Excepción básica para errores lanzados por los autos'''
    def __init__(self, auto, mensaje=None):
        if mensaje is None:
            # Set algún valor para el mensaje de error
            mensaje='Un error ha ocurrido con el auto  %s' % str(auto)
        super(ErrorAuto, self).__init__(mensaje)
        self.__auto = auto
        self.__mensaje='ErrorAuto: '+mensaje
    def getAuto(self):
        return self.__auto
    def getMensaje(self):
        return self.__mensaje
class ChoqueAuto(ErrorAuto):
    '''Cuando se maneja demasiado rápido'''
    __velocidad=0
    __otroAuto=None
    def __init__(self, auto, otroAuto, velocidad):
        super().__init__(
            auto, mensaje="ChoqueAuto: auto chocó con %s a la velocidad %d km/h" %
(otroAuto, velocidad))
        self.__velocidad = velocidad
        self.__otroAuto = otroAuto
    def getOtroAuto(self):
        return self.__otroAuto
    def getVelocidad(self):
        return self.__velocidad
class ColorNoValido(ErrorAuto,ValueError):
    '''Lanzado cuando el atributo color de un auto no tiene un valor válido'''
    def __init__(self, auto, mensaje):
        super().__init__(auto,'ColorNoValido '+mensaje)
```

Definiendo nuevas Excepciones (V)

```
def manejarAuto(auto):
    auto.arrancar()
    #auto.arrancar()
    for i in range(6):
        auto.acelerar()
    choque = True
    if choque:
        velocidad=auto.getVelocidad()
        otroAuto=Auto('Toyota','Corolla',2.0,'HXT 200',2009)
        raise ChoqueAuto(auto,otroAuto,velocidad)
def llamarAl911():
    print('Llamar al 911')
def verColorAuto(auto):
    color = auto.getColor()
    if color==None or color=='':
        raise ColorNoValido(auto, 'El auto debe tener un color')
```

Consola Python

```
ERROR
-----
Auto
Marca Renault, modelo Kwid
Cilindradas: 1.0
Dominio: AC 213 FG
Año de fabricación: 2019, color None
--------------
ErrorAuto: ColorNoValido El auto debe tener un color
Accidente
---------
Llamar al 911
Auto
Marca Renault, modelo Kwid
Cilindradas: 1.0
Dominio: AC 213 FG
Año de fabricación: 2019, color None
ErrorAuto: ChoqueAuto: auto chocó con Auto
Marca Toyota, modelo Corolla
Cilindradas: 2.0
Dominio: HXT 200
Año de fabricación: 2009, color None a la velocidad 70 km/h
```

```
def testExcepciones():
    unAuto=Auto('Renault','Kwid',1.0,'AC 213 FG',2019)
    try:
        verColorAuto(unAuto)
    except ColorNoValido as e:
        print('ERROR')
        print('-----')
        print(e.getAuto())
        print('--------------')
        print(e.getMensaje())
    try:
        manejarAuto(unAuto)
    except ChoqueAuto as e:
        if e.getVelocidad() >= 30:
            print('Accidente')
            print('---------')
            llamarAl911()
            print(e.getAuto())
            print(e.getMensaje())
    except ErrorAuto as e:
        print('ERROR')
        print('-----')
        print(e.getAuto())
        print('---------------')
        print(e.getMensaje())
if __name__=='__main__':
    testExcepciones()
```

Consejos sobre excepciones

- 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.

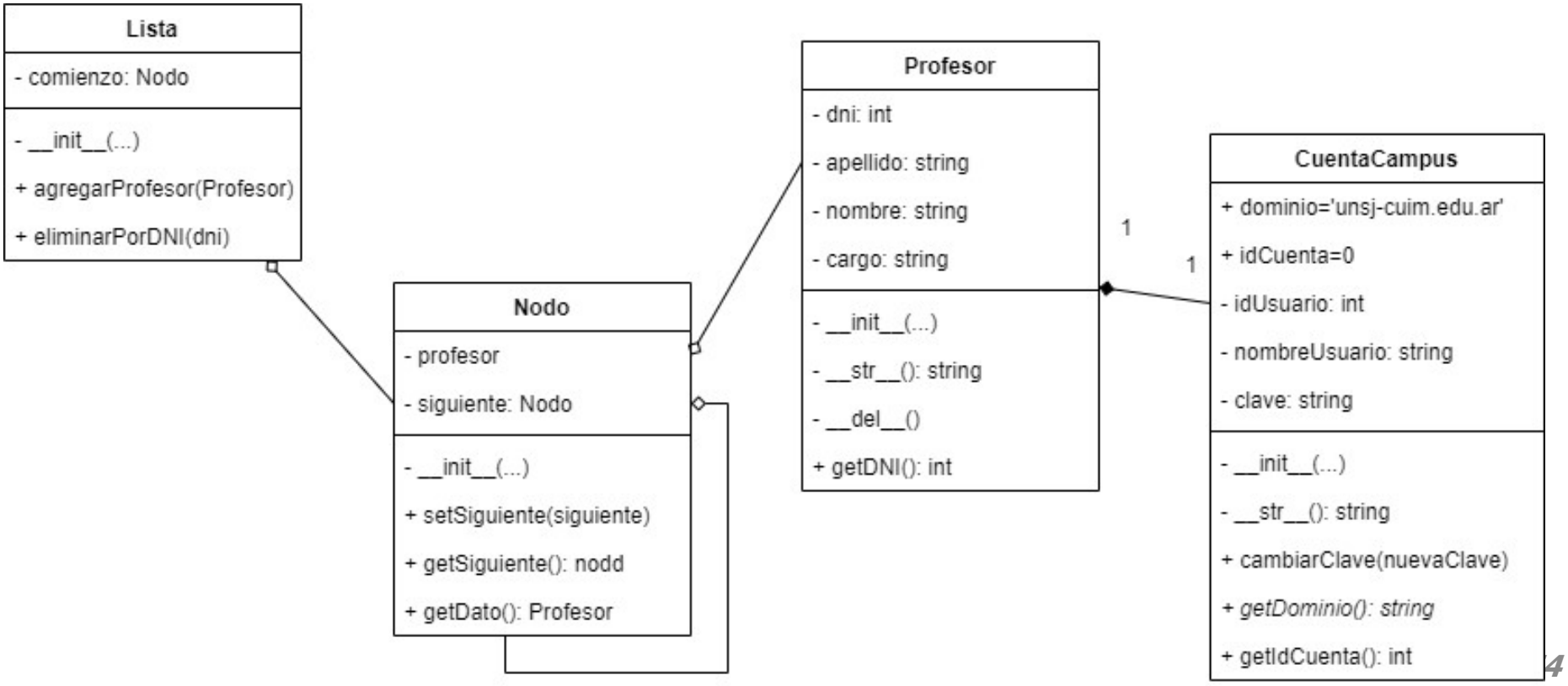
LAS EXECPCIONES SON PARA CONTROLAR SITUACIONES EXCEPCIONALES

LISTAS DEFINIDAS POR EL PROGRAMADOR

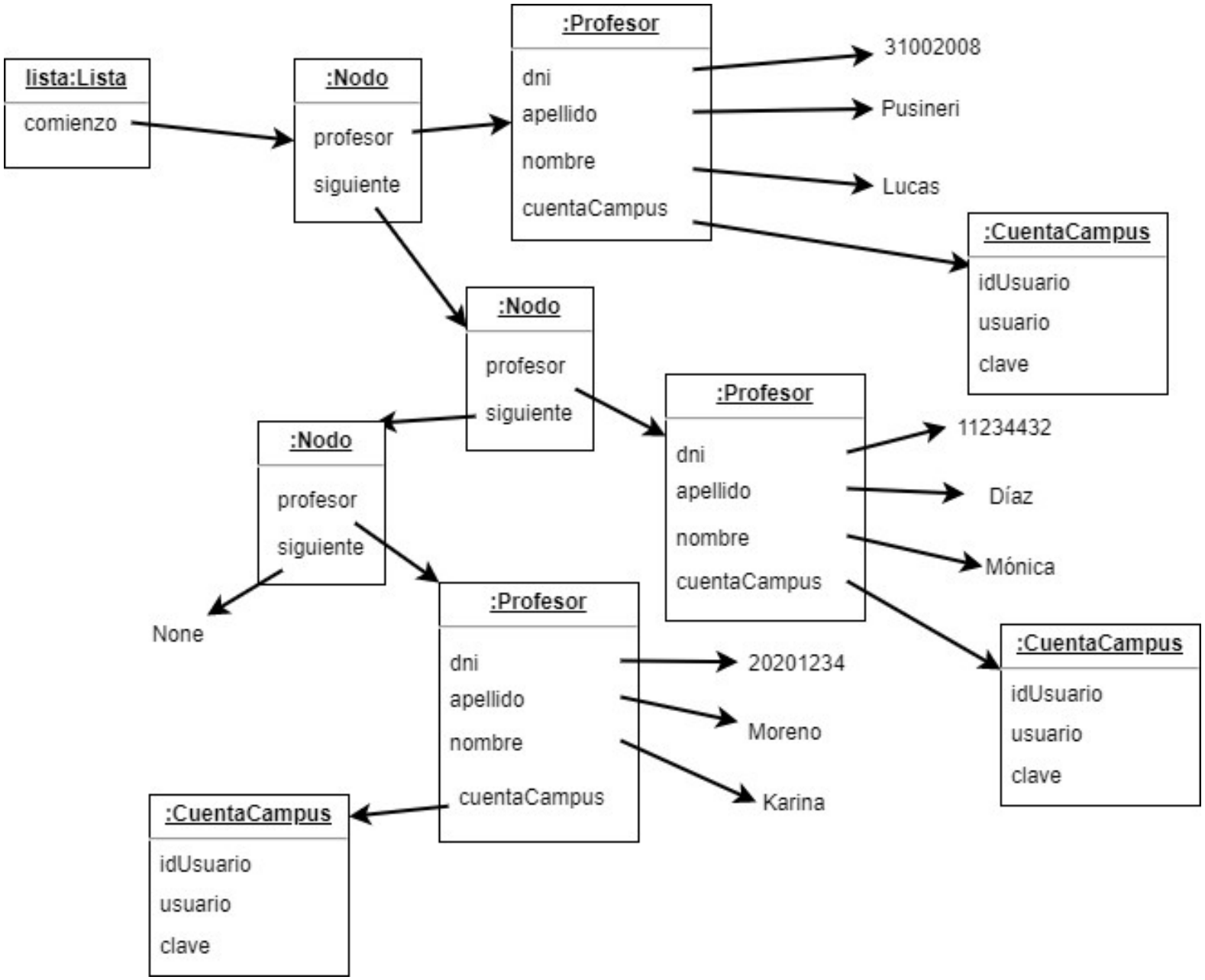
La consultora de desarrollo de software «In-PhOne» necesita implementar una lista enlazada para administrar los profesores que dictan clases on line a través de un campus. Le solicita a usted el desarrollo de las clases necesarias para implementar la lista enlazada que permita:

1: Agregar un Profesor a la lista.
2: Mostrar datos de todos los Profesores.
3: Dado el número de DNI de un profesor, si se encuentra en la lista, eliminarlo de la misma.

Nota:
Utilizar la clase **Profesor** anteriormente, si es necesario, agregar los métodos que crea conveniente



Esquema de la lista enlazada de alumnos



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))
```

Test de Lista

```
def testLista():
    lista = Lista()
    profesor = Profesor(20201234, 'Moreno', 'Karina')
    lista.agregarProfesor(profesor)
    profesor = Profesor(11234432, 'Díaz', 'Mónica')
    lista.agregarProfesor(profesor)
    profesor = Profesor(31002008, 'Pusineri', 'Lucas')
    lista.agregarProfesor(profesor)
    lista.listarDatosProfesores()
    lista.eliminarPorDNI(11234432)
    print('Luego de Borrar')
    lista.listarDatosProfesores()
if __name__=='__main__':
    testLista()
```

Consola Python

```
Profesor:
Apellido y nombre: Pusineri, Lucas
Usuario: lucaspusineri@unsj-cuim.edu.ar
Clave 31002008
Profesor:
Apellido y nombre: Díaz, Mónica
Usuario: mónicadíaz@unsj-cuim.edu.ar
Clave 11234432
Profesor:
Apellido y nombre: Moreno, Karina
Usuario: karinamoreno@unsj-cuim.edu.ar
Clave 20201234
Encontrado y eliminado:
Profesor:
Apellido y nombre: Díaz, Mónica
Usuario: mónicadíaz@unsj-cuim.edu.ar
Clave 11234432
Borrando cuenta de usuario....
Luego de Borrar
Profesor:
Apellido y nombre: Pusineri, Lucas
Usuario: lucaspusineri@unsj-cuim.edu.ar
Clave 31002008
Profesor:
Apellido y nombre: Moreno, Karina
Usuario: karinamoreno@unsj-cuim.edu.ar
Clave 20201234
Borrando cuenta de usuario....
Borrando cuenta de usuario....
```

Cómo hacer para que la lista responda al código siguiente???

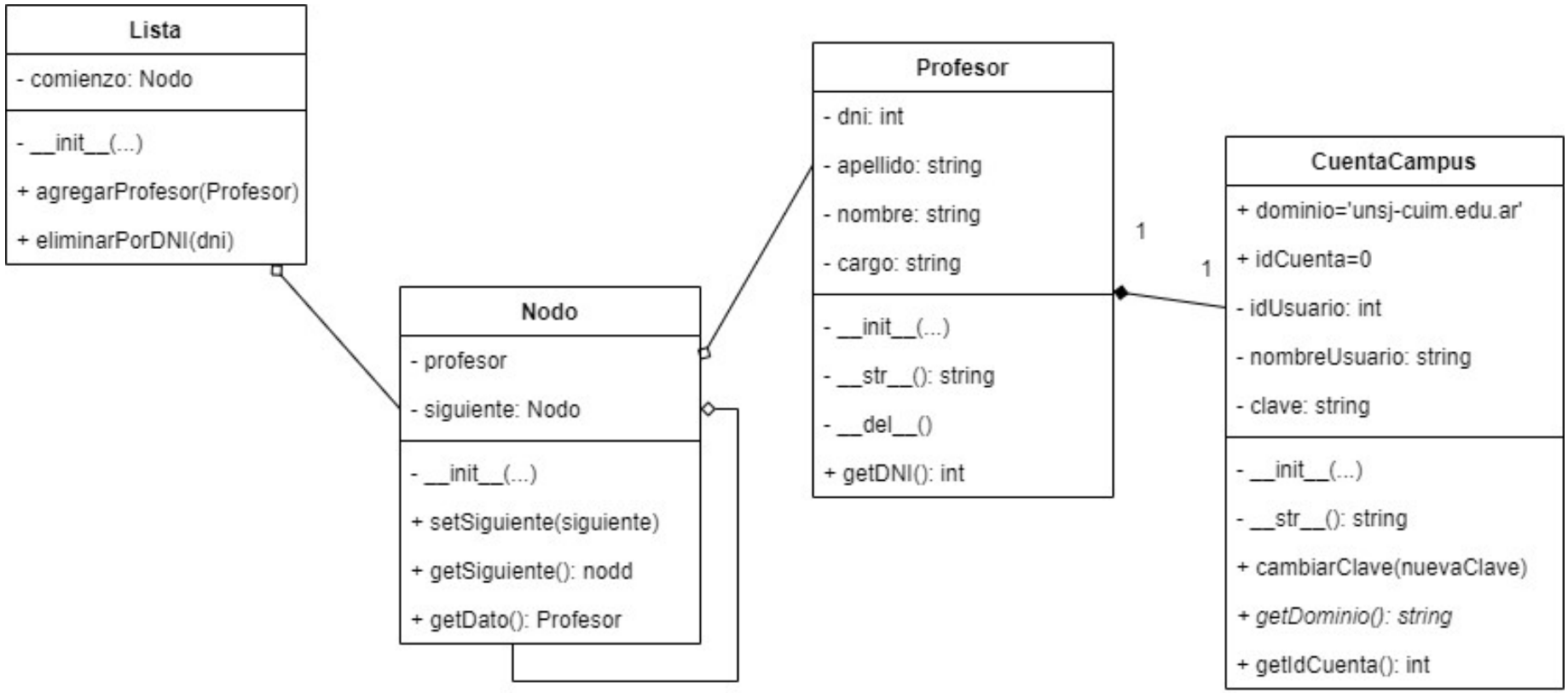
```
for dato in lista:
    print('Datos del',dato)
```

Iterador sobre la clase Lista (I)

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.

Mark Lutz – Chapter 14 – Páginas 416 a 424 y Chapter 30 – Páginas 895 a 909

Iterador sobre la clase Lista (II)



Iterador sobre la clase Lista (III)

Clase Lista

```
class Lista:
    __comienzo: Nodo
    __actual: Nodo
    __indice: int
    __tope: int
    def __init__(self):
        self.__comienzo=None
        self.__actual=None
    def __iter__(self):
        return self
    def __next__(self):
        if self.__indice==self.__tope:
            self.__actual=self.__comienzo
            self.__indice=0
            raise StopIteration
        else:
            self.__indice+=1
            dato = self.__actual.getDato()
            self.__actual=self.__actual.getSiguiente()
            return dato
    def agregarProfesor(self, profesor):
        nodo = Nodo(profesor)
        nodo.setSiguiente(self.__comienzo)
        self.__comienzo=nodo
        self.__actual=nodo
        self.__tope+=1
```

Iterador sobre la clase Lista (IV)

Clase Lista - Continuación

```
def eliminarPorDNI(self, dni):
        aux=self.__comienzo
        encontrado = False
        if aux.getDato().getDNI()==dni:
            encontrado=True
            print('Encontrado y eliminado:\n'+str(aux.getDato()))
            self.__comienzo = aux.getSiguiente()
            self.__tope-=1
            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 y eliminado:\n'+str(aux.getDato()))
                ant.setSiguiente(aux.getSiguiente())
                self.__tope-=1
                del aux
            else:
                print('El DNI {}, no está en la lista'.format(dni))
```

Iterador sobre la clase Lista (V)

```
def testLista():
    lista = Lista()
    profesor = Profesor(20201234, 'Moreno', 'Karina')
    lista.agregarProfesor(profesor)
    profesor = Profesor(11234432, 'Díaz', 'Mónica')
    lista.agregarProfesor(profesor)
    profesor = Profesor(31002008, 'Pusineri', 'Lucas')
    lista.agregarProfesor(profesor)
    for dato in lista:
        print('Datos del',dato)
    lista.eliminarPorDNI(11234432)
    print('Luego de Borrar')
    for dato in lista:
        print('Datos del',dato)
if __name__=='__main__':
    testLista()
```

Consola Python

```
Datos del Profesor:
Apellido y nombre: Pusineri, Lucas
Usuario: lucaspusineri@unsj-cuim.edu.ar
Clave 31002008
Datos del Profesor:
Apellido y nombre: Díaz, Mónica
Usuario: mónicadíaz@unsj-cuim.edu.ar
Clave 11234432
Datos del Profesor:
Apellido y nombre: Moreno, Karina
Usuario: karinamoreno@unsj-cuim.edu.ar
Clave 20201234
Encontrado y eliminado:
Profesor:
Apellido y nombre: Díaz, Mónica
Usuario: mónicadíaz@unsj-cuim.edu.ar
Clave 11234432
Borrando cuenta de usuario....
Luego de Borrar
Datos del Profesor:
Apellido y nombre: Pusineri, Lucas
Usuario: lucaspusineri@unsj-cuim.edu.ar
Clave 31002008
Datos del Profesor:
Apellido y nombre: Moreno, Karina
Usuario: karinamoreno@unsj-cuim.edu.ar
Clave 20201234
Borrando cuenta de usuario....
Borrando cuenta de usuario....
```

Testing (I)

•Comprender los conceptos de testing es esencial para garantizar la calidad del software. Los testers desempeñan un papel crucial al identificar y corregir errores y fallos antes de que los usuarios finales los experimenten.

•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.

•Un software confiable y libre de errores es fundamental para la satisfacción del cliente. Los testers ayudan a crear productos que funcionen correctamente y cumplan con las expectativas de los usuarios.

•El testing de seguridad es vital en un mundo donde las amenazas cibernéticas son constantes. Los testers de seguridad identifican vulnerabilidades y protegen la integridad de los datos y la información confidencial.

•En industrias reguladas, como la salud o las finanzas, el testing de software es esencial para cumplir con los requisitos legales y las normativas de seguridad.

•El testing no solo detecta errores, sino que también proporciona información valiosa para mejorar los procesos de desarrollo.

•En un mercado altamente competitivo, tener un software de alta calidad y confiabilidad es una ventaja competitiva. Los testers contribuyen a mantener a las empresas relevantes y competitivas.

Testing (II)

Ejecutar el programa, permite corregir los errores, es una forma burda de prueba.
Los programadores Python son capaces escribir algunas líneas de código y ejecutar el programa para asegurarse de que esas líneas funcionan como se espera que funcionen.
Pero cambiar algunas líneas de código puede afectar partes del programa que el desarrollador no se había dado cuenta de que se verá influenciado por los cambios y, por lo tanto, no se probarán.
Además, 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.

Testing (III)

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.

Testing (IV) – Desarrollo basado en pruebas

Mantra 1: "Escriba las pruebas primero"
Mantra 2: "código no probado es código roto" (sugiere que solo el código no escrito debe estar sin probar)

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, decir: "**Hmm…, esto parece funcionar, no se tiene que escribir ninguna prueba para esto. …fue solo un pequeño cambio nada podría haberse roto…**". Si la prueba ya está escrita antes de escribir el código, se sabrá exactamente cuando 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.

Testing (V) – 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).
Normalmente, las pruebas establecen algunos valores en un objeto, luego ejecutan un método, y usan los métodos de comparación incorporados para asegurarse de que se hayan calculado los resultados correctos.

Testing (VI)

```
import unittest

class TestNumeros(unittest.TestCase):
    def test_int_float(self):
        self.assertEqual(1,1.0)
    def test_int_str(self):
        self.assertEqual(1,'1')

if __name__ == '__main__':
    unittest.main()
```

Este código crea una subclase de TestCase, define:
* un método ***test_int_float()***, que invoca al método TestCase.assertEqual(1, 1.0). Este método podrá tener éxito si 1 == 1.0, o podrá fallar si 1 != 1.0.
* Un método test_int_str(), que invoca al método TestCase.assertEqual(1,'1'). Este método tiene éxito si 1=='1', o falla si 1 != '1'.

Si se ejecuta este código, la función **main** de unittest producirá la siguiente salida:

```
.F
======================================================================
FAIL: test_int_str (__main__.TestNumeros.test_int_str)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\Users\Usuario\Documents\2024\POO\Unidad 3\codigo\ejemploTest.py", line 7, in test_int_str
    self.assertEqual(1,'1')
AssertionError: 1 != '1'

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Testing (VII)

```
import unittest

class TestNumeros(unittest.TestCase):
    def test_int_float(self):
        self.assertEqual(1,1.0)
    def test_int_str(self):
        self.assertEqual(1,'1')

if __name__ == '__main__':
    unittest.main()
```

```
.F
======================================================================
FAIL: test_int_str (__main__.TestNumeros)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:/Users/morte/Documents/2021/POO/Unidad 2 -
Python/Código/Testing/venv/Scripts/claseTestNumeros.py", line 7, in test_int_str
    self.assertEqual(1,'1')
AssertionError: 1 != '1'
```

Testing (VIII)

Se pueden tener tantos métodos de prueba en una clase o subclase, TestCase, como sean necesarios.
Cada prueba debe ser completamente independiente de otras pruebas.
Los resultados de una prueba previa no deberían tener ningún impacto en la prueba actual.
La clave para escribir buenas pruebas unitarias es mantener cada método de prueba lo más breve posible, probando una pequeña unidad de código en cada caso de prueba.

Testing (IX) – Métodos Assert

Una aserción, en Python, es en la práctica similar a una aserción en el lenguaje cotidiano. Cuando se hace una afirmación, se dice hay algo que no está necesariamente probado, pero que se cree que es verdad.
En Python, una aserción es un concepto similar. 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, el chequeo se hace mediante el uso de métodos de aserción (assert) de TestCase.

Testing (X)

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.

Testing (XI) – setUp y tearDown

Después de escribir algunas pruebas pequeñas, puede darse la situación donde el programador se encuentre que tiene que hacer el mismo código de configuración para varias pruebas relacionadas.

```
from collections import defaultdict
class ListaEstadistica(list):
    def mediaAritmetica(self):
        return sum(self) / len(self)
    def mediana(self):
        retorno=0.0
        if len(self) % 2:
            retorno = self[int(len(self) / 2)]
        else:
            idx = int(len(self) / 2)
            retorno = (self[idx] + self[idx-1]) / 2
        return retorno
    def moda(self):
        freqs = defaultdict(int)
        for item in self:
            freqs[item] += 1
        moda_freq = max(freqs.values())
        modas = []
        for item, valor in freqs.items():
            if valor == moda_freq:
                modas.append(item)
        return modas
```

Se van a querer probar situaciones con cada uno de estos tres métodos que tienen entradas muy similares; se querrá ver qué pasa con listas vacías o con listas que contienen valores no numéricos o con listas que contienen un conjunto de datos normal. Se puede usar el método **setUp()** de la clase TestCase 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.

Testing (XII)

```
from claseListaEstadistica import ListaEstadistica
import unittest
class TestValidInputs(unittest.TestCase):
    __listaEstadistica: list
    def setUp(self):
        self.__listaEstadistica = ListaEstadistica([1,2,2,3,3,4])
    def test_media(self):
        self.assertEqual(self.__listaEstadistica.mediaAritmetica(),2.5)
    def test_mediana(self):
        self.assertEqual(self.__listaEstadistica.mediana(), 2.5)
        self.__listaEstadistica.append(4)
        self.assertEqual(self.__listaEstadistica.mediana(), 3)
    def test_moda(self):
        self.assertEqual(self.__listaEstadistica.moda(), [2,3])
        self.__listaEstadistica.remove(2)
        self.assertEqual(self.__listaEstadistica.moda(), [3])

if __name__=='__main__':
    unittest.main()
```

Testing (XIII)

El Banco Sureño SA, lo contrata para desarrollar el módulo Caja de Ahorro, una caja de ahorro, posee los siguientes datos: numero, cbu, saldo, importe extraído (a las 0:00h de cada día, se pone a 0), este importe se actualiza cuando el cliente extrae dinero, apellido y nombre del titular, CUIL del titular.
Reglas de Negocio:
El importe a extraer, nunca puede superar al saldo
El importe extraído en el día no puede superar el importe máximo que se puede extraer en un día, que la entidad bancaria ha fijado en $10000 por día.

```
class CajaDeAhorro:
    __numero: int
    __saldo: float
    __cbu: str
    __pesosExtradidos: int
    __apellido: str
    __nombre: str
    __CUIL: str
    __MAXIMOEXTRACCIONDIARIA=10000
    def __init__(self, numero, saldo, cbu, apellido, nombre, cuil, pesosExtraidos=0):
        self.__numero=numero
        self.__saldo=saldo
        self.__cbu=cbu
        self.__apellido=apellido
        self.__nombre=nombre
        self.__CUIL=cuil
        self.__pesosExtradidos=pesosExtraidos
```

```
    def getSaldo(self):
        return self.__saldo
    def extraer(self, importe):
        self.__saldo-=importe
        self.__pesosExtraidos+=importe
    def depositar(self, importe):
        self.__saldo+=importe
    @classmethod
    def getMaximoExtraccionDiara(cls):
        return cls.__MAXIMOEXTRACCIONDIARIA
```

Testing (XIV)

```
from claseCajaDeAhorro import CajaDeAhorro
import unittest
class TestCajaDeAhorro(unittest.TestCase):
    def setUp(self):
        self.__caja= CajaDeAhorro(1001,35000,11131001,'Castro','Luciana', '27-34111222-7')
    def test_depositos(self):
        self.__caja.depositar(5001)
        self.assertEqual(self.__caja.getSaldo(),40001)
    def test_extracciones_OK(self):
        self.__caja.extraer(5001)
        self.assertEqual(self.__caja.getSaldo(),29999)
    def test_extracciones_NOOK(self):
        self.__caja.extraer(5001)
        self.__caja.extraer(31000)
        self.assertEqual(self.__caja.getSaldo(),29999)

if __name__=='__main__':
    unittest.main()
```

Nota: agregar una función test que contemple el testing de la función depositar con un valor negativo.

```
.F.
======================================================================
FAIL: test_extracciones_NOOK (__main__.TestCajaDeAhorro)
----------------------------------------------------------------------
Traceback (most recent call last):
  File «../Scripts/claseTestCajaDeAhorro.py", line 15, in test_extracciones_NOOK
    self.assertEqual(self.__caja.getSaldo(),29999)
AssertionError: -1001 != 29999

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)

Process finished with exit code 1
```

Testing Ejemplo completo (I)

Dadas las clases Nodo y Lista vistas anteriormente (diapositiva 56), escriba una clase derivada de TestCase, con funciones test que permitan testear los métodos de agregar elementos a la lista, supresión de elementos de la lista, chequear que la lista no está vacía y vaciar lista y chequear que la lista está vacía.

Testing Ejemplo completo (II)

- Diseñar el método setUp que crea la lista y agrega elementos
- Diseñar los test para:
  - Verificar que agregó elementos a la lista.
  - Verificar que la lista no está vacía
  - Eliminar un elemento intermedio de la lista, verificar que lo eliminó.
  - Eliminar el primer elemento, verificar que lo eliminó.
  - Eliminar último elemento de la lista, verificar que lo eliminó.
  - Vaciar la lista, verificar que está vacía.

Clase Base Abstracta (I)

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.

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.

En la diapositiva 34 se dejó constancia de que la clase Cuerpo tenía un método sin cuerpo, el método superficieBase(), ya que la clase no posee los suficientes atributos para poder llevar a cabo un cálculo de la superficie de la base, por lo que se proponía como método abstracto, transformado así a la clase en clase abstracta.

Dusty Phillips – Chapter 3 – Páginas 78 a 81
Steven F. Lott – Chapter 4 – Páginas 131 a 133

Clase Base Abstracta (II)

```
import abc
from abc import ABC
import numpy as np
import math
class Cuerpo(ABC):
    __altura: int
    def __init__(self, altura):
        self.__altura=altura
    @abc.abstractmethod
    def superficieBase():
        pass
    def volumen(self):
        return self.superficieBase()*self.__altura
    def getAltura(self):
        return self.__altura
```

A partir de esta definición de la clase Cuerpo, las clases ParalelepipedoRectangulo y Cilindro, se ven obligadas a reescribir el método superficieBase(), de lo contrario, lo heredan como método abstracto, y automáticamente se convierten en clases abstractas.

Si se intenta instanciar a la clase Cuerpo, se obtiene el siguiente mensaje de error:

```
c = Cuerpo(3)
```

TypeError: Can't instantiate abstract class Cuerpo with abstract methods superficieBase

Actividad Conjunta (I)

Contexto: La empresa de transportes Trenes del Oeste necesita el desarrollo de un sistema informático para administrar las formaciones de vagones.
Una formación tiene una máquina y vagones. Los vagones pueden ser para transporte de líquidos (cilíndricos) o para transporte de personas o transporte de cargas (éstos dos últimos tienen forma de paralelepípedo rectángulo). Las formaciones pueden ser mixtas o solo de carga. Las formaciones mixtas pueden llevar cinco de transporte de líquido, diez vagones de carga y veinte vagones de transporte de pasajeros. Las formaciones de carga pueden transportar hasta cien vagones de carga, siempre y cuando la carga máxima no supere los 6000tn que la máquina puede tirar.
Una formación tiene una fecha y hora de salida, una lugar de salida y lugar de destino.
Una máquina posee número de serie, marca y tipo de combustible.
Una máquina es manejada por un maquinista, del que el sistema debe registrar: DNI, nombre y apellido, y fecha de nacimiento.
Para el cálculo del volumen se necesita:

- Para los vagones de forma cúbica: alto, ancho, profundidad
- Para los vagones cilíndricos: alto y radio.

Todos los vagones tienen un número de serie que los identifica.
Los vagones que transportan líquidos pueden llevar solamente combustibles (Nafta Infinia, Nafta Súper o Gasoil)
Los vagones de carga tienen la carga transportada (en toneladas).
Los tipos de vagones de pasajeros son: Primera Clase, Clase Turista. Los vagones de pasajero tienen la cantidad de asientos, y los asientos efectivamente ocupados.
La empresa solicita que el sistema permita:
Crear una formación
Calcular el volumen de carga de una formación
Verificar que el volumen de carga no supere las 6000tn de carga total
Mostrar todos los datos de una formación.
Para cada vagón de pasajeros, determinar el número de asientos no utilizados
Para cada vagón de carga, determinar la capacidad ociosa (volumen del vagón – carga transportada)

Actividad Conjunta (II)

Diseño propuesto

Codificar la formación con sus vagones.

Interfaces (I)

- 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 **interface tendrá acceso sólo a 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 implementar los métodos abstractos (subclase abstracta), mientras que las **clases que implementen una interfaz deben proveer implementación de todos los métodos de la interfaz**.

Interfaces (II)

- 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.

- Python no permite definir interfaces, es necesario incorporar una librería externa para emular el comportamiento de las interfaces que proveen otros lenguajes como Java y C#.

- La librería que se usará para mostrar el uso de las interfaces como forma de restricción de métodos, es la librería Zope.

https://pypi.org/project/zope.interface/

https://zopeinterface.readthedocs.io/en/latest/README.html

¿Cuando definir interfaces?

- Las interfaces no implican una sobrecarga en el procesamiento.
- 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**.

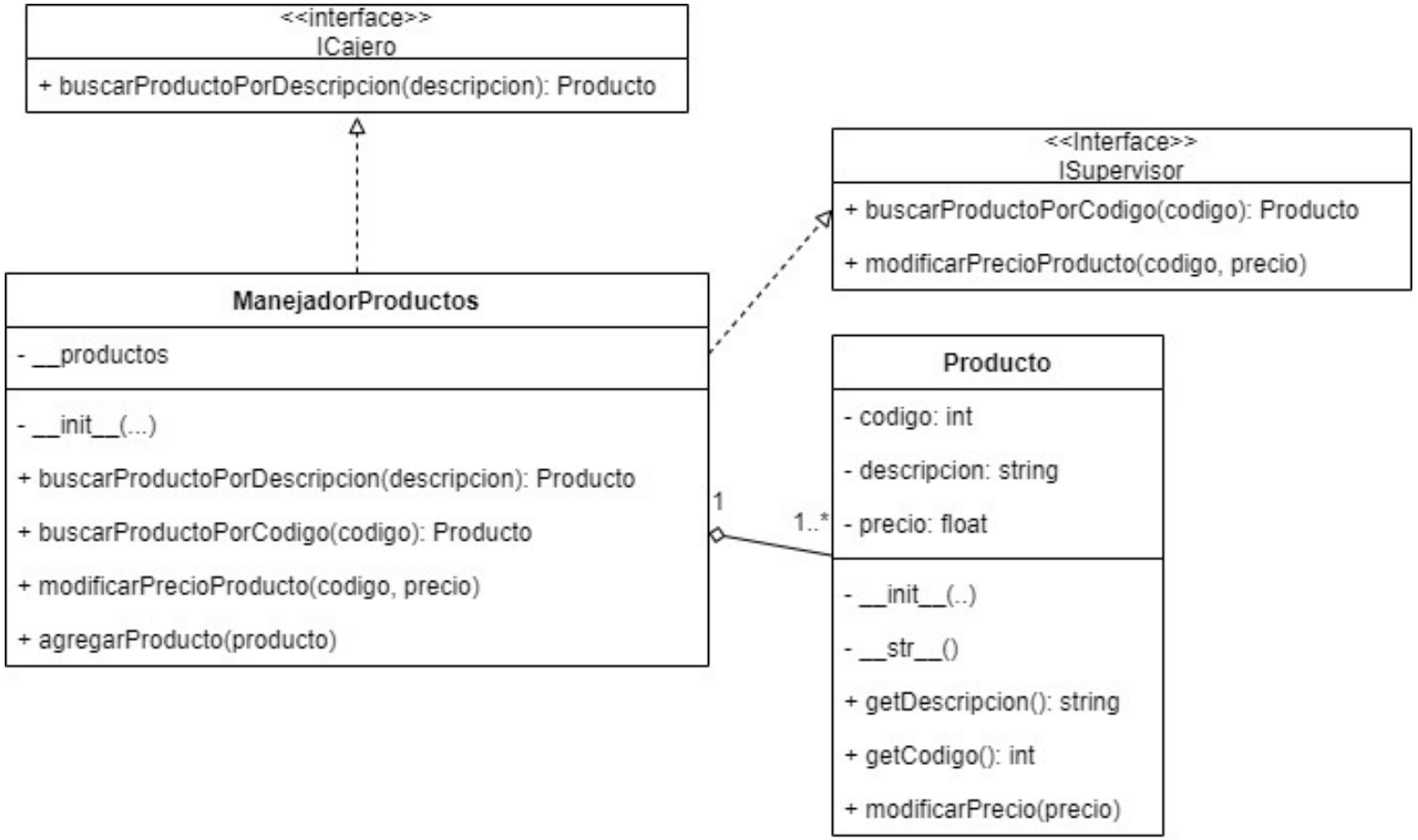
Interfaces para restringir acceso a funciones miembro (I)

El uso de interfaces, permite restringir el acceso a funciones miembro de las clases que las implementen.

Ejemplo: El supermercado «El Changuito lleno», necesita administrar el conjunto de productos que vende. Los productos poseen un código, descripción y precio. Existen dos tipos de usuarios, el usuario Cajero, quien sólo tiene acceso a buscar los productos por la descripción del mismo, y el usuario Supervisor que tiene acceso a buscar los productos por código de producto y además puede modificar el precio de los mismos.

Interfaces para restringir acceso a funciones miembro (II)

Diagrama UML de la situación problemática planteada



Interfaces para restringir acceso a funciones miembro (III)

Librería zope, y declaración de interfaces

```
from zope.interface import Interface
from zope.interface import implementer
'''Declaración de interface ICajero
   El Cajero solo puede buscar productos por descripción
   el método declarado es
   buscarProductoPorDescripcion(descripcion)
   '''
class ICajero(Interface):
    def buscarProductoPorDescripcion(descripcion):
        pass
'''Declaración de interface ISupervisor
   El Supervisor modificar el precio de un producto, que busca por código
   Los métodos que declara la intereface es
   buscarProductoPorCodigo(codigo)
   modificarPrecioProducto(codigo, precio)
   '''
class ISupervisor(Interface):
    def buscarProductoPorCodigo(codigo):
        pass
    def modificarPrecioProducto(codigo, precio):
        pass
```

Interfaces para restringir acceso a funciones miembro (IV)

Clase Producto

```
class Producto(object):
    __codigo: int
    __descripcion: str
    __precio: float
    def __init__(self, codigo, descripcion, precio):
        self.__codigo=codigo
        self.__descripcion=descripcion
        self.__precio=precio
    def __str__(self):
        cadena = 'Codigo    Descripcion      Precio    \n'
        cadena += '{0:6d}    {1:15s} {2:6.2f}'.format(self.__codigo, self.__descripcion,self.__precio)
        return cadena
    def getDescripcion(self):
        return self.__descripcion
    def getCodigo(self):
        return self.__codigo
    def modificarPrecio(self, precio):
        self.__precio=precio
```

Interfaces para restringir acceso a funciones miembro (V)

```
@implementer(ICajero)
@implementer(ISupervisor)
class ManejadorProductos:
    __productos: list
    def __init__(self):
        self.__productos=[]
    def agregarProducto(self, producto):
        self.__productos.append(producto)
    '''Método de la interface ICajero'''
    def buscarProductoPorDescripcion(self, descripcion):
        i=0
        bandera=False
        retorno=None
        while not bandera and i<len(self.__productos):
            unProducto=self.__productos[i]
            if unProducto.getDescripcion()==descripcion:
                bandera=True
            else:
                i+=1
        if bandera:
            retorno = self.__productos[i]
        return retorno
    '''Métodos de la interface ISupervisor'''
    def buscarProductoPorCodigo(self, codigo):
        i=0
        bandera=False
        retorno=None
        while not bandera and i<len(self.__productos):
            unProducto=self.__productos[i]
            if unProducto.getCodigo()==codigo:
                bandera=True
            else:
                i+=1
        if bandera:
            retorno = self.__productos[i]
        return retorno
    def modificarPrecio(self, codigo, precio):
        producto = self.buscarProductoPorCodigo(codigo)
        if producto == None:
            print('Producto código {}, no encontrado'.format(codigo))
        else:
            producto.modificarPrecio(precio)
```

Declara que la clase que viene a continuación implementa las interfaces ICajero e ISupervisor, lo hace a través del decorador:

@implementer(Interface)

Interfaces para restringir acceso a funciones miembro (VI)

```
def cajero(manejarVendedor: ICajero):
    descripcion=input('Descripcion de producto a buscar: ')
    producto = manejarVendedor.buscarProductoPorDescripcion(descripcion)
    if producto == None:
        print('Producto {}, no encontrado'.format(descripcion))
    else:
        print(producto)
def supervisor(manejarSupervisor: ISupervisor):
    codigo=int(input('Código del producto a cambiar precio: '))
    producto = manejarSupervisor.buscarProductoPorCodigo(codigo)
    if producto == None:
        print('No hay un Producto con código {}'.format(codigo))
    else:
        print(producto)
        precio=float(input('Nuevo precio: '))
        manejarSupervisor.modificarPrecio(codigo, precio)
        print(producto)
def testInterfaces():
    manejadorProductos = ManejadorProductos()
    unProducto=Producto(1,'Arroz 1kg',52)
    manejadorProductos.agregarProducto(unProducto)
    unProducto=Producto(2,'Yerba 1/2kg',120)
    manejadorProductos.agregarProducto(unProducto)
    usuario=input('Usuario (Admin/Cajero): ')
    clave=input('Clave:')
    if usuario.lower() == 'Admin'.lower() and clave =='a54321':
        '''testeando supervisor '''
        supervisor(ISupervisor(manejadorProductos))
    else:
        if usuario.lower() == 'Cajero'.lower() and clave == 'c12345':
            '''testeado cajero'''
            cajero(ICajero(manejadorProductos))
if __name__=='__main__':
    testInterfaces()
```

Restricción a métodos exclusivos de ISupervisor

Restricción a métodos exclusivos de ICajero

Diccionarios en Python (I)

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 de las 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 y ente 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})
```

El acceso a las componentes es a través de las claves, al estilo de un arreglo o una lista Python:

```
diccionario=dict({'nombre':'Arnold', 'edad':55,'genero':'masculino','hijos':'Si',
            'casado':'No', 'cantidad de hijos':4, 'sueldo':33876
            })
print('Diccionario: ',diccionario)
print('Nombre: ', diccionario['nombre'])
print('Edad: ',diccionario['edad'])
print('Cantidad de hijos: ',diccionario['cantidad de hijos'])
```

Consola Python

```
Diccionario:  {'nombre': 'Arnold', 'edad': 55,
'genero': 'masculino', 'hijos': 'Si', 'casado':
'No', 'cantidad de hijos': 4, 'sueldo': 33876}
Nombre:  Arnold
Edad:  55
Cantidad de hijos:  4
```

Mark Lutz - Chapter 8 – Páginas 252 a 271

Diccionarios en Python (II)

Desde un diccionario se pueden inicializar los atributos de una clase:

```
class Alumno(object):
    __registro: int
    __nombre: str
    __apellido: str
    __carrera: str
    def __init__(self, registro, nombre, apellido, carrera):
        self.__registro=registro
        self.__nombre=nombre
        self.__apellido=apellido
        self.__carrera=carrera
    def __str__(self):
        cadena='Registro: {}, Carrera: {}\n'.format(self.__registro, self.__carrera)
        cadena+='Apellido: {}, Nombre: {}\n'.format(self.__apellido, self.__nombre)
        return cadena
if __name__=='__main__':
    diccionario = dict({'registro':12345, 'nombre':'Carla',
                        'apellido':'Ibaceta','carrera':'LCC'
                        })
    unAlumno= Alumno(**diccionario)
    print(unAlumno)
```

Consola Python

```
Registro: 12345, Carrera: LCC
Apellido: Ibaceta, Nombre: Carla
```

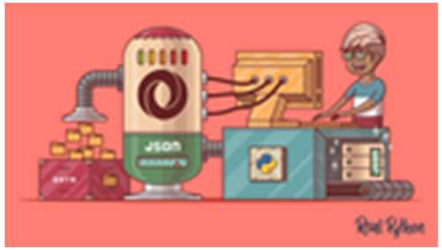
Para que funcione las claves del diccionario deben coincidir en cantidad y nombre con los parámetros formales

Alumno(diccionario): indica cantidad arbitraria de argumentos**

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, otro formato de intercambio de datos que requiere más metainformación y, por lo tanto, consume más ancho de banda y recursos.
JSON puede ser codificado y decodificado con Python, se utiliza la librería JSON de Python.
Los objetos de Python y los definidos por el programador pueden codificarse y decodificarse a través del lenguaje.
El flujo de trabajo será el siguiente:

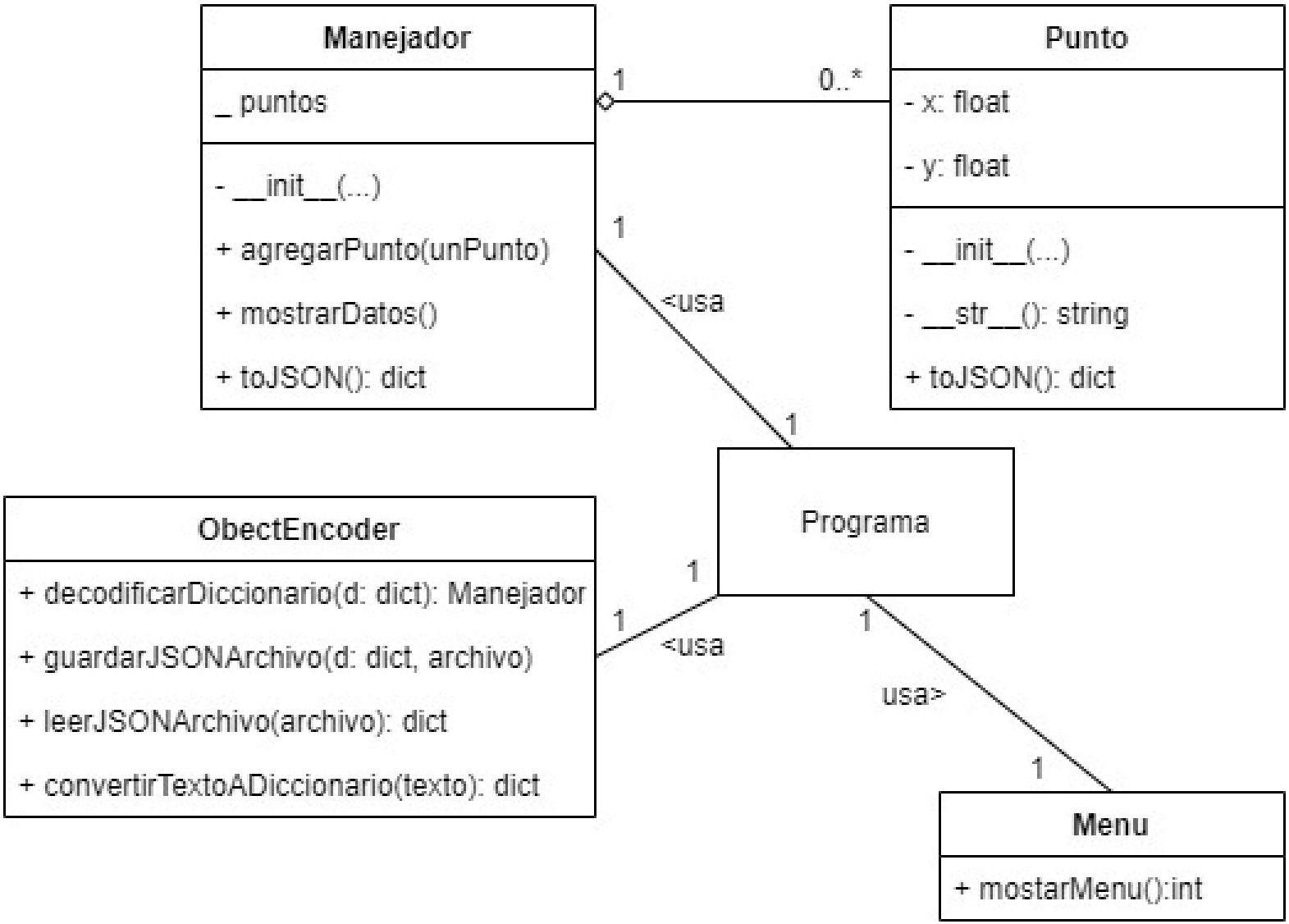
1. Importar la librería json
2. Proveer a las clases un método para codificar la representación tipo diccionario necesaria para JSON
3. Leer los datos usando las funciones load() o loads()
4. Procesar los datos
5. Escribir los datos usando las funciones dump() o dumps()



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().
Se deberá proveer una función para decodificar el texto generado al codificar la clase.
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.**

Steven F.Lott – Chapter 10 – Páginas 414 a 444

Ejemplo – JSON (I)



Ejemplo – JSON (II)

Clase Punto

```
import json
from pathlib import Path
class Punto(object):
    __x: int
    __y: int
    def __init__(self,x , y):
        self.__x=x
        self.__y=y
    def __str__(self):
        cadena='(x,y)=({},{})'.format(self.__x, self.__y)
        return cadena
    def toJSON(self):
        d = dict(
                __class__=self.__class__.__name__,
                __atributos__=dict(
                                x=self.__x,
                                y=self.__y
                            )
                )
        return d
```

Ejemplo – JSON (III)

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.

Ejemplo – JSON (IV)

Clase ObjectEncoder

```
class ObjectEncoder(object):
    def decodificarDiccionario(self, d):
        if '__class__' not in d:
            return d
        else:
            class_name=d['__class__']
            class_=eval(class_name)
            if class_name=='Manejador':
                puntos=d['puntos']
                dPunto = puntos[0]
                manejador=class_()
                for i in range(len(puntos)):
                    dPunto=puntos[i]
                    class_name=dPunto.pop('__class__')
                    class_=eval(class_name)
                    atributos=dPunto['__atributos__']
                    unPunto=class_(**atributos)
                    manejador.agregarPunto(unPunto)
            return manejador
```

```
def guardarJSONArchivo(self, diccionario, archivo):
        with Path(archivo).open("w", encoding="UTF-8") as destino:
            json.dump(diccionario, destino, indent=4)
            destino.close()
    def leerJSONArchivo(self,archivo):
        with Path(archivo).open(encoding="UTF-8") as fuente:
            diccionario=json.load(fuente)
            fuente.close()
            return diccionario
    def convertirTextoADiccionario(self, texto):
        return json.loads(texto)
```

Ejemplo – JSON (V)

Clase Menu

```
class Menu(object):
    def mostrarMenu(self):
        print('Menú de Opciones: ')
        print('-----------------')
        print('1 - Crear un Punto')
        print('2 - Guardar Puntos en Archivo')
        print('3 - Leer datos de Puntos')
        print('4 - Mostrar datos Puntos')
        print('5 - Salir')
        opcionCorrecta = False
        while not opcionCorrecta:
            opcion=int(input('Seleccione un número del 1 al 5: '))
            if opcion in [1,2,3,4,5]:
                opcionCorrecta=True
        return opcion
```

Ejemplo – JSON (VI)

Programa Principal

```
if __name__=='__main__':
    jsonF=ObjectEncoder()
    puntos = Manejador()
    bandera=True
    while bandera:
        menu=Menu()
        opcion=menu.mostrarMenu()
        if opcion==1:
            print('Creando un nuevo Punto')
            x=int(input('Coordenada x: '))
            y=int(input('Coordenada y: '))
            punto=Punto(x,y)
            puntos.agregarPunto(punto)
        else:
            if opcion==2:
                d=puntos.toJSON()
                jsonF.guardarJSONArchivo(d,'datosPuntos.json')
            else:
                if opcion==3:
                    diccionario=jsonF.leerJSONArchivo('datosPuntos.json')
                    puntos=jsonF.decodificarDiccionario(diccionario)
                else:
                    if opcion==4:
                        puntos.mostrarDatos()
                    else:
                        bandera=False
                        print('Ha seleccionado salir, hasta la vuelta')
```

Ejemplo – JSON (VI)

Entrada:
(x,y)=(2,6)
(x,y)=(3,-6)
(x,y)=(6,-1)
(x,y)=(0,0)

Archivo 'datosPuntos.json'

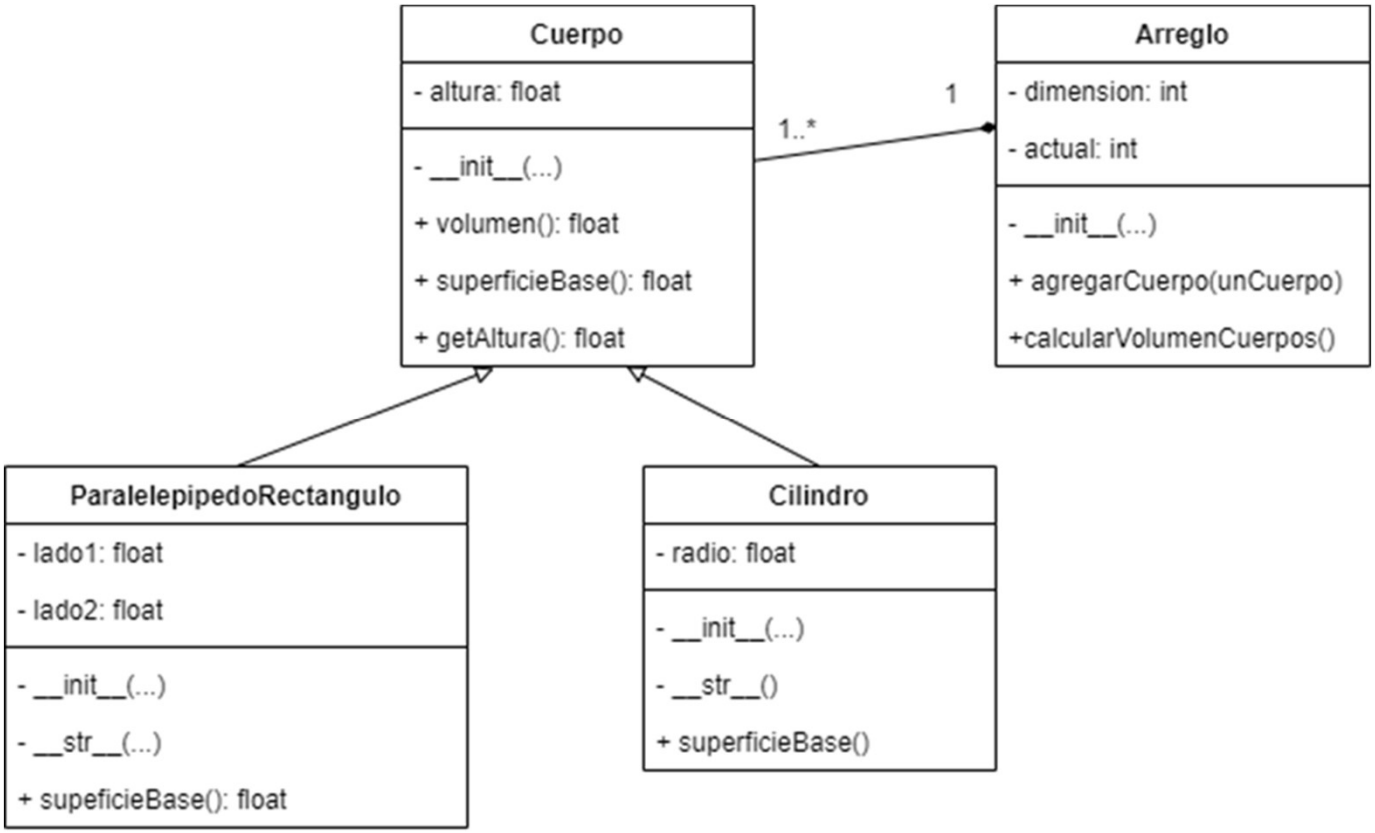

```
{
   "__class__": "Manejador",
   "puntos": [
      {
         "__class__": "Punto",
         "__atributos__": {
            "x": 2,
            "y": 6
         }
      },
      {
         "__class__": "Punto",
         "__atributos__": {
            "x": 3,
            "y": -6
         }
      },
      {
         "__class__": "Punto",
         "__atributos__": {
            "x": 6,
            "y": -1
         }
      },
      {
         "__class__": "Punto",
         "__atributos__": {
            "x": 0,
            "y": 0
         }
      }
   ]
}
```

Actividad Conjunta

Dada la jerarquía de clases y la colección arreglo, ya analizadas y codificadas (diapositivas 34 a 37), cuyo diagrama UML se muestra a continuación, se pide proveer los mecanismos de persistencia utilizando archivos con formato JSON, para la colección de objetos.

Además, se deberá crear un menú de opciones para llevar a cabo las siguientes acciones:

- ✓ Crear y almacenar en la colección un Paralelepípedo Rectángulo (datos leídos desde teclado)
- ✓ Crear y almacenar en la colección un Cilindro (datos leídos desde teclado).
- ✓ Guardar la colección en un archivo JSON.
- ✓ Recuperar la colección desde un archivo JSON.
- ✓ Mostrar los datos de los objetos de la colección.
- ✓ Dado el volumen de producción de jugo de uva (dato leído desde teclado) de la fábrica «Jugos del Oeste», calcular si alcanzan o no los objetos almacenados en la colección, si faltaran informar el volumen que quedará sin envasar.



FIN UNIDAD 3