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 2023

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.

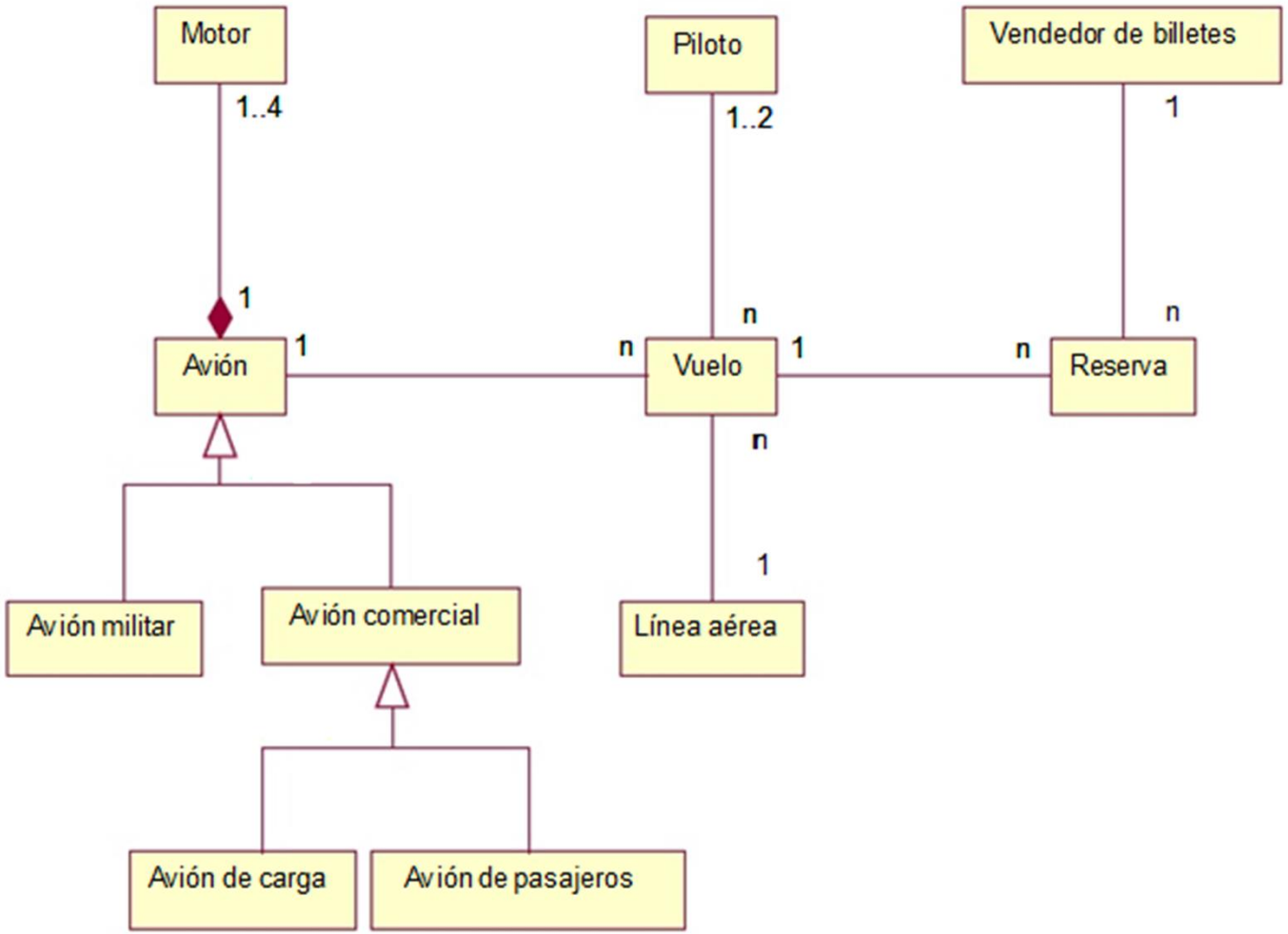
Aunque algunas clases pueden existir aisladas, la mayoría no puede, y deben cooperar unas con otras.

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.

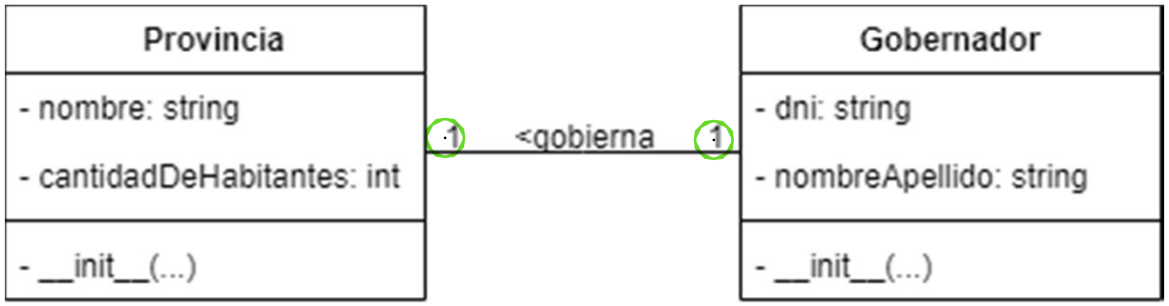
Asociación - Agregación – Composición - Herencia

Relaciones entre Clases en Python (II)



Identificar y nombrar las distintas relaciones

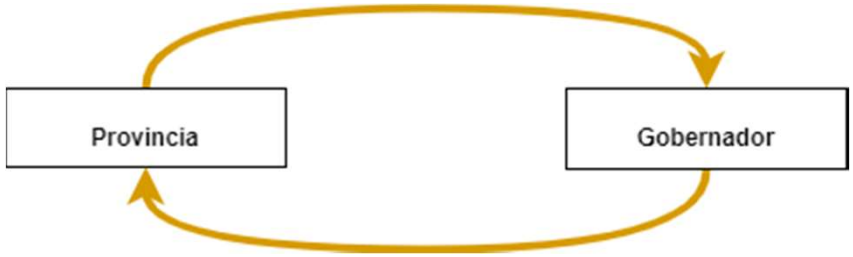
Asociación (I)




```
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

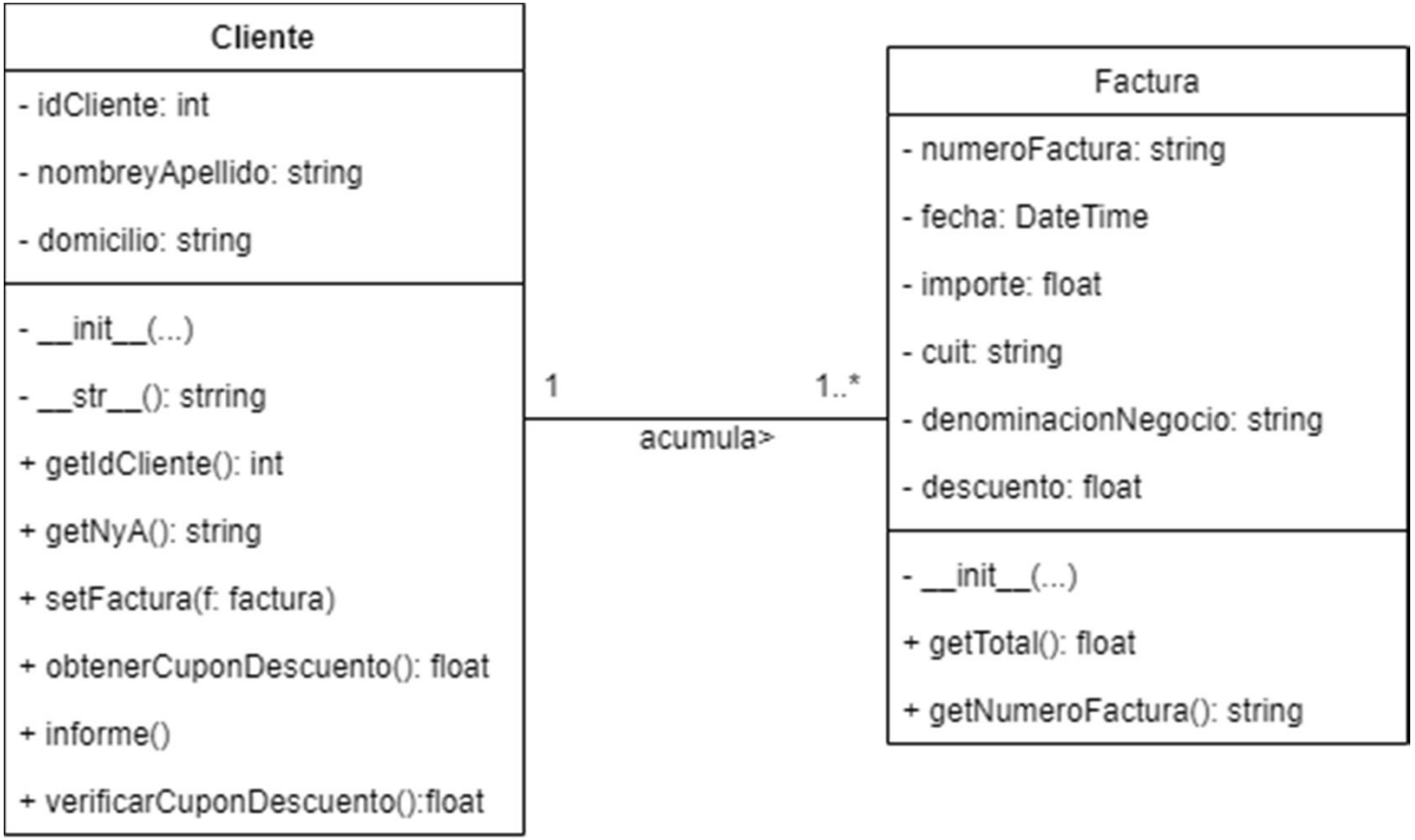
ImportError: cannot import name 'Gobernador' from partially initialized module 'claseGobernador' (most likely due to a circular import)

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
```

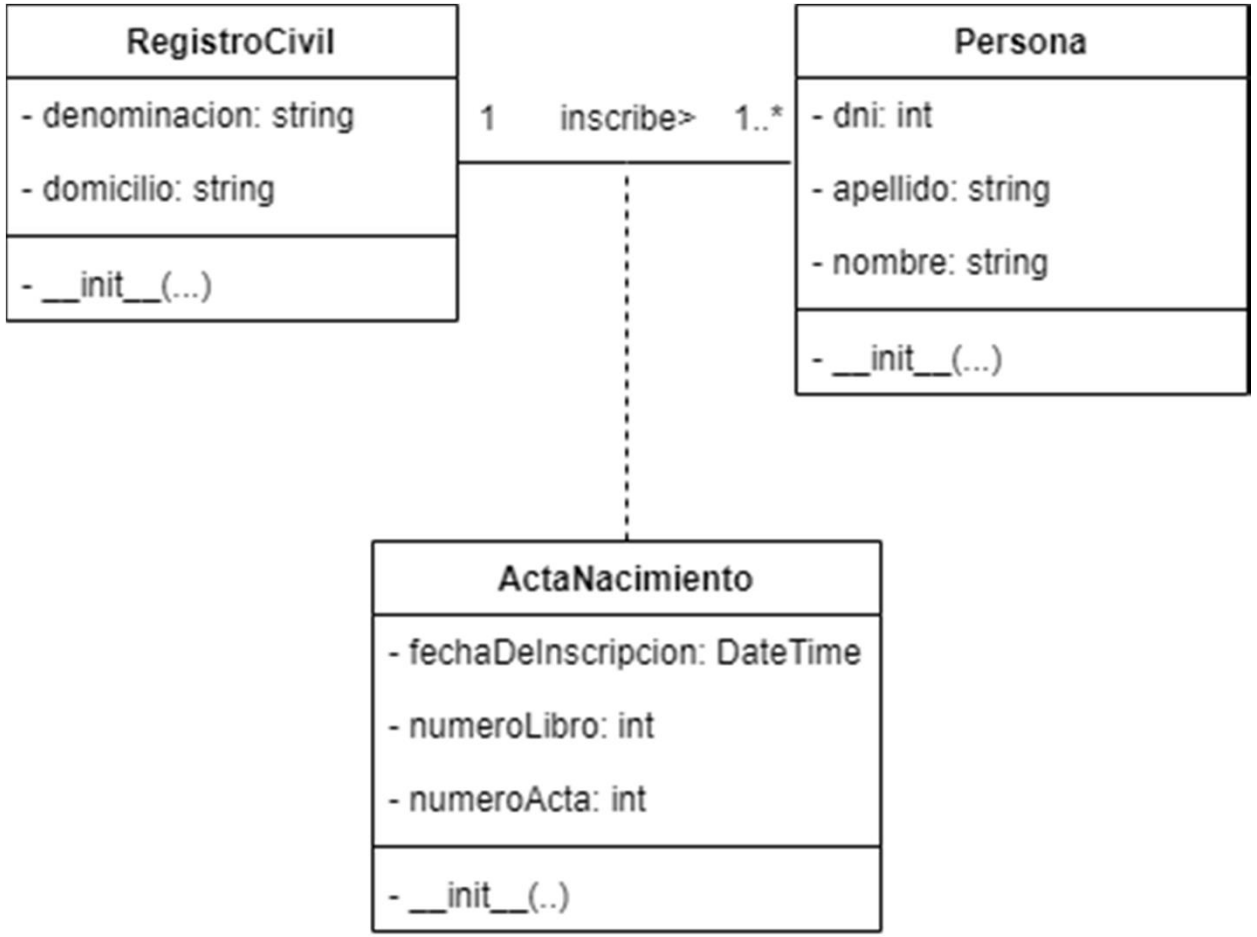
Asociación (V)

```
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=0
        __apellido=''
        __nombre=''
        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
```

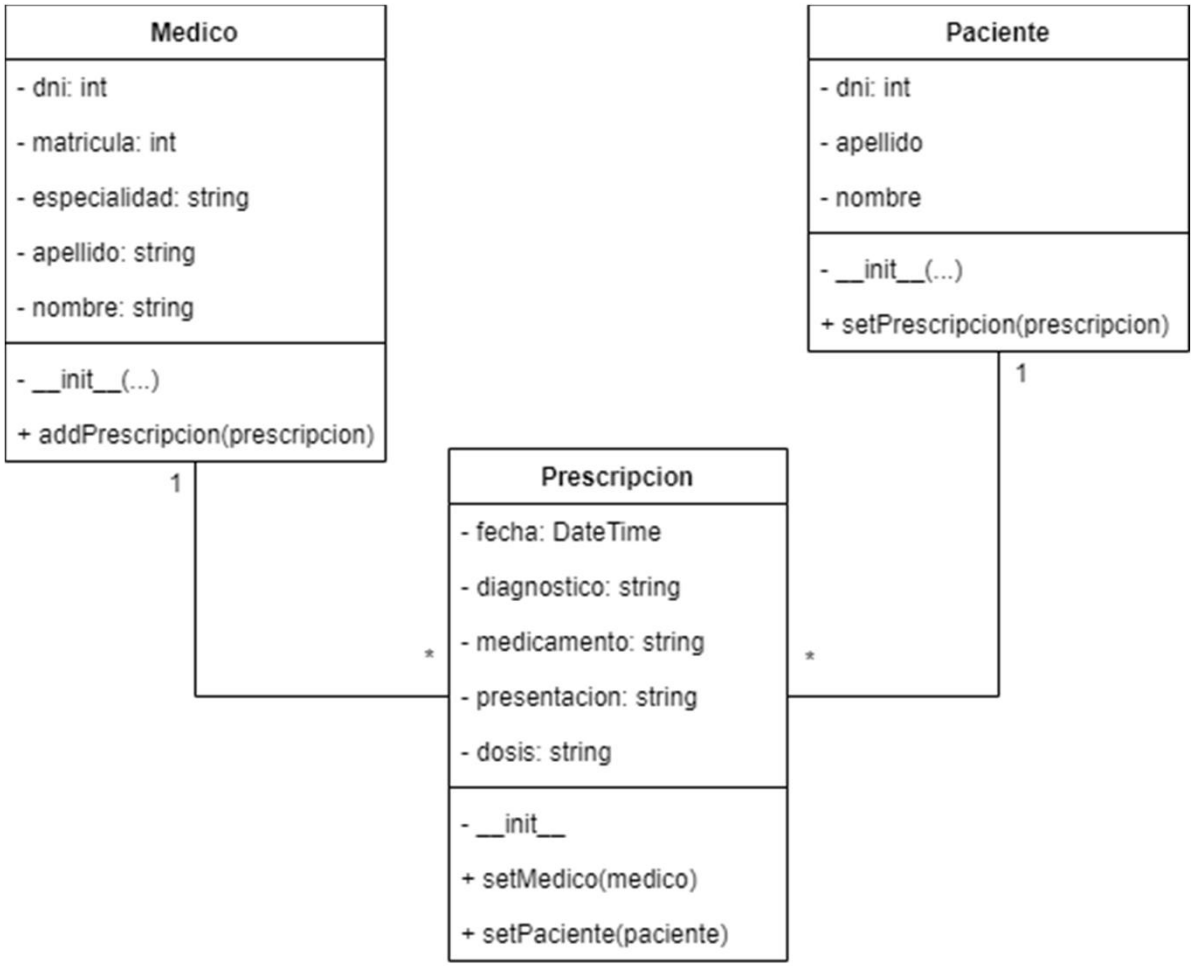
```
def testClaseAsociacion():
    registro = RegistroCivil('Registro Cuarta Zona', 'Av. Córdoba y
Urquiza')
    persona = Persona(20112113, 'Carlos', 'Vargas')
    persona1 = Persona(3444222, 'Anastacia', 'Arboleda')
    registro.inscribirPersona(persona, '28/01/2019')
    registro.inscribirPersona(persona1,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

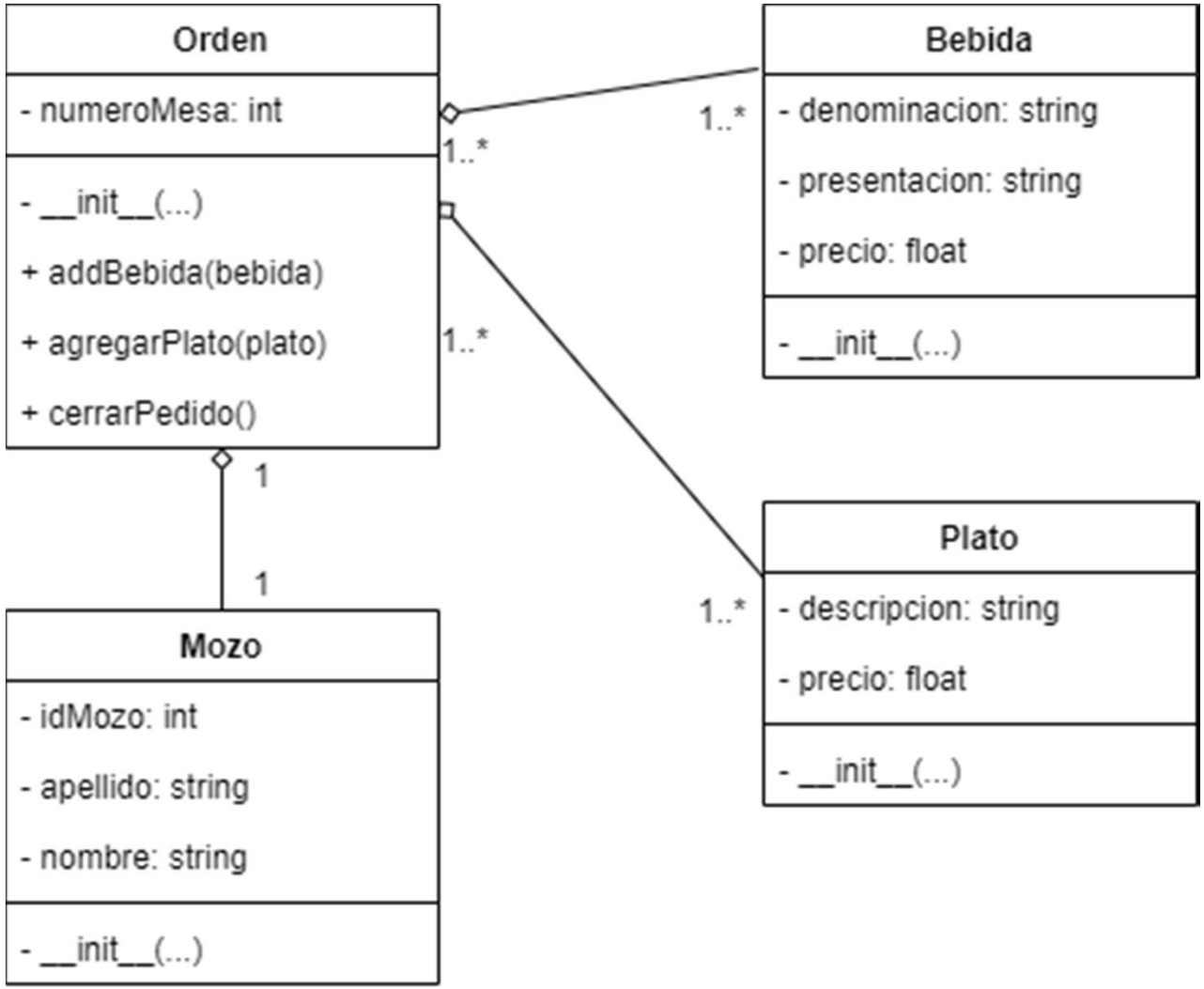
```
def testClaseModelaAsociacion():
    paciente = Paciente(14555699, 'Vergara', 'Andrea')
    medico = Medico(19327881, 1125, 'Clínica Médica',
'González', 'Jorge')
    prescripcion = Prescripcion('11/01/2020', 'Rinitis', 'Hexaler',
'10 comprimidos', '1 por día',medico, paciente)
    prescripcion2 = 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

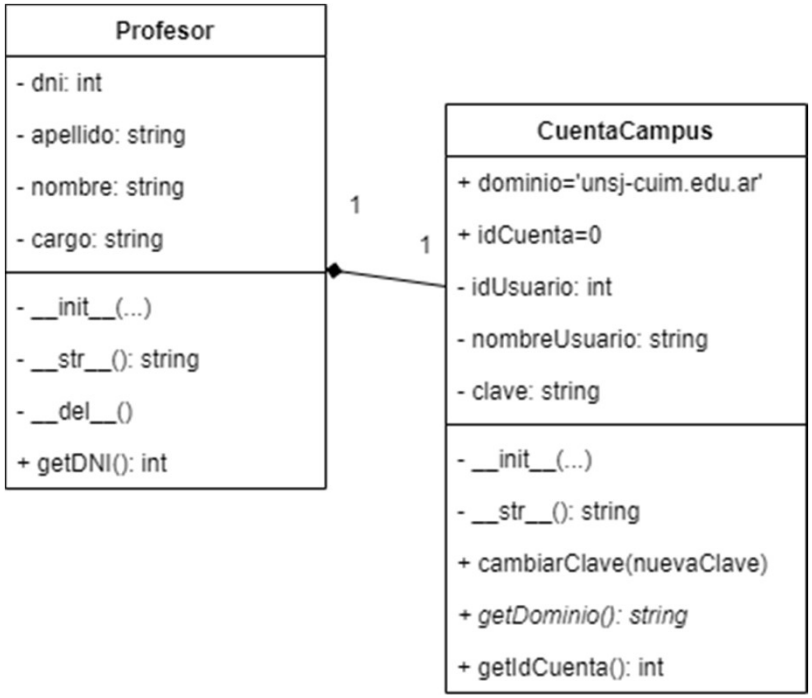
```
Pedido número:  1
Bebidas
Coca cola          150.00
Aquarius           100.00
Aquarius           100.00
Platos
Lomo especial       325.00
Papa frita chica    125.00
Lomo especial       325.00
Lomo especial       325.00
Lomo especial       325.00
Total a pagar:     1775.00
Pedido número:  2
Bebidas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

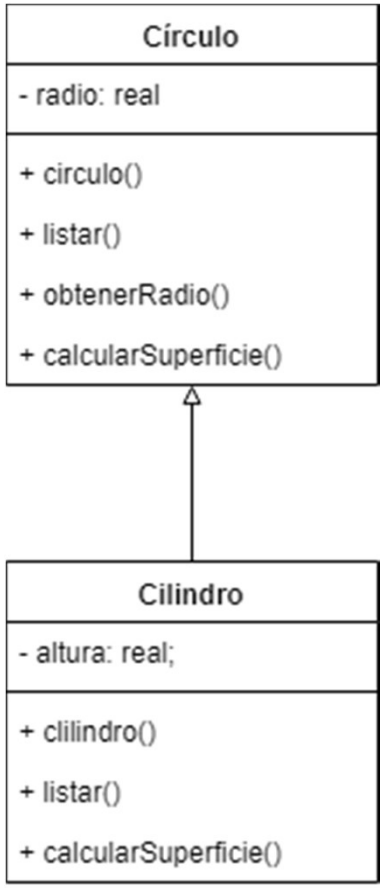
```
Profesor:
Apellido y nombre: Rodríguez, Myriam
Usuario: myriamrodríguez@unsj-cuim.edu.ar
Clave 11334441
Borrando cuenta de usuario....
```

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

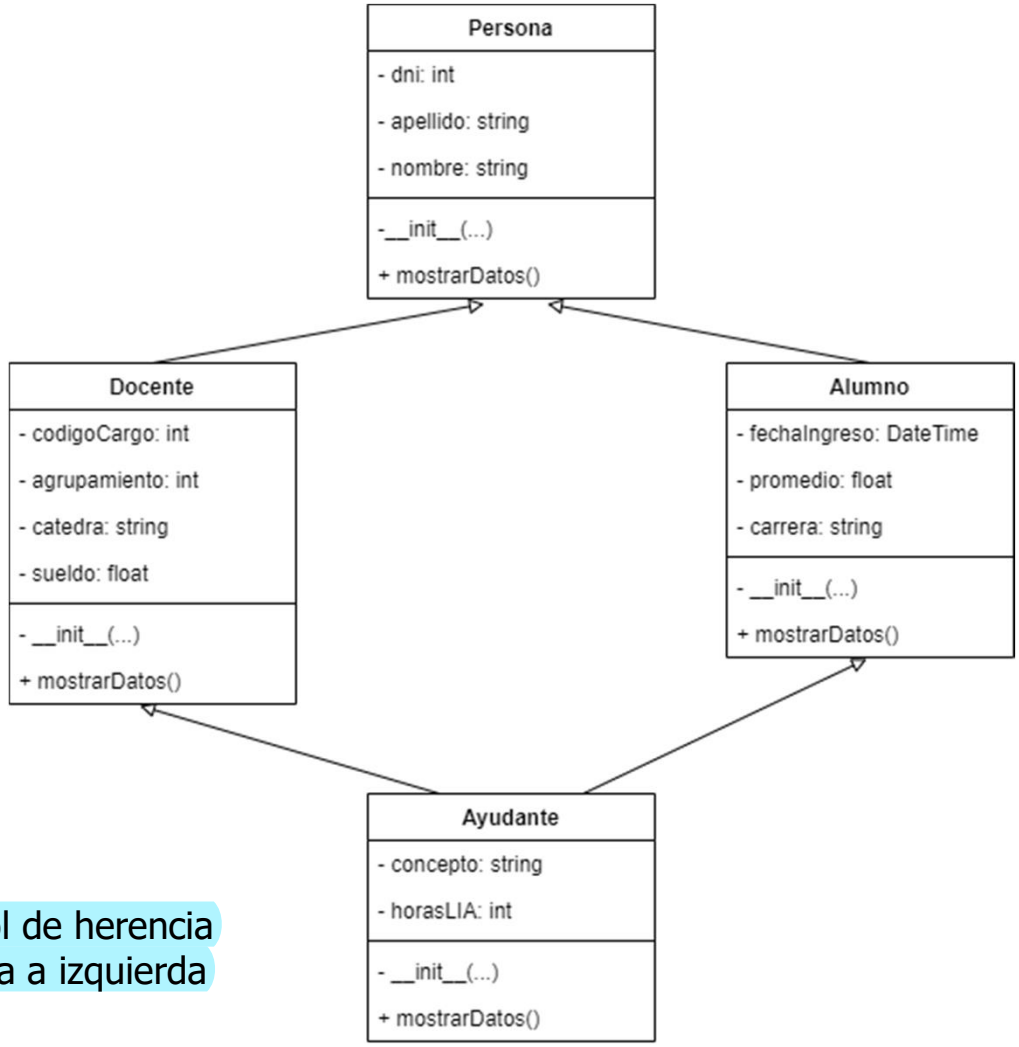
```
['_Punto__x', '_Punto__y', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',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

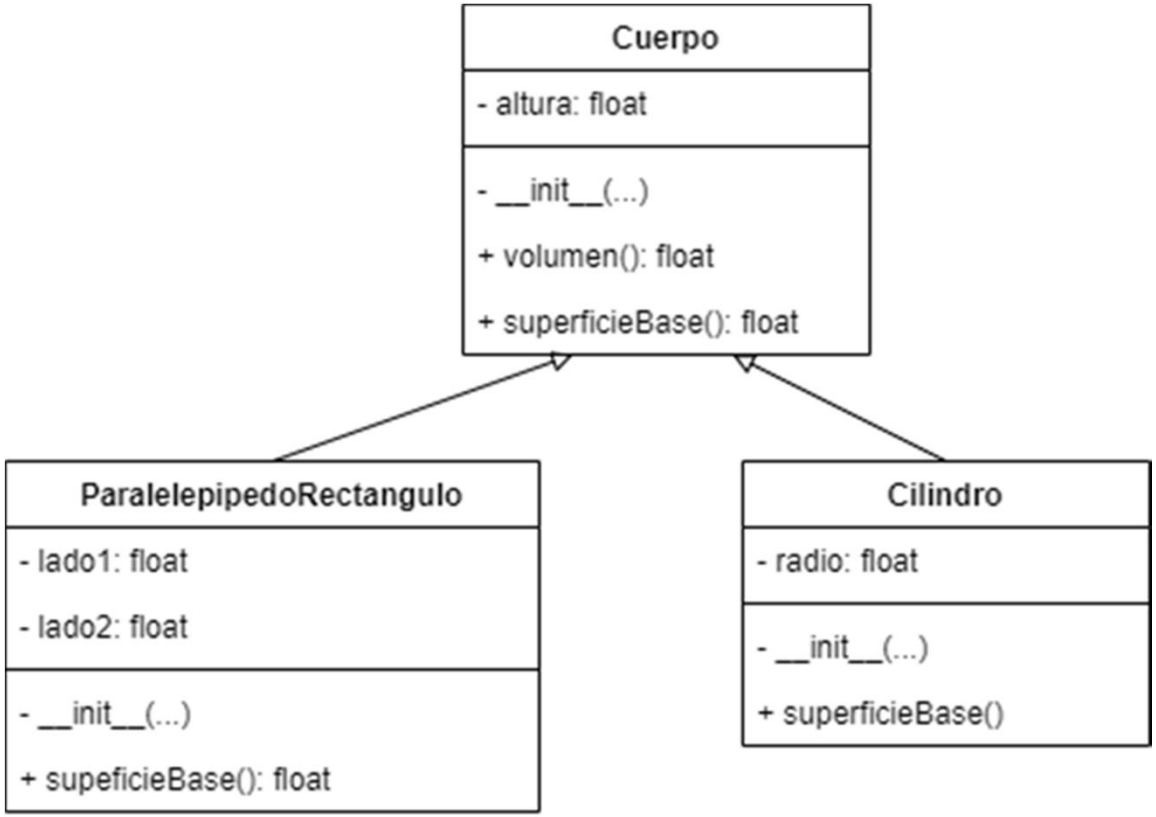
```
class Cilindro(Circulo):
    __altura: float
    def __init__(self, radio, **kwargs):
        super().__init__(radio)
        self.__altura=kwargs['altura']
    def __str__(self):
        return f'Radio: {self.getRadio()}, Altura: {self.__altura}'
```

```
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:

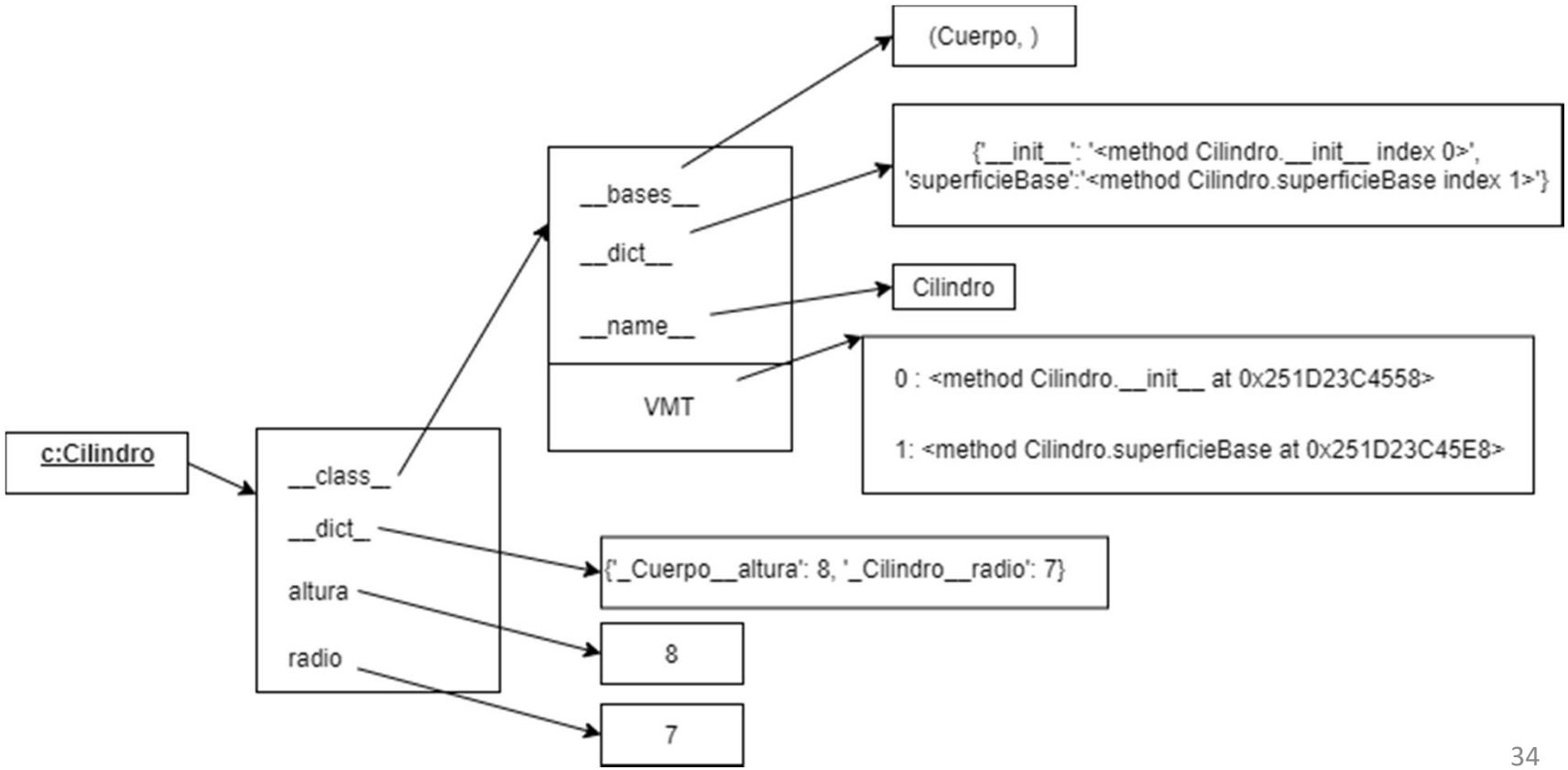


Polimorfismo: Vinculación Dinámica (II)

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

c = Cilindro(7,8)

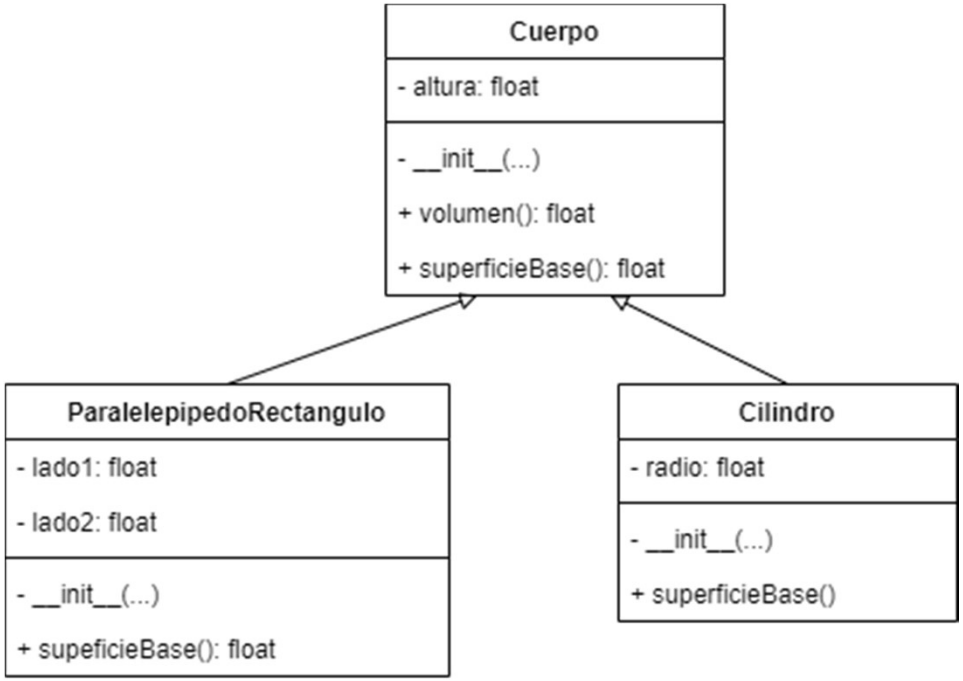
El diagrama de memoria que incluye la tabla de Métodos Virtuales, puede observarse en la figura



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
```

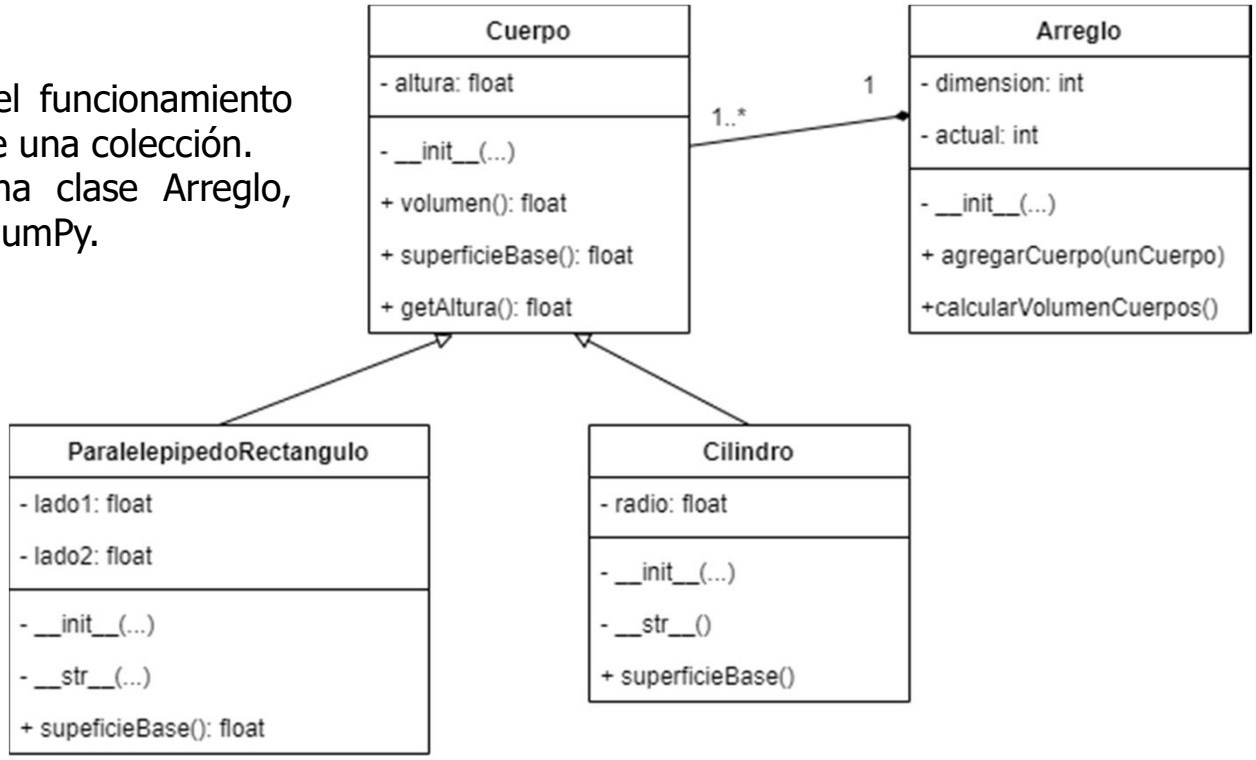
```
class Cilindro(Cuerpo):
    __radio: float
    def __init__(self, altura, radio):
        Cuerpo.__init__(self,altura)
        self.__radio=radio
    def __str__(self):
        cadena ='Cilindro, altura ={}, radio = {}'.format(self.getAltura(),
self.__radio)
        return cadena
    def superficieBase(self):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

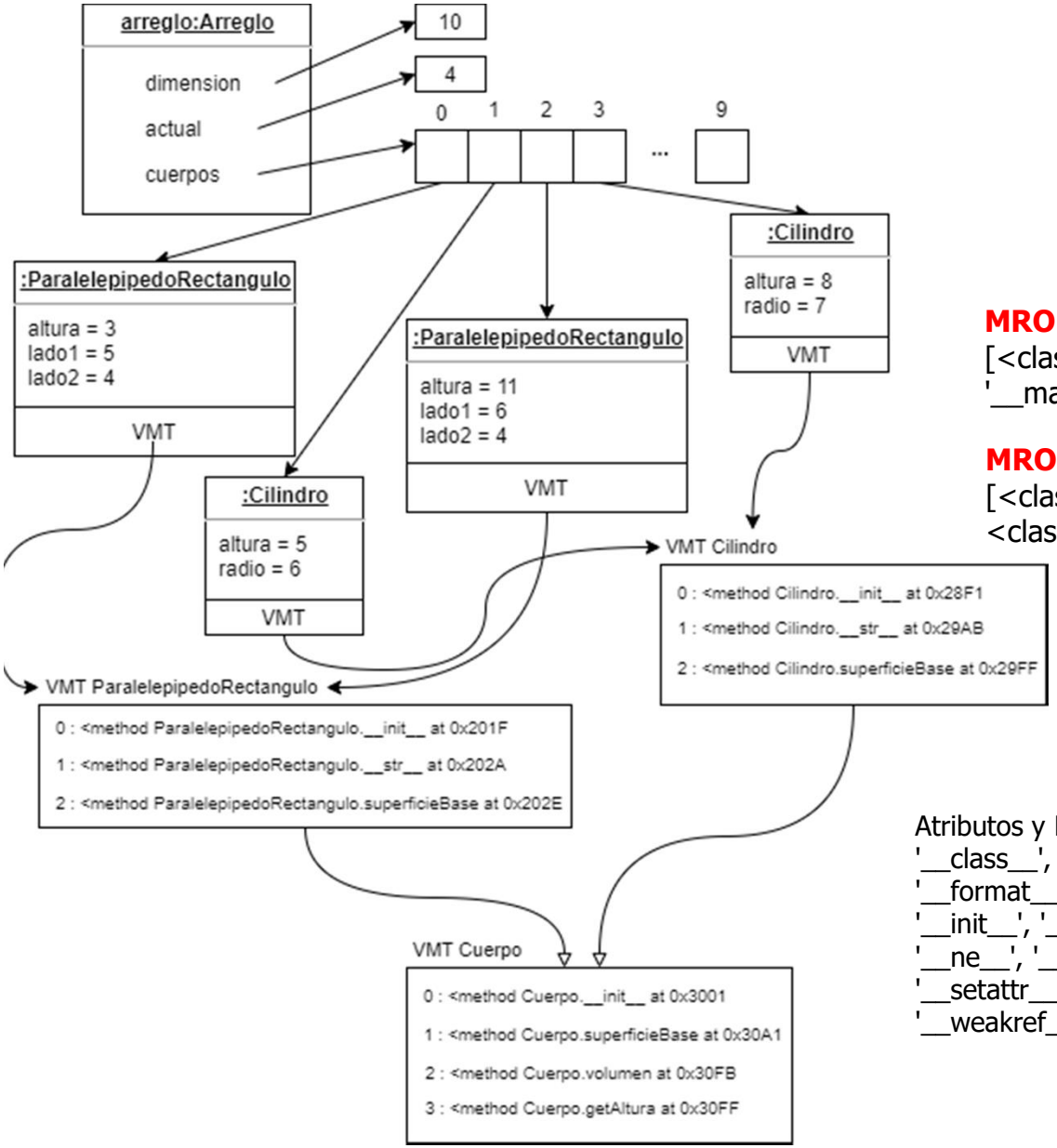
```
def testPolimorfismo():
    arreglo = Arreglo()
    p = ParalelepipedoRectangulo(3,5,4)
    arreglo.agregarCuerpo(p)
    c = Cilindro(5,6)
    arreglo.agregarCuerpo(c)
    p = ParalelepipedoRectangulo(11,6,4)
    arreglo.agregarCuerpo(p)
    c = Cilindro(8,7)
    arreglo.agregarCuerpo(c)
    arreglo.calcularVolumenCuerpos()

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)
```