PYTHON aplicado a la ingeniería Estructural y Geotécnica

Clase 01

Julio Sucasaca, Mag.

STRUCTURAL TECH

Ingeniería Estructural

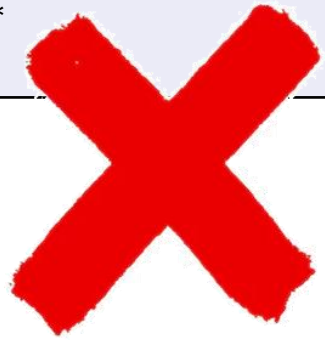
Contenido

- Guía de estilo
- Desempaquetado (*)
- Args y kwargs
- Cantidad variable de parámetros
- POO – Clases y Objetos
- POO – 4 pilares de POO
- POO – Herencia
- POO - Polimorfismo

Guía de estilo

- Imports al inicio del archivo. Evitar importar módulos completos
- `snake_case` para nombre de variables, funciones, atributos y métodos. `CamelCase` para clases.
- Un espacio después de “, “
- No usar tabs, usar 4 espacios para indentar.
- Lineas de código de máximo 100 caracteres.

```python
import numpy as np
class viga:
    def __init__(self,b,h,As,Asc):
        self.b=b;self.h=h
        self.As=As;self.Asc=Asc
        self.rho,self.rhoc=None,None
    def Calcular_As(self):
        self.rho = self.As/(self.b*self.h)
    def Calcular_Asc(self):
        self.rhoc=self.Asc/(self.b*self.h)
from random import *
import os
```



```python
import numpy as np
from random import choice, randint
import os

class VigaRectangular:
    def __init__(self, base, altura, area_acero_inf, area_acero_sup):
        self.base = base
        self.ancho = altura
        self.area_acero_inf = area_acero_inf
        self.area_acero_sup = area_acero_sup
        self.cuantia_inf = None
        self.cuantia_sup = None

    def calcular_cuantia_inf(self):
        self.cuantia_inf = self.area_acero_inf / (self.base * self.ancho)

    def calcular_cuantia_sup(self):
        self.cuantia_sup = self.area_acero_sup / (self.base * self.ancho)
```

Guía de estilo

- Nombre de variables descriptivos con la acción que realizan

```python
def funcion(l):
    i = 0
    a = l[0]
    for j in range(1, len(l)):
        var = l[j]
        if var > a:
            j = i
            a = var
    return i
```

```python
def indice_de_maximo(lista):
    indice_actual = 0
    maximo_actual = lista[0]
    for indice in range(1, len(lista)):
        elemento = lista[indice]
        if elemento > maximo_actual:
            indice_actual = indice
            maximo_actual = elemento
    return indice_actual
```

Guía de estilo

- Utilizar el mismo vocabulario para variables similares

```
get_user_info()
get_client_data()
get_customer_record()
```

```
get_user_info()
get_user_data()
get_user_record()
```

- Comentar el código para entender lo que sucede dentro del código

```
get_user_info()
get_client_data()
get_customer_record()
```

```
# Obtener informacion general del usuario
get_user_info()

# Obtener datos del usuario
get_user_data()

# Obtener historial de registros del usuario
get_user_record()
```

Desempaquetado (*)

- Función con parámetros por defecto

```python
def suma_y_producto(n1=0, n2=0):
    """ Funcion que devuelve la suma y producto de dos numeros
        Parametros:
        n1->int, float: primer número
        n2->int, float: segundo numero"""
    suma = n1 + n2
    producto = n1 * n2
    print(f"suma: {suma}, producto: {producto}")
    return suma, producto
```

```python
help(suma_y_producto)
```

```
Help on function suma_y_producto in module __main__:

suma_y_producto(n1=0, n2=0)
    Funcion que devuelve la suma y producto de dos numeros
    parametros:
    n1->int, float: primer número
    n2->int, float: segundo numero
```

- Cuando las funciones retornan más de un valor, lo hacen empaquetando todos los valores en una tupla.

```python
# Desempaquetado de elementos
sumprod = suma_y_producto()
sum = sumprod[0]
prod = sumprod[1]
```

```python
# Ignorar una salida
sum, _ = suma_y_producto()
```

- Podemos desempaquetar salidas de funciones con el operador *

```python
# Desempaquetar argumentos de funciones
numeros = (3.0, 5.0)
sum, prod = suma_y_producto(*numeros)
```

Args y kwargs

- Al llamar funciones, por defecto se usa el **orden** de los argumentos.

```
# Argumentos por defecto: en orden
def ejemplo(a, b, c):
    print(f'a: {a}, b: {b}, c: {c}')
```

```
ejemplo('hola', 'PIEG', 728)
```

```
a: hola, b: PIEG, c: 728
```

- Sin embargo, se pueden especificar en **desorden** utilizando su **nombre como palabra clave**.

```
# Argumentos en desorden: Usando nombre como palabra clave
ejemplo(b='PIEG', c=728, a='hola')
```

- En Python existe:
  - Argumento posicional (**args**) : Argumento, sigue el orden de definición.
  - Argumento por palabra clave (**kwargs**) : Argumento precedido por un identificador (nombre=) en llamado a función.
- Se pueden usar ambos a la vez, pero respetando las **reglas** siguientes:
  - No pueden existir args después de kwargs.
  - No se puede establecer kwargs de un args ya establecido.

```
# Argumentos en desorden: Usando nombre como palabra clave
ejemplo(b='PIEG', c=728, a='hola')
ejemplo('hola', 'PIEG', 728)
ejemplo('hola', 'PIEG', c=728)
ejemplo('hola', b='PIEG', c=728)
ejemplo('hola', c=728, b='PIEG')
ejemplo(a='hola', b='PIEG', c=728)
ejemplo(c=728, a='hola', b='PIEG')
```

```
# Lanzan errores:

# Posicional después de palabra clave
ejemplo(a='hola', 'PIEG', 728)

# Palabra clave vuelve a usar parámetro usado por
argumento posicional
ejemplo('hola', 'PIEG', a=728)
```

Args y kwargs

- Existen dos formas de establecer argumentos en llamada a función usando * y **
  - `función(*argumentos)` : `argumentos` es `lista` o `tupla` y se desempaquetan su contenido como `args`.
  - `función(**argumentos)` : `argumentos` es un `diccionario` y desempaquetan llave-valor como `kwargs`.

```
# Llamadas a función equivalentes
lista = ['hola', 'PIEG', 728]
tupla = ('hola', 'PIEG', 728)
diccionario = {'a': 'hola', 'b': 'PIEG', 'c': 728}

ejemplo(*lista)
ejemplo(*tupla)
ejemplo(**diccionario)
```

```
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
```

- Se pueden combinar el uso de todos los mecanismos anteriores, respectando las reglas anteriores.

```
# Combinando el uso de mecanismos anteriores
ejemplo('hola', *['PIEG', 728])
ejemplo(*['hola', 'PIEG'], 728)
ejemplo(*['hola', 'PIEG'], *[728])
ejemplo(*['hola', 'PIEG'], c=728)
ejemplo('hola', 'PIEG', **{'c': 728})
ejemplo(*['hola', 'PIEG'], **{'c': 728})
ejemplo(*['hola'], **{'c': 728}, b='PIEG')
ejemplo(*['hola'], **{'c': 728}, **{'b': 'PIEG'})
```

```
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
a: hola, b: PIEG, c: 728
```

Cantidad variable de parámetros

- Nos da flexibilidad para definir y llamar funciones. Hay varias formas de hacerlo
  - Usando valores de parámetros por defecto (el ejemplo siguiente se limita a un máximo de 3 argumentos)

```python
# Valores de parámetros por defecto
def ADAM(theta_0, rho1=0.90, rho2=0.99):
    pass
```

```python
# Uso
x_opt = ADAM(0.)
x_opt = ADAM(0., **{"rho1": 0.91})
```

  - Usando `*args` y `**kwargs`. Permite definir una cantidad arbitraria de argumentos posicionales y por palabra clave.
    - El nombre args y kwargs solo es una convención, puede tener cualquier nombre de variable de Python
    - Solo se puede usar un `*args`, y un `**kwargs`, y en ese orden.

- La forma más general:

```
funcion(arg1, arg2, *args, kwarg1, kwarg2, **kwargs)
```

```python
# Ejemplo completo
def ejemplo_completo(a, b, *otros_posicionales, c="True", d=False, **otros_palabra_clave):
    print(f'arg1: {a}')
    print(f'arg2: {b}')
    print(f'*args: {otros_posicionales}')
    print(f'kwarg1: {c}')
    print(f'kwarg2: {d}')
    print(f'**kwargs: {otros_palabra_clave}\n')

ejemplo_completo("Hola", "MIE", *(7, 2, 8), c="False", d=True, **{"e": "MIC", "f": 617} )
ejemplo_completo("Hola", "MIE", c="False", d=True)
ejemplo_completo("Hola", "MIE", c="False")
ejemplo_completo("Hola", "MIE")
```

```
arg1: Hola
arg2: MIE
*args: (7, 2, 8)
kwarg1: False
kwarg2: True
**kwargs: {'e': 'MIC', 'f': 617}

arg1: Hola
arg2: MIE
*args: ()
kwarg1: False
kwarg2: True
**kwargs: {}

arg1: Hola
arg2: MIE
*args: ()
kwarg1: False
kwarg2: False
**kwargs: {}

arg1: Hola
arg2: MIE
*args: ()
kwarg1: True
kwarg2: False
**kwargs: {}
```

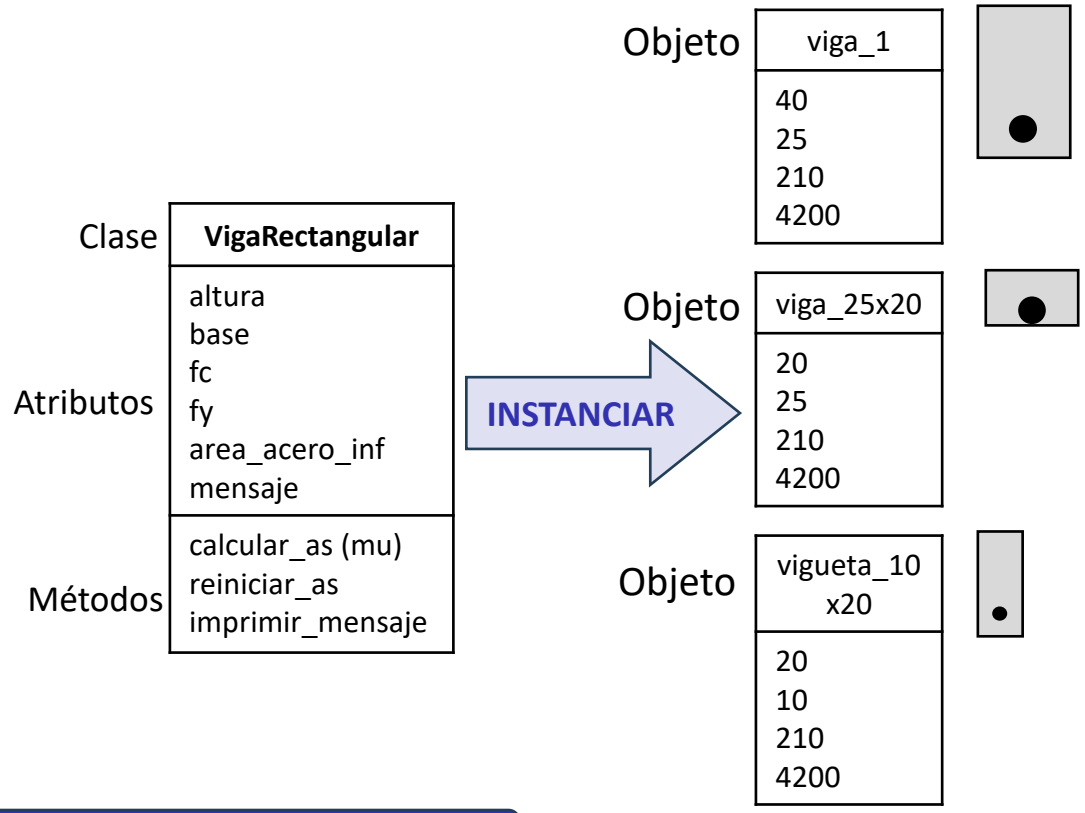
POO – Paradigmas de programación

POO – Clases y Objetos

En POO (*Object-Oriented Programming*) los programas modelan las funcionalidades a través de la interacción entre objetos por medio de sus datos y sus comportamientos.

POO → Nuevo tipo de dato: **OBJETO**

- En POO los objetos son descritos de manera general mediante **clases** (Plantilla para crear objetos)
  - Los **atributos** son descripciones de los datos que caracteriza a un objeto.
  - Los **métodos** son descripciones de los comportamientos de los objetos. Acciones que se realizan sobre el objeto
- Cada vez que creamos un objeto a partir de una clase definida, decimos que estamos **instanciando** esa clase. **Un objeto es una instancia de una clase**.
  - La instanciación la realiza el **método constructor**: `.__init__()`
  - Python lo llama automáticamente para cada objeto creado.
  - Inicializa atributos del objeto.
  - `self` representa la instancia de una clase. Es una convención

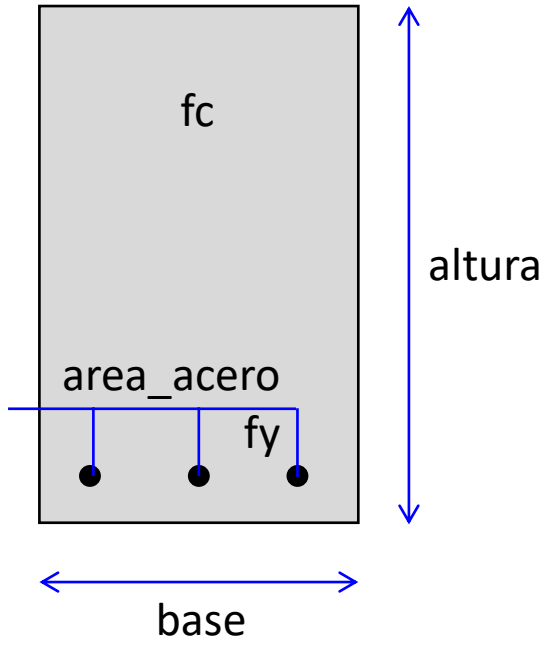


Cuidado al instanciar objetos

```
# Instanciando objetos
viga_1 = VigaRectangular(“vig”, 40, 25, 210, 4200) # Objeto

# Copiando referencia a clases
viga_1 = VigaRectangular   # Copia de clase (OJO)
```

Ejemplo: Modelar la clase `VigaRectangular` con los atributos y métodos indicados. Luego, sobreescribir los métodos `__str__` y `__del__`

VigaRectangular
+ nombre: str + altura: float + base: float + fc: float + fy: float + area_acero: float + mensaje: str
+ calcular_as (mu): float + reiniciar_as(): None + imprimir_mensaje(): str

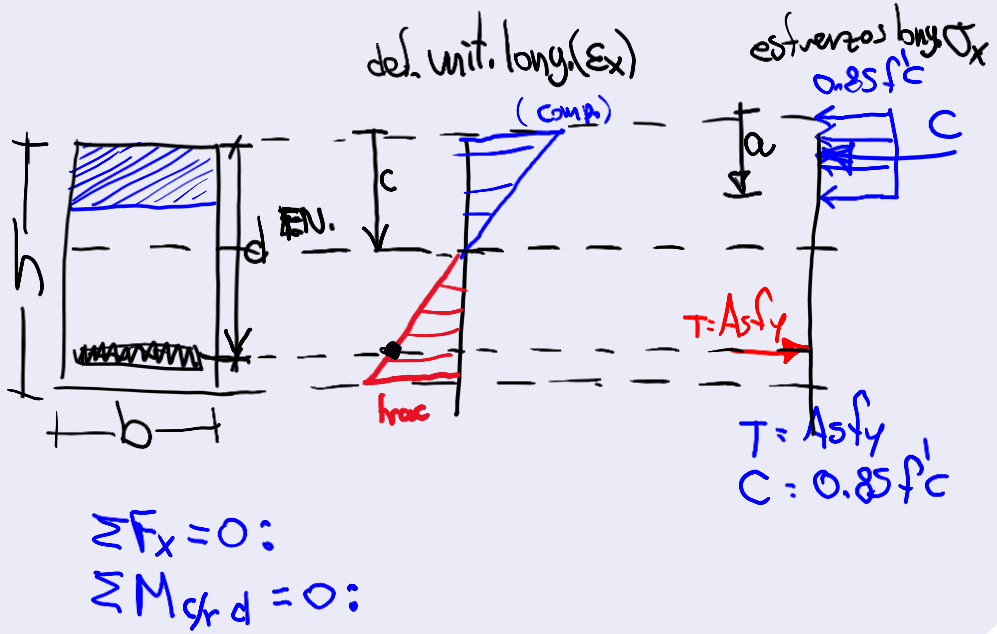


Recordar los parámetros y definiciones de diseño de vigas a flexión en concreto armado:

$$A_s = \frac{M_u}{\phi f_y \left(d - \frac{a}{2}\right)}$$

$$a = \frac{A s f_y}{0.85 f_c b}$$

Que provienen de un análisis seccional simplificado.



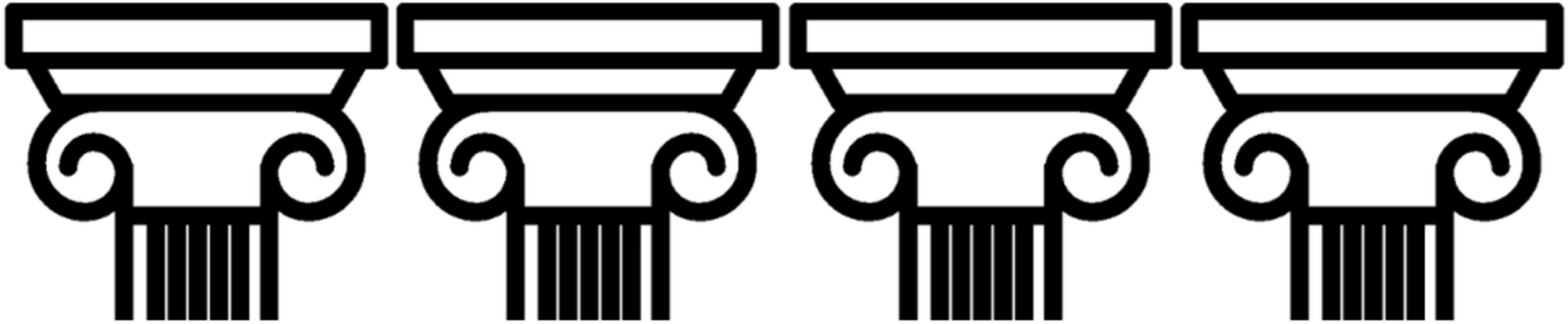
POO – Clases y Objetos

En Python casi todo, hasta las clases, son objetos.

```
# En python, casi todo, hasta las clases, son objetos:
print(type(True))
print(type("abc"))
print(type(1.0))
print(type(1))
print(type(viga_1))             # __main__ significa que la clase se definio en el script
print(type(VigaRectangular))    # Las clases son instancias de la clase type
```

```
<class 'bool'>
<class 'str'>
<class 'float'>
<class 'int'>
<class '__main__.VigaRectangular'>
<class 'type'>
```

POO – 4 pilares de POO



POO – Herencia

Concepto:

- Es un concepto que nos permite reutilizar código mediante **especialización** y **generalización** entre clases.
- Una clase **hereda** atributos y métodos de otra clase.
  - Quién hereda es una: subclase, especialización, extensión
  - La otra es la: superclase, clase padre/madre, clase superior.
- La clase quien hereda, además de lo heredado, tiene sus atributos y métodos específicos

Acceder a métodos de clase superior

- Especificando nombre: `NombreSuperClase.metodo(self, args)`
- Usando super(). : `super().metodo(args)`

Sobreescritura (*overriding*):

- Se puede **sobreescribir** métodos volviendo a definir el mismo nombre del método de la clase padre.

Comportamiento normal:

```
lista = [1,2,3,4]
print(lista)
```

```
[1, 2, 3, 4]
```

Comportamiento especializado:

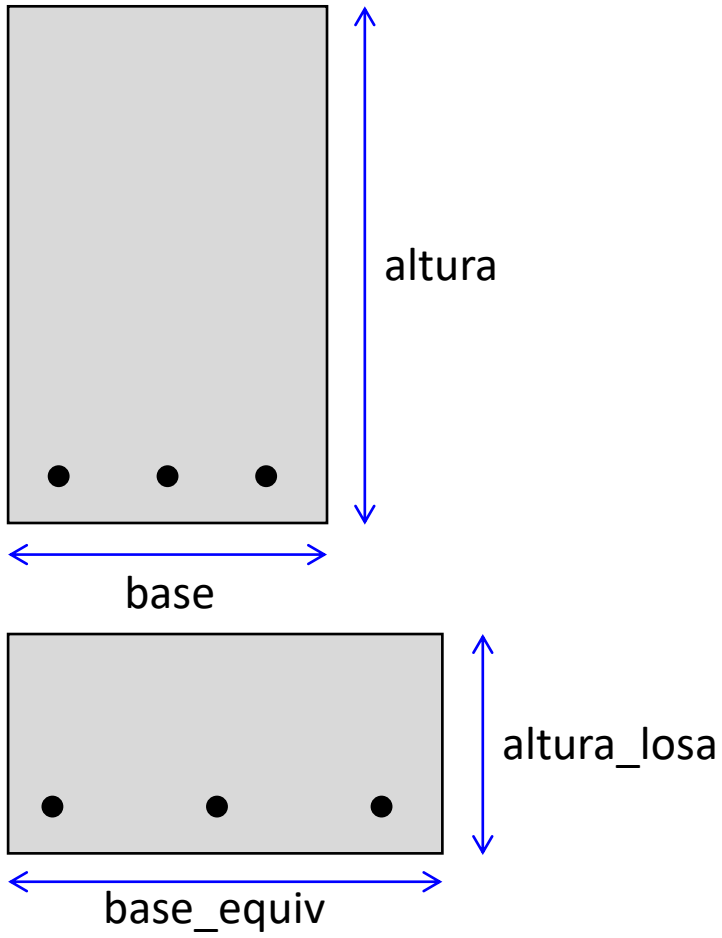
```
class MiLista(list):

    def __str__(self):
        print("Sobreescribí este metodo")
        print("para que no imprima la lista")
        print("sino, este texto")
        return "y que regrese texto innecesario"
```

```
mi_lista = MiLista(lista)
print(mi_lista)
```

```
Sobreescribí este metodo
para que no imprima la lista
sino, este texto
y que regrese un texto innecesario
```

Ejemplo: Modelar la clase `VigaChataEquivalente` que hereda de la clase `VigaRectangular`, incluir y modificar los atributos y métodos indicados en color rojo.

VigaRectangular
+ nombre: str + altura: float + base: float + fc: float + fy: float + area_acero: float + mensaje: str
+ calcular_as (mu): float + reiniciar_as: None + imprimir_mensaje: str

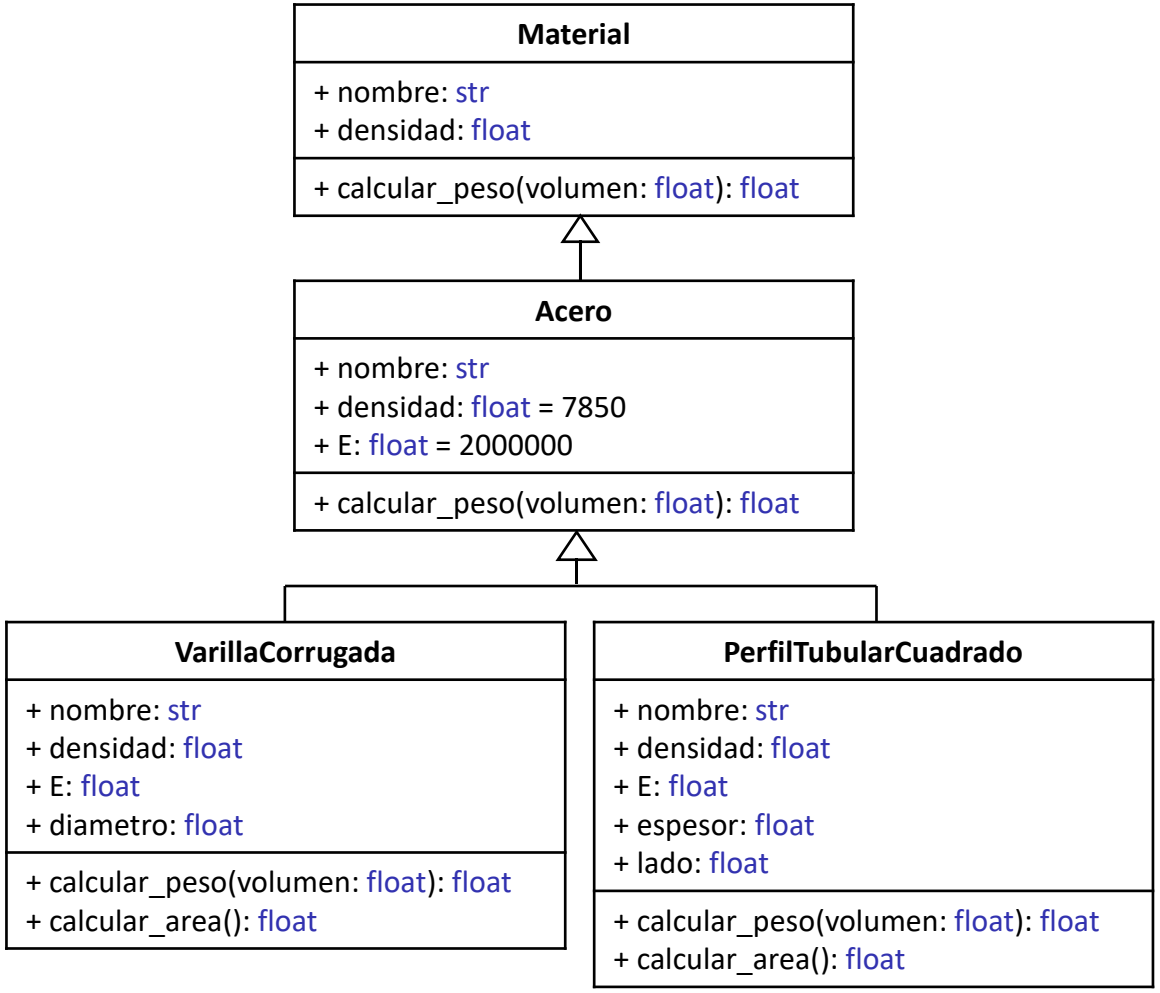


$$\text{base\_equiv} = \frac{(\text{base})\ (\text{altura})^3}{\text{altura\_losa}^3}$$

VigaChataEquivalente
+ nombre: str + altura: float + base: float + base_equivalente: float + altura_losa: float + fc: float + fy: float + area_acero: float + mensaje: str
+ calcular_as (mu): float + reiniciar_as: None + imprimir_mensaje: str

Recordando herencia

- La herencia puede tener muchos niveles y presentan jerarquía lineal y especialización gradual

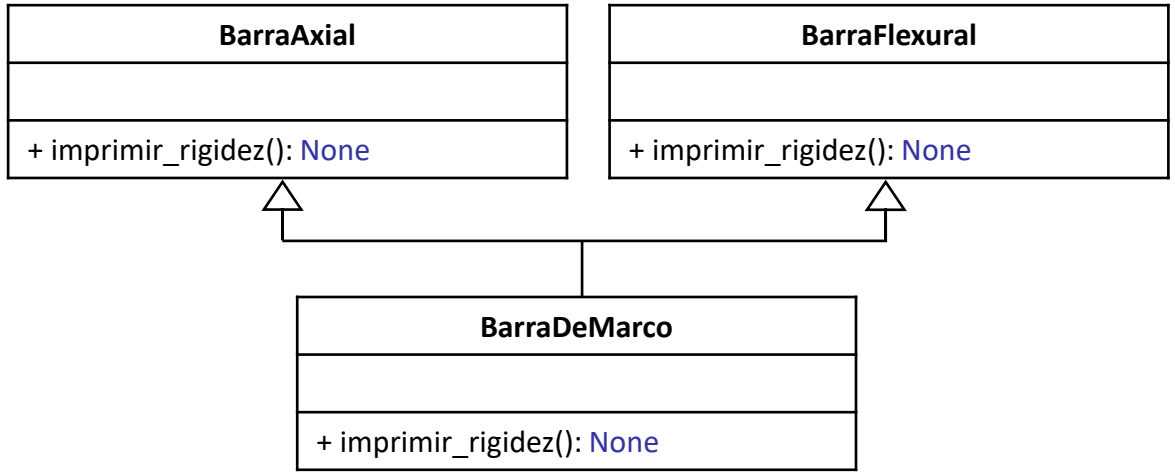



```python
class Material:

    def __init__(self, nombre, densidad):
        self.nombre = nombre
        self.densidad = densidad

    def calcular_peso(self, volumen):
        return volumen * self.densidad

class Acero(Material):

    def __init__(self, nombre, E=2000000, densidad=7850):
        self.E = E
        super().__init__(nombre, densidad)

class VarillaCorrugada(Acero):

    def __init__(self, nom, diam):
        super().__init__(nom)
        self.diametro = diam

    def calcular_area(self):    # Nuevo metodo
        return 3.14159265/4*(self.diametro)**2

class PerfilTubularCuadrado(Acero):

    def __init__(self, nombre, espesor, lado):
        super().__init__(nombre)
        self.espesor = espesor
        self.lado = lado

    def calcular_area(self):        # Sobreescribo método
        return 4 * self.lado * self.espesor
```

POO – Multiherencia

Concepto

- Heredar atributos y métodos de más de una clase a la vez.
- Se usa cuando un objeto **combina** características de otros de **forma natural**. Ejm: La barra de marco es a la vez una barra con comportamiento axial y flexural.



- Puede ser ambiguo, ¿De quién hereda el método `imprimir_rigidez()`?. En esos casos, tener claro el orden de resolución de métodos (MRO)
  - Primero se busca el método en la clase `BarraDeMarco`
  - Si no existe, se busca en la primera clase padre: `BarraAxial`
  - Si no existe, se busca en la segunda clase padre: `BarraFlexural`
- Es decir, importa el **orden de definición** de las clases padre.

```python
class BarraAxial:

    def imprimir_rigidez(self):
        print("Esta es la rigidez axial")

class BarraFlexural:

    def imprimir_rigidez(self):
        print("Esta es la rigidez flexural")

class BarraDeMarco(BarraAxial, BarraFlexural):
    pass

# Instanciando un objeto de clase BarraDeMarco
barra = BarraDeMarco()
barra.imprimir_rigidez()
```

- En otros casos más complejos, es útil ver la estrategia del interprete de Python para dicha clase usando el atributo: `Clase.__mro__`

mro

```python
# Orden de resolución de métodos
BarraDeMarco.__mro__
```

```
(__main__.BarraDeMarco, __main__.BarraAxial, __main__.BarraFlexural, object)
```

- La multiherencia debe usarse cuidadosamente.

POO – Polimorfismo

Concepto

- *Cualidad de tener o tomar múltiples formas*. (ver https://es.wikipedia.org/wiki/Polimorfismo_%28inform%C3%A1tica%29)
- Permite incorporar nuevos comportamientos sin alterar el código.
- Utilizar objetos de distinto tipo, con la misma interfaz (Interfaz: modelo para el diseño de clases).
- La interfaz se mantiene inalterada pero cambia el comportamiento en función del objeto que estemos usando. Entonces, si un objeto tiene los métodos que nos interesan, nos basta con eso, su clase es irrelevante.

```
# Instanciando objetos de diferentes clases
barra_de_armadura = BarraAxial()
barra_de_emparrillado = BarraFlexural()

# Funcion para mostrar la rigidez de una barra
def funcion_mostrar_rigidez(barra):
    return barra.imprimir_rigidez()

# Demostrando polimorfismo
funcion_mostrar_rigidez(barra_de_armadura)
funcion_mostrar_rigidez(barra_de_emparrillado)
```

Objetos diferentes

Los objetos son capaces de responder al mensaje que se les envia

Ejemplos de polimorfismo

- **Sobreescritura** (*overriding*) de métodos en una subclase.
- **Sobrecarga** (*overloading*). El mimo operador funciona de distinta forma de acuerdo al tipo de argumento que recibe. Ejemplo de operador `+`. Podemos personalizar con el método `__add__()`

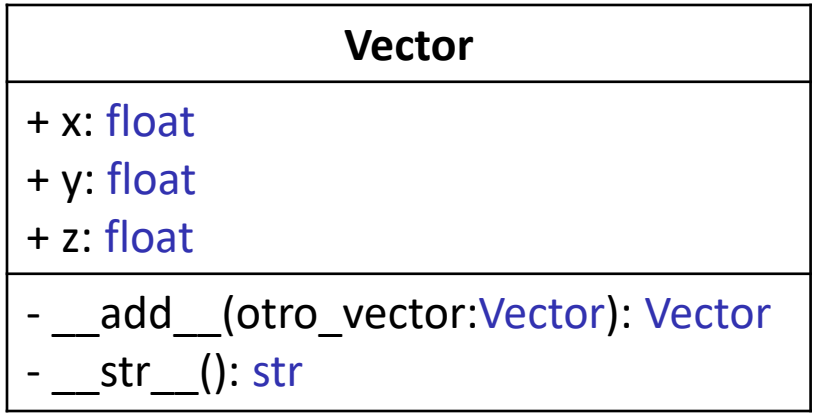
```
# Operador +
print( "MIE" + "728")
print( 123 + 456)
print( [1,2,3] + [4,5,6])
```

- Podemos personalizar otros operadores.

Menor que	: `__lt__`
Menor o igual que	: `__le__`
Mayor que	: `__gt__`
Mayor o igual que	: `__ge__`
Igual que	: `__eq__`
Distinto	: `__ne__`

POO – Polimorfismo

Ejemplo: Implementar la clase `Vector` y definir la acción que el operador `+` y la función `print()` realizarán sobre objetos de clase `Vector`.



POO – Abstracción

Concepto

- Clase cuya intensión no es ser instanciada, sino, usarse como parte del modelamiento para otras subclases. Establecer su estructura.
- Tienen **métodos abstractos**, que son comportamientos que las subclases **deben** tener, implementar y sobreescribir.
- Es una interfaz que define al conjunto de métodos que debe tener un objeto.
- Clase abstracta:
  - Define una interfaz para todas las subclases.
  - Clase que no debería instanciarse directamente
  - Contiene uno o más métodos abstractos.
  - Sus subclases implementan todos sus métodos abstractos
- En Python, se simulan las interfaces usando el módulo `abc` (*Abstract Base Classes*) y el decorador `@abstractmethod`, que define una forma de crear interfaces y que fuerzan a las subclases a implementar los métodos.

```python
from abc import ABC, abstractmethod

# Definiendo la clase Base, asbstracta
class Barra(ABC):           # Al heredar de ABC, la clase Base: Barra, se define como abstracta
    @abstractmethod         # El decorador, hace que el metodo imprimir_rigidez() sea abstracto.
    def imprimir_rigidez(self):
        pass
    @abstractmethod
    def calcular_rigidez(self):
        pass
```

Métodos abstractos

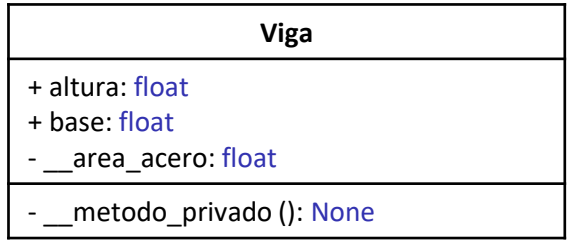
```python
# Definiendo subclases

class BarraAxial(Barra):

    def imprimir_rigidez(self):
        print("Esta es la rigidez axial")

    def calcular_rigidez(self):
        print("La rigidez axial se calculo")

class BarraFlexural(Barra):

    def imprimir_rigidez(self):
        print("Esta es la rigidez flexural")
```

Esta clase no implemento método `calcular_rigidez()`. Generará error

POO – Encapsulamiento

Concepto

- **Ocultar** los estados internos (atributos y métodos) al exterior.
- Encapsular consiste en hacer que los atributos y métodos a una clase no se puedan acceder ni modificar desde afuera, sino que solamente el propio objeto pueda acceder a ellos mediante métodos definidos por el programador.
  - Razón: Algunos atributos son de interés solamente para el comportamiento interno del objeto, y no son de interés para otros objetos. Por tanto, el código es más limpio si este atributo permanece oculto o encapsulado dentro del objeto.
- En Python, se **simula** el encapsulamiento nombrando a los atributos o métodos ocultos con doble guion bajo.




```
class Viga():

    def __init__(self, peralte, ancho, area_acero):
        self.altura = peralte
        self.base = ancho
        self.__area_acero = area_acero  # Atributo oculto

    def __metodo_privado(self): # Método oculto
        print("No deberias usarme porque soy privado")
```

- En Python, realmente todos los atributos y métodos son siempre públicos. La simulación anterior tiene el objetivo de disminuir la posibilidad de errores del programador al definir atributos o métodos privados en un lenguaje donde no existen. Es una buena práctica modelar de esta forma los atributos y métodos ocultos.

```
# Instanciando objeto de clase Viga
viga_1 = Viga(peralte = 0.4, ancho = 0.25, area_acero = 5.94)

# Mostrando atributos públicos
print(viga_1.altura)
print(viga_1.base)

# No podemos mostrar atributos ni métodos ocultos
# Genera error, no podemos acceder, el encapsulamiento funcionó!
print(viga_1.__area_acero)
viga_1.__metodo_privado()
```

- En el fondo, el truco anterior solo cambia el nombre del atributo o método "oculto" `self.__atributo` por `self._Clase__atributo`. (*Name mangling*)

```
# Realmente solo fue un truco de cambio de nombre
print(viga_1._Viga__area_acero)# No debería usarlo, igual lo hago
viga_1._Viga__metodo_privado()# No debería usarlo, igual lo hago
```

POO – Decoradores

Recordando: ¿Cuál es la diferencia?

`variable_1 = función()` : Llama a la función y devuelve sus salidas

`variable_2 = función` : Guarda una referencia a la función

- Las funciones también son objetos, y se pueden guardar en variables, ser usadas como argumentos y salidas de otras funciones.

```
def funcion(otra_funcion):
    return otra_funcion
```

Concepto

- Un decorador **es una función** que toma otra función como argumento, y devuelve una versión modificada de esa función, la función decorada, sin cambiar su código original. Añade funcionalidades. Evitar herencias con jerarquías complejas.
- Es como un **envoltorio** (*wrapper*) sobre la función original.

```
# Ver ejemplo ilustrativo usando
funciones dentro de funciones
```

Sintaxis

- Se aplica mediante `@nombre_decorador` inmediatamente arriba de la definición de la función o método que será decorada.

```
# Ver ejemplo de aplicación de
decoradores usando @nombre_decorador
```

Decoradores más usados:

`@property`

- Viene por defecto en Python
- Se usa para decorar un método que actúe como un atributo de solo lectura. Simula de mejor forma atributos privados según el concepto de encapsulamiento (ver https://ellibrodepython.com/decorador-property-python)

`@abc.abstractmethod`

- Función que se importa del módulo `abc`.
- Define una forma de crear interfaces y fuerza a las subclases a implementar los métodos.

`@staticmethod`

- Viene por defecto en Python
- Definen métodos que perteneciendo a una clase, no requieren de los atributos de la instancia. Es decir, no requieren el primer argumento `self`, y actúan como funciones normales. Implementan funcionalidades.

POO – Decoradores

Ejemplo: Modelar la clase `BarraAxial` que tiene el método `calcular_deformación()`, y decorarla:

1. Decorador que no haga nada : `@no_hago_nada_nuevo`
2. Decorador que valide que la fuerza axial sea menor a 10 : `@validar_entrada`
3. Decorador que mida el tiempo de cómputo del método : `@tiempo_ejecucion`

Además, la clase debe implementar una funcionalidad para sumar dos números

$$\delta = \frac{PL}{EA}$$

BarraAxial
+ area: float + modulo_elasticidad: float + longitud: float
+ calcular_deformacion(fuerza_axial: float): float + <static> calcular_suma_dos_números(num_1: float, num_2: float): float

```python
# Ayuda de Código inicial
class BarraAxial:

    def __init__(self, area, modulo_elasticidad, longitud):
        self.area = area
        self.modulo_elasticidad = modulo_elasticidad
        self.longitud = longitud

    def calcular_deformacion(self, fuerza_axial):
        time.sleep(0.3)    # Simulando un calculo costoso computacionalmente
        deformacion = fuerza_axial * self.longitud / (self.modulo_elasticidad * self.area)
        print(f"La deformación es: {deformacion}")
        return deformacion
```

POO – Diagrama de clases

Concepto

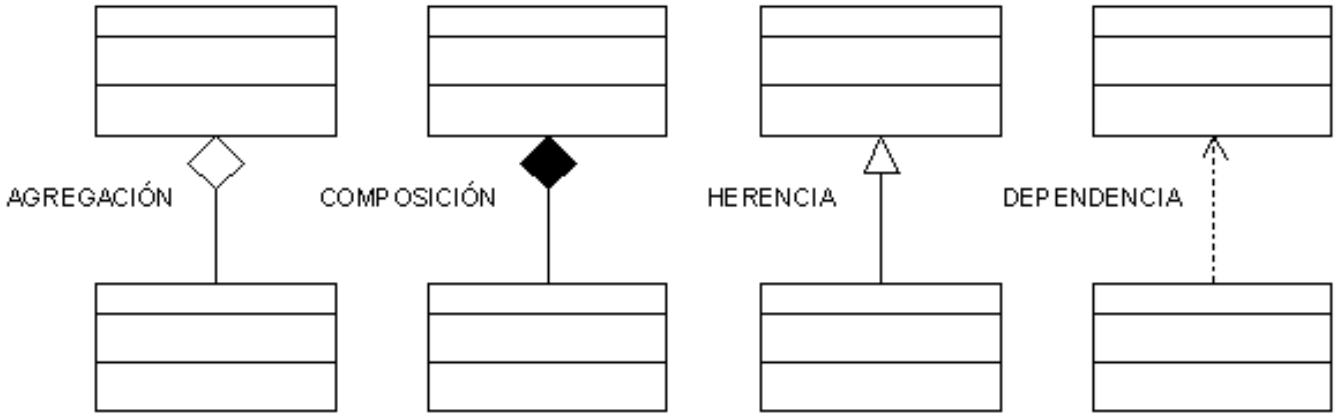
- Forma gráfica de visualizar **interacción entre clases**, atributos y métodos que componen un programa.

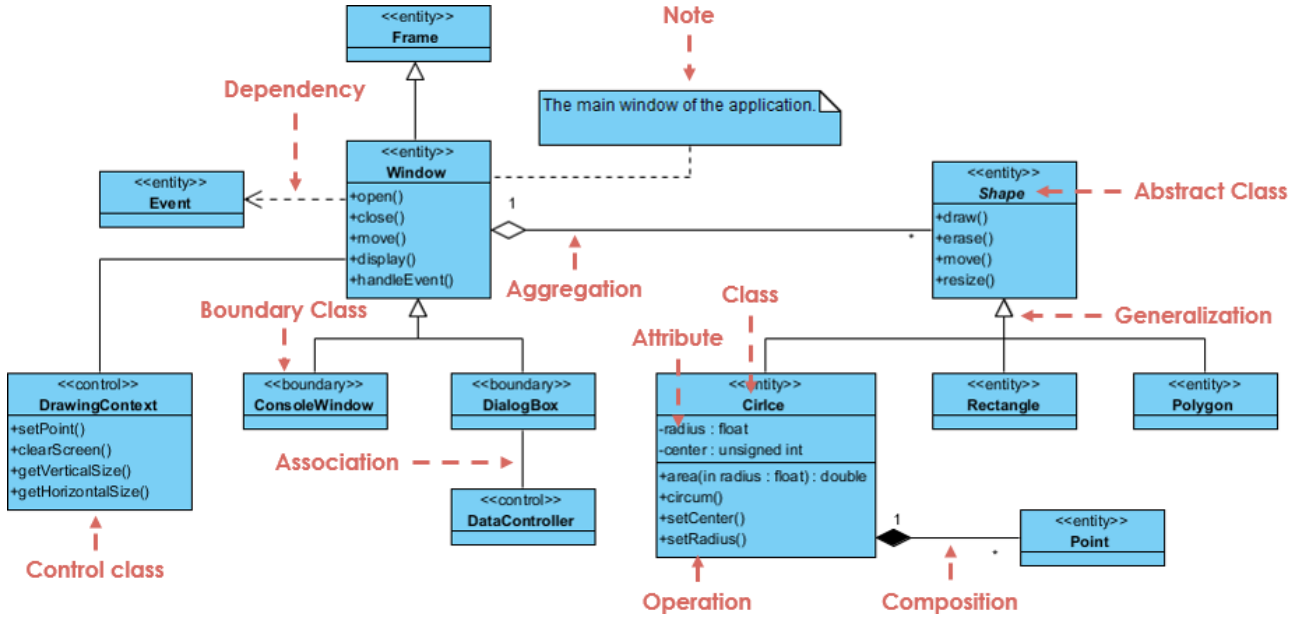
Relaciones

Contención: Un objeto contiene a un objeto de otra clase como atributo.

- **Composición** (⧫): El tiempo de vida del objeto contenido **depende** de la existencia del objeto que lo contiene. **No tienen** sentido como clases independientes.
- **Agregación** (◊): El tiempo de vida del objeto contenido es **independiente** de la existencia del objeto que lo contiene. **Tienen** sentido como clases independientes.

Herencia: Relación de especialización.

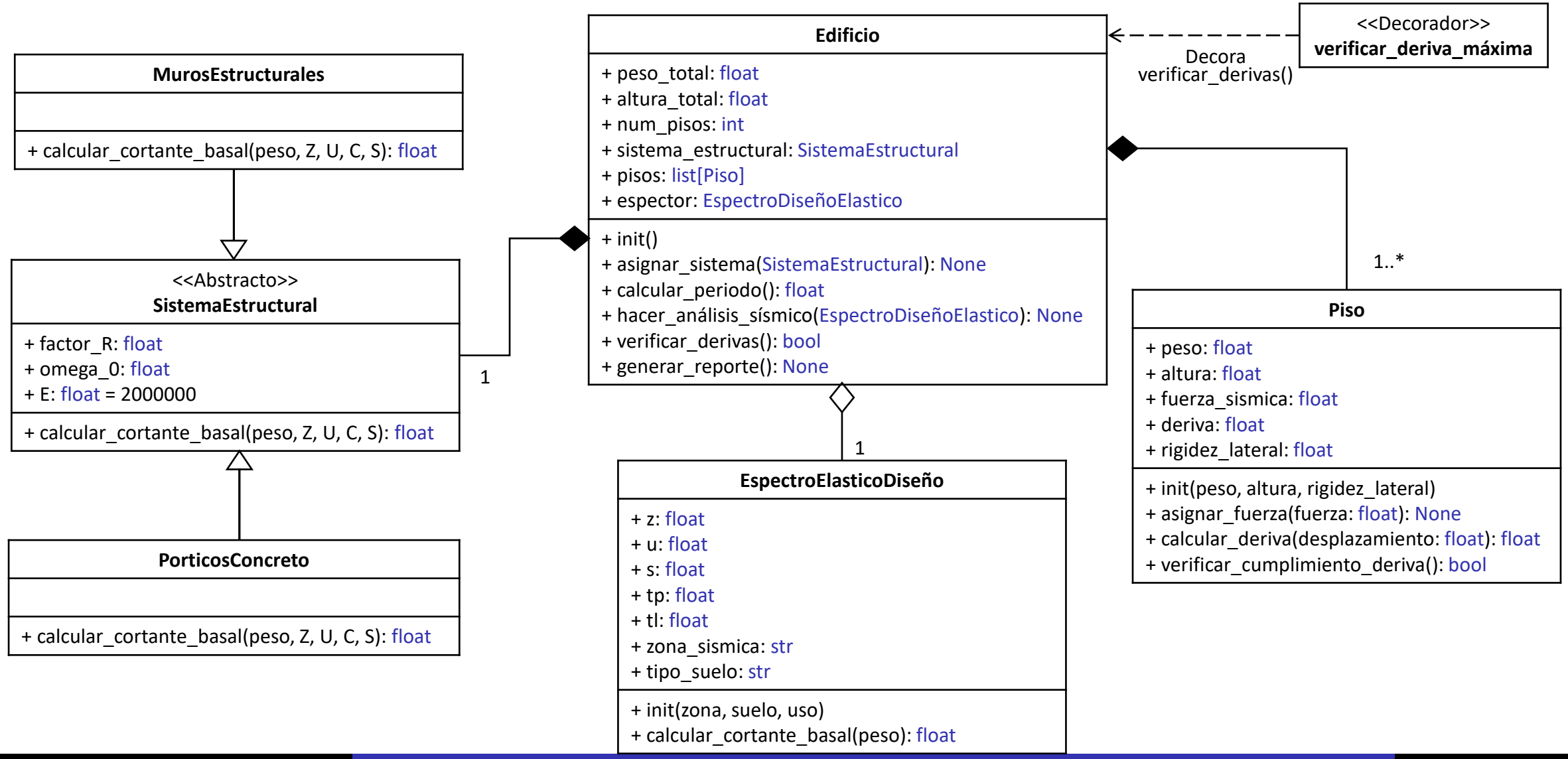




Mas información: Buscar en internet, por ejemplo https://www.cybermedian.com/es/a-comprehensive-guide-to-uml-class-diagram/

POO – Diagrama de clases

Un ejemplo “simple” de programa que realiza análisis sísmico



Gracias por su atención

Julio Sucasaca, Mag.
j.sucasacar@gmail.com