Lenguaje de Programación Python

Soporte Orientado a Objetos: Clases y Objetos

Dr. Mario Marcelo Berón
Argentina Programa
Universidad Nacional de San Luis

Índice

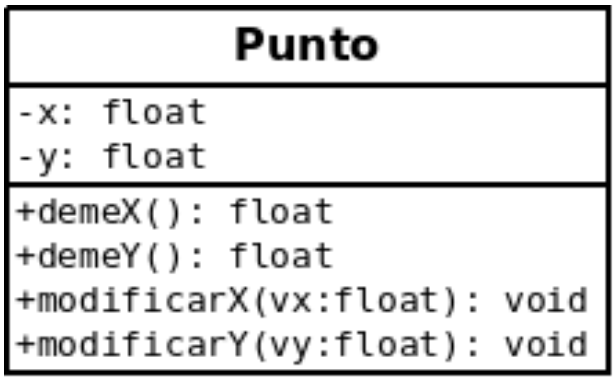
1. Clases
2. Alcance
3. Objetos
4. Relaciones entre Clases
   - Herencia
   - Herencia Múltiple
5. Polimorfismo
6. Excepciones

Python: Clases

Las clases son moldes para crear objetos. Una clase consta de características las cuales pueden ser:

- Atributos (Estado).
- Rutinas (Métodos-Comportamiento) algunas de estas rutinas son especiales dado que permiten la creación de instancias.

Representación en UML



Python: Clases

Para definir una clase en Python se sigue el siguiente patrón:

```
class NombreClase:
  cuerpo
```

Clase Punto - UML

Clase Punto - Python

```
class Punto:
  pass
```

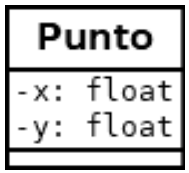
Las sentencia *pass* no hace nada.

Python: Clases

Variables de instancia:

```
class NombreClase:
  def __init__(self,param.):
    cuerpo_variables
```

Clase Punto - UML



Clase Punto - Python

```
class NombreClase:
  def __init__(self,x,y):
    self.x=x
    self.y=y
```

Comentarios

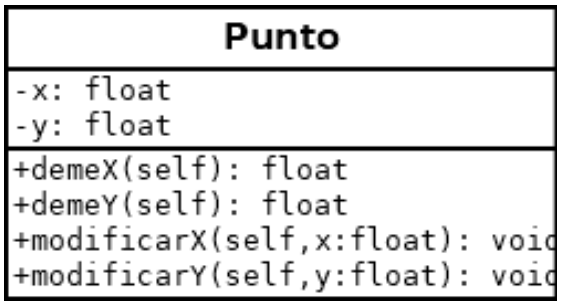
- Los métodos se distinguen de las funciones porque llevan el parámetro *self* que es el objeto receptor.
- Las variables de instancia se asocian al *self*.

Python: Clases

Métodos de instancia:

```
class NombreClase:
 def __init__(self,param.):
  cuerpo_variables
 def método(self,param.):
  cuerpo
 ...
```

Clase Punto - UML



Clase Punto - Python

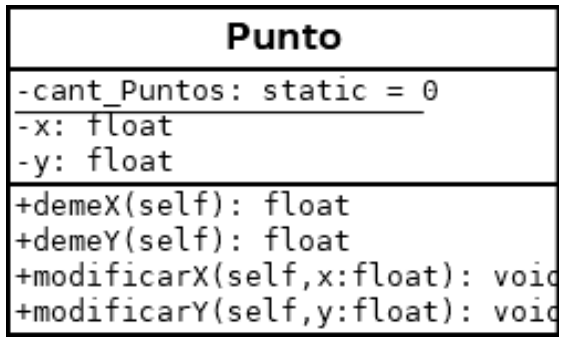
```
class NombreClase:
 ...
 def demeX(self):
  return self.x
 ...
```

Python: Clases

Variables de Clase:

```
class NombreClase:
 variables de clase
 def __init__(self,param.):
  cuerpo _ variables
 def método(self,param.):
  cuerpo
 ...
```

Clase Punto - UML



Clase Punto - Python

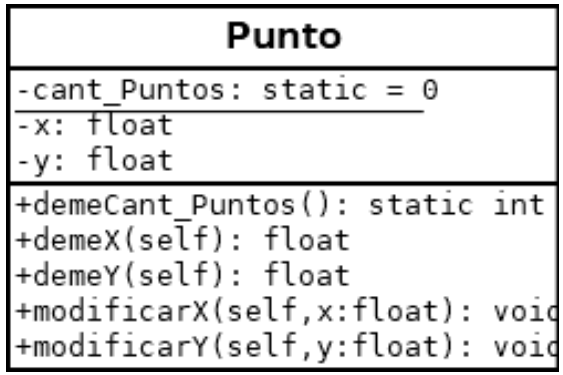
```
class NombreClase:
 cant_Puntos=0
 def demeX(self):
  return self.x
 ...
```

Python: Clases

Método de Clase:

```
class NombreClase:
 variables de clase
 def __init__(self,param.):
  cuerpo _ variables
 @classmethod
 def mClase(cls,param.):
  cuerpo
 ...
```

Clase Punto - UML



Clase Punto - Python

```
class NombreClase:
 cant_Puntos=0
 @classmethod
 def demeCant_Puntos(cls):
  return cant_Puntos
 ...
```

Python: Alcance

Python no tiene modificadores de acceso. Las variables se pueden acceder directamente desde el exterior.

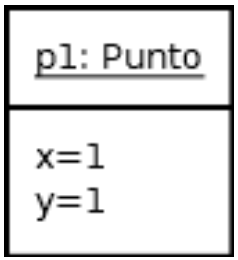
Para limitar el acceso a las variables se utiliza una convención que consiste en agregar al identificador un prefijo consistente de ___.

Python: Objetos

Objeto: instancia de tiempo de ejecución de una clase.

La representación UML es como se muestra en la figura.

La forma del objeto está determinada por los atributos.



Python: Objetos

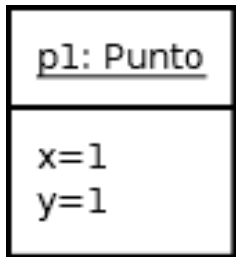
Los campos de los objetos pueden ser tipos básicos u otros objetos.

Un objeto se crea de la siguiente manera:

```
p1=Punto()
```

Con un constructor con parámetros la creación del punto mostrado en la figura se realiza como sigue:

```
p1=Punto(1,1)
```



ver 

Python: Objetos

En una sentencia de asignación lo que se copia es la dirección del objeto.

```
p=Punto(1,1)
q=p
```

q no tiene una copia del objeto sino que p y q apuntan al mismo objeto. Para comparar objetos se debe proporcionar un método para tal fin caso contrario lo que se comparan son referencias.

```
p=Punto(1,1)
q=Punto(1,1)
p==q
False
```

Produce False porque la dirección de p es distinta de la dirección de q.

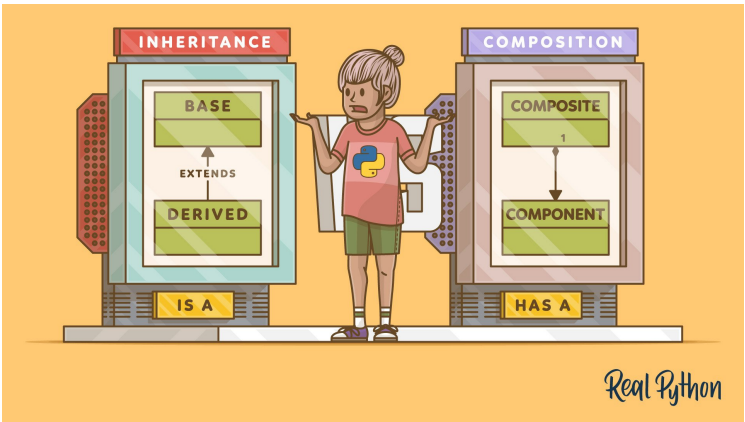
Python: Objetos

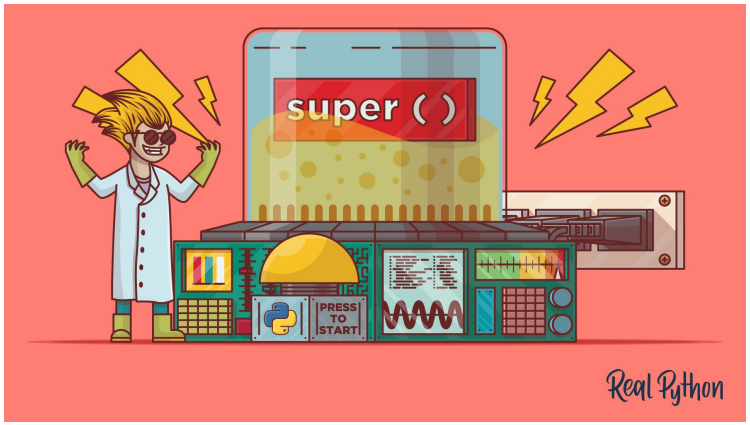
Las operaciones que se sugieren provea una clase para permitir la manipulación adecuada de objetos por parte del programador son las siguientes:

- Creación de Objetos
- Observadores
- Modificadores
- Comparación
- Clonación Superficial
- Clonación Profunda
- Copia Superficial
- Copia Profunda

Python: Herencia

Es una relación que existe entre las clases donde una clase (conocida como descendiente propio, subclase) hereda las características (atributos y rutinas) de otra clase padre (conocida como ancestro propio, superclase).



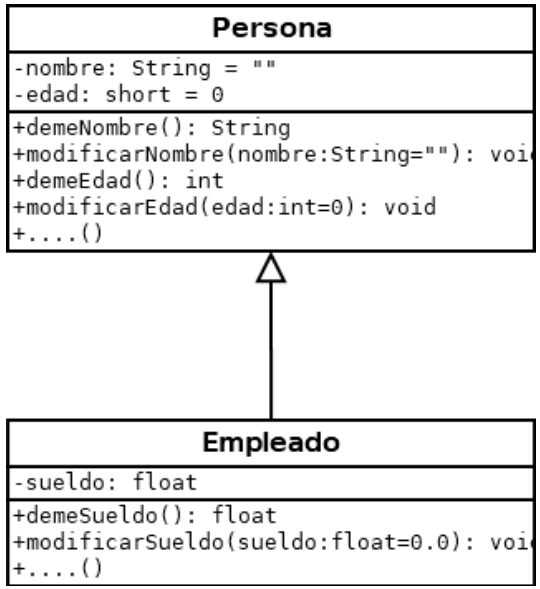


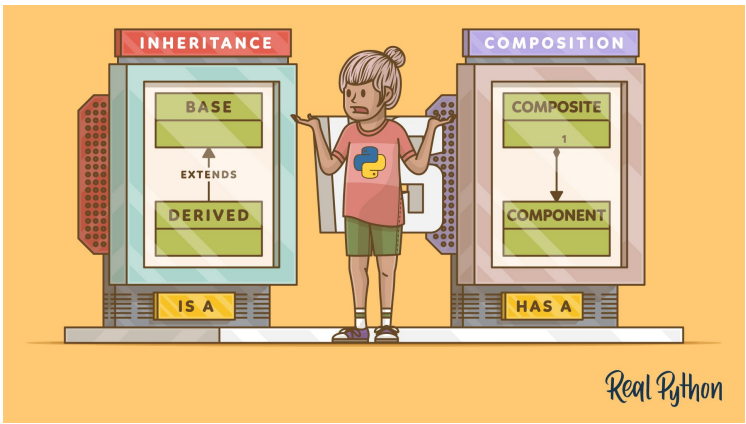
- Cualquier instancia de un heredero puede ser vista como una instancia de su padre pero no a la inversa.
- El código del heredero debe indicar quién es su padre pero no a la inversa.

Python: Herencia

Si B es una subclase de A entonces:

- B hereda todas las variables y métodos que no son privados.
- B puede definir nuevas variables y métodos de instancia propios.

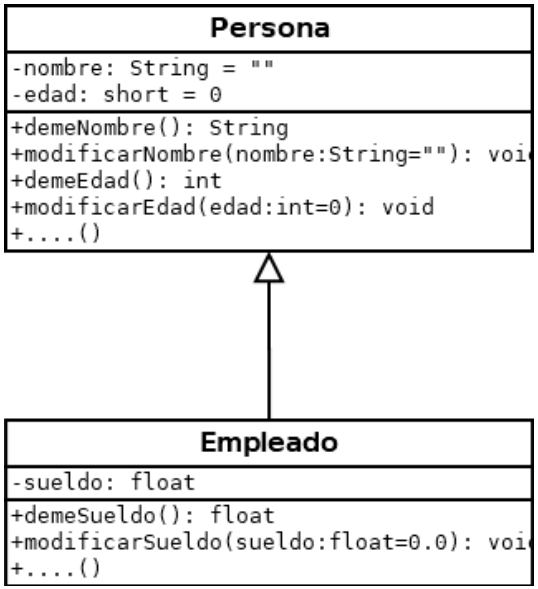





```
class Persona:
    #Operaciones
    ....

class Empleado(Persona):
    #Operaciones
    ...
```

Python: Herencia

- *Persona* es la super clase, clase padre o clase base.
- *Empleado* es la subclase, clase hijo o clase derivada.



Python: Redefinición de Variables y Métodos

```
class A:
 def __init__(self,a):
  self.a=a
  ......

class B (A):
 def __init__(self,a,b):
  super().__init__(a)
  self.a=b
```

Cuando se redefine una variable Python se queda con la última definición. No hay dos copias de la variable

Python: Redefinición de Variables y Métodos

```
class A:
 def __init__(self,a):
  self.a=a
 def imprimir(self):
  print("A:",self.a)

class B (A):
 def __init__(self,a,b):
  super().__init__(a)
  self.b=b
 def imprimir(self):
  super().imprimir()
  print("B:",self.b)

b=B(100,200)
b.imprimir()
```

En el caso de los métodos se puede invocar al método de la superclase produciendo los resultados esperados.

Python: Herencia Múltiple

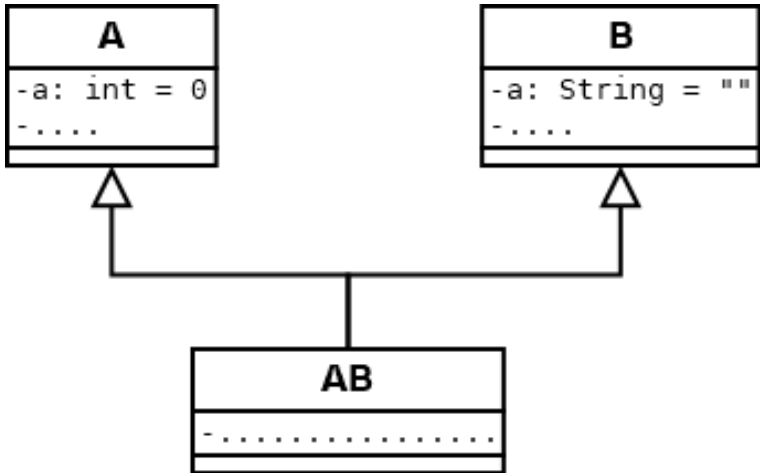
Herencia Múltiple: cuando una clase hijo tiene más de una clase padre.


```
class Avión:
    ....

class Activo:
    ....

class AviónCompania(Avión, Activo):
    ....
```

Python: Herencia Múltiple

Herencia Múltiple Problemas:

- ***Ambiguedad***
- Problema del Diamante

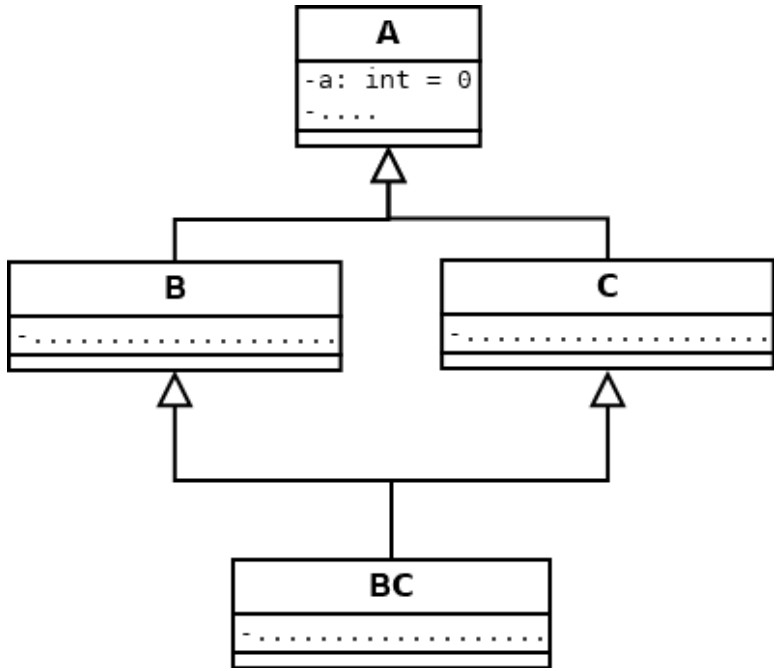



```
class A:
    .... #define a con int

class B:
    ....#define a con string

class AB(A, B):
    ....# a int o string?
```

Python: Herencia y Múltiple

Herencia Múltiple Problemas:

- Ambiguedad
- ***Problema del Diamante***

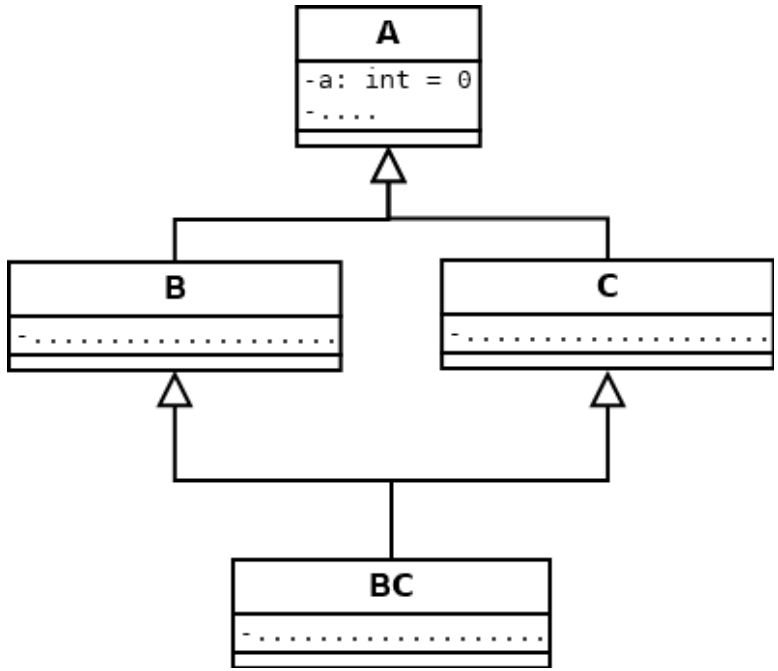



```
class A:
    .... #define a con int

class B:
    ....#hereda a
```

```
class C:
....#hereda a

class AB(B, C):
....# dos a int
```

Python: Herencia y Múltiple

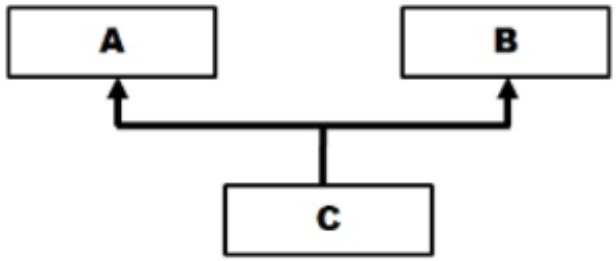
Solución: ***MRO (Method Resolution Order)*** es la estrategia utilizada por Python para encontrar un método en una jerarquía de clases.



Python: Herencia y Múltiple

Caso 1:

```
class A:
 def process(self):
  print('A process()')

class B:
 pass

class C(A, B):
 pass

obj = C()
obj.process()
print(C.mro())
#print MRO for class C
```

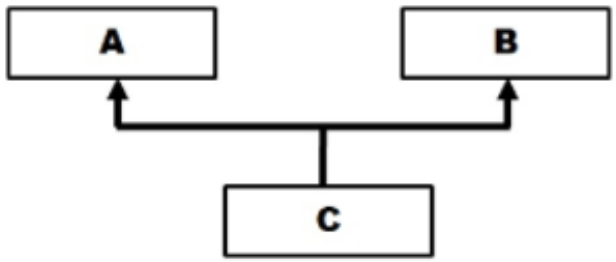


Salida: A process()
[< *class*′ ___*main*___.*C*′ >, <
class′ ___*main*___.*A*′ >, <
class′ ___*main*___.*B*′ >, <
class′ *object*′ >]

Python: Herencia y Múltiple

Caso 2:

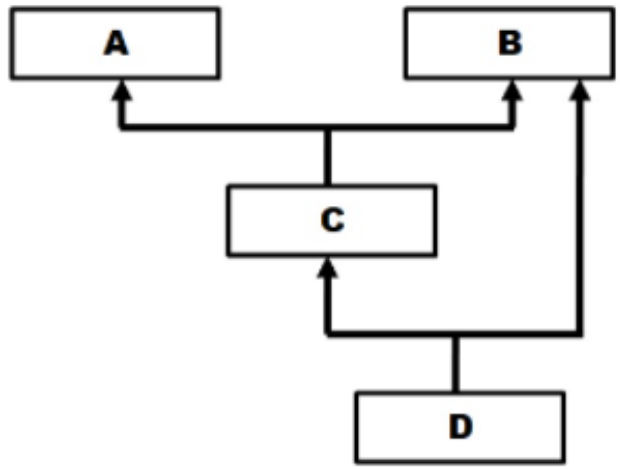
```
class A:
 def process(self):
  print('A process()')

class B:
 def process(self):
  print('B process()')

class C(A, B):
 pass

obj = C()
obj.process()
```



Salida: A process()

Caso 3:

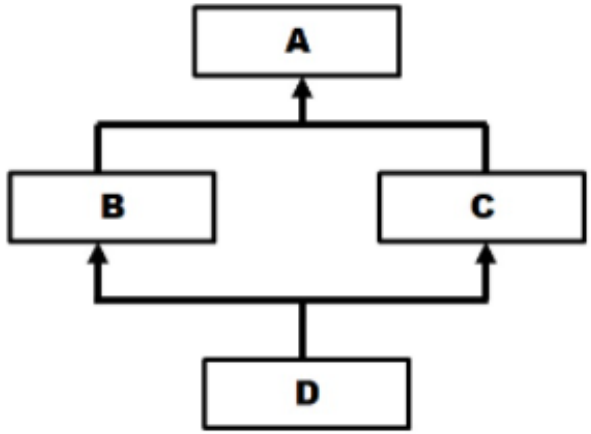
```
class A:
 def process(self):
  print('A process()')
class B:
 def process(self):
  print('B process()')
class C(A, B):
 def process(self):
  print('C process()')
class D(C,B):
 pass
obj = D()
obj.process()
print(D.mro())
```



Salida: C process()
[< *class*′ ___*main*___.*D*′ >
, < *class*′ ___*main*___.*C*′ >
, < *class*′ ___*main*___.*A*′ >
, < *class*′ ___*main*___.*B*′ >
, < *class*′ *object*′ >]

Caso 4:

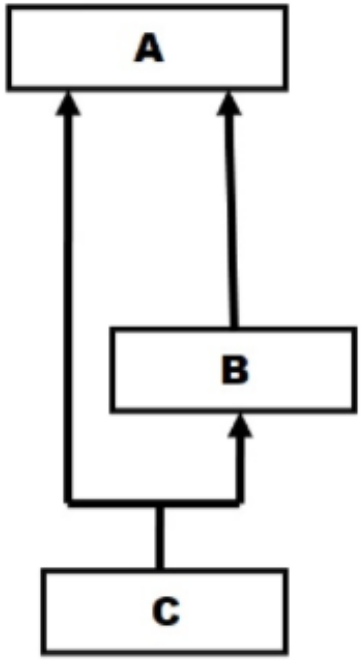
```
class A:
 def process(self):
  print('A process()')

class B(A):
 pass

class C(A):
 def process(self):
  print('C process()')

class D(B,C):
 pass

obj = D()
obj.process()
```



Salida: C process()
[< *class*′ ___*main*___.*D*′ >
, < *class*′ ___*main*___.*B*′ >
, < *class*′ ___*main*___.*C*′ >
, < *class*′ ___*main*___.*A*′ >
, < *class*′ *object*′ >]

Caso 5:

```
class A:
 def process(self):
  print('A process()')

class B(A):
 def process(self):
  print('B process()')

class C(A, B):
 pass

obj = C()
obj.process()
```

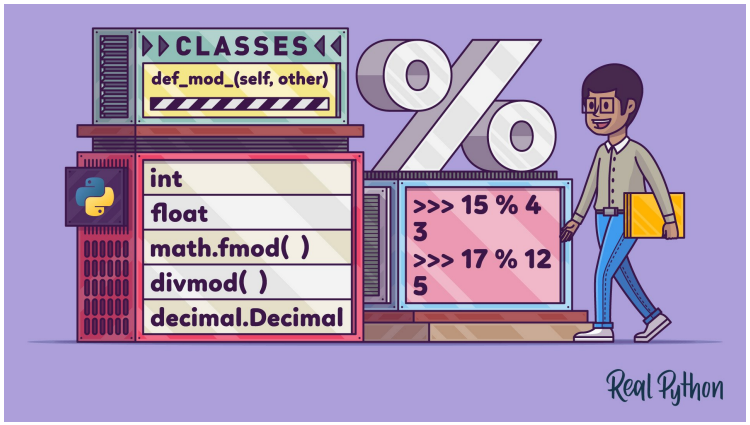


Salida: TypeError: Cannot create a consistent method resolution order (MRO) for bases A, B

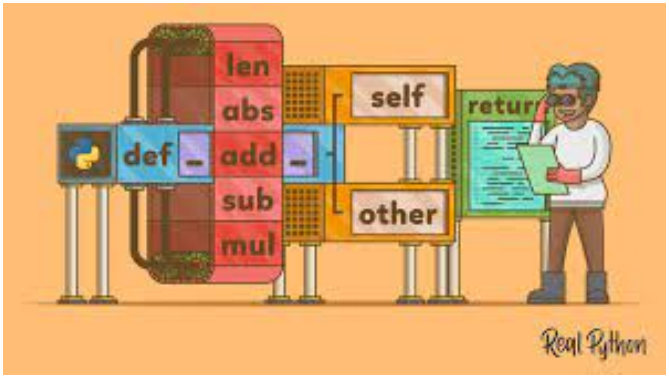
Python: Sobrecarga de Funciones

PYTHON

Python no soporta sobrecarga de funciones. Python deja la última definición de la función.
Los métodos *dunder methods* (double underscore methods) pueden ser sobrecargados.



Python: Sobrecarga de Operadores



Python permite la sobrecarga de operadores a través de la implementación de métodos especiales.

Special Method	Usage	Description
`__lt__(self, other)`	`x < y`	Returns True if x is less than y
`__le__(self, other)`	`x <= y`	Returns True if x is less than or equal to y
`__eq__(self, other)`	`x == y`	Returns True if x is equal to y
`__ne__(self, other)`	`x != y`	Returns True if x is not equal to y
`__ge__(self, other)`	`x >= y`	Returns True if x is greater than or equal to y
`__gt__(self, other)`	`x > y`	Returns True if x is greater than y

Python: Excepciones Definidas por el Usuario

- Los programadores pueden crear sus propias excepciones definiendo una clase que derive directa o indirectamente de la clase ***Exception***.
- Las clases que implementan excepciones usualmente son simples y ofrecen atributos que pueden ayudar a identificar el error.
- Es aconsejable agregar al nombre de la clase el sufijo *Error* para ser consistente con la forma de nombrar las excepciones estándar.

Python: Excepciones Definidas por el Usuario

```
class Error (Exception):
"""Clase base para excepciones en el módulo."""
 pass

class EntradaError ( Error ):
"""Excepción lanzada por errores en las
entradas.
Atributos:

expresión - expresión de entrada en la qu
ocurre mensaje -
explicación del error"""

 def __init__ (self ,expresion ,mensaje):
  self.expresion = expresion
  self.mensaje=mensaje
```

Python: Excepciones Definidas por el Usuario

```
...
try :
 raise EntradaError(1,"Este␣fue␣un\
␣␣mensaje")
except EntradaError as ee:
 print("Expresion:", ee.expresion,
       "␣Mensaje:␣",ee.mensaje)
...
```