Herencia entre Clases

Juan Álvarez Rubio
jalvarez@dcc.uchile.cl

En esta sección se presenta el concepto central de la programación orientada a objetos: la herencia. La POO surgió como el paradigma de programación que facilitaría la construcción de software al permitir reutilizar componentes ya desarrolladas y probadas. La reutilización se logra permitiendo a una clase heredar las componentes de otra clase y agregando o reemplazando métodos. La notación y la implementación en Python del concepto de herencia es particularmente sencilla y se presentará a través de ejemplos.

Herencia simple

Utilizando POO un programa que calcula el perímetro de una circunferencia se puede escribir de la siguiente manera:

```
from Circunferencia import *
#obtener valor del radio
r=float(input('radio?'))
#crear objeto de clase Circunferencia
c=Circunferencia(r)
#aplicar método para calcular perimetro
print('perímetro:',c.perimetro())
```

El programa supone que la clase Circunferencia tiene las siguientes instrucciones:

```
from  math import *
class Circunferencia:
    def __init__(self,x): #metodo constructor
      assert (type(x)==int or type(x)==float) and x>0
      self._r=x #atributo para radio
    def perimetro(self):
      return 2*pi*self._r
#prueba
C=Circunferencia(1)
assert C.perimetro()==2*pi
```

Análogamente, el siguiente programa calcula el área y el perímetro de un círculo:

```
from Circulo import *
c=Circulo(float(input("radio?")))
print('área:',c.area())
print('perímetro:',c.perimetro())
```

La solución de agregar a la clase Circunferencia un método para calcular el área en la clase no es conceptualmente adecuada puesto que los círculos y no las circunferencias tiene área. En lugar de escribir la clase Circulo completa, con los métodos constructor, área y perímetro, se puede definir la clase Circulo heredando los métodos constructor y perímetro de la clase Circunferencia y agregar el método área:

```
from Circunferencia import *
class Circulo(Circunferencia):
  def area(self):
    return pi*self._r**2
#prueba
c=Circulo(1)
assert c.area()==pi
assert c.perimetro()==2*pi
```

El atributo self._r, que fue definido en el constructor de la clase Circunferencia, es también visible en la clase Circulo puesto que su nombre comienza con uno y no dos subrayados.

En síntesis, se reutilizó la clase Circunferencia, que ya estaba escrita y probada, en la nueva clase Circulo agregando un método para calcular el área. Si bien este es un ejemplo muy sencillo, sin embargo, la idea es aplicable a clases con muchos más métodos.

Clases Pila y Cola usando herencia

Los objetos de las clases Pila (Stack) y Cola (Queue) son contenedores de información que solo difieren en la forma en que se saca un valor: la pila saca el último que ingresó (principio LIFO) y la cola saca el primero que ingresó (FIFO). Pero todos los otros métodos son similares, por lo que se pueden heredar desde una clase común:

```
#_L: list
class Contenedor:
    def __init__(self): self._L=[]
    def vaciar(self): self._L=[]
    def vacio(self): return self.__L==[]
    def poner(self,x): self._L.append(x)

class Pila(Contenedor):
    def sacar(self): return self._L.pop()
P=Pila(); P.poner(1); P.poner(2)
assert P.sacar()==2

class Cola(Contenedor):
    def sacar(self): return self._L.pop(0)
C=Cola(); C.poner(1); C.poner(2)
assert C.sacar()==1
```

Alternativamente, se puede definir la clase Cola heredando desde la clase Pila pero reemplazando el método sacar:

```
#_L: list
class Pila:
    def __init__(self): self._L=[]
    def vaciar(self): self._L=[]
    def vacio(self): return self._L==[]
    def poner(self,x): self._L.append(x)
    def sacar(self): return self._L.pop()
P=Pila(); P.poner(1); P.poner(2)
assert P.sacar()==2

class Cola(Pila):
    def sacar(self): return self._L.pop(0)
C=Cola(); C.poner(1); C.poner(2)
assert C.sacar()==1
```

Este ejemplo ilustró que al heredar se puede reemplazar o redefinir uno más métodos. También se pueden agregar atributos, como se muestra en el siguiente ejemplo en que define una clase para pilas con una altura máxima:

```
from Pila import Pila
class Pila1(Pila):
    def __init__(self,h):
        super().__init__()
        assert type(h)==int and h>0
        self._h=h
    def poner(self,x):
        if len(self._L)<self._h:
            super().poner(x)
#prueba
P=Pila1(2); P.poner(1); P.poner(2); P.poner(3)
assert P.sacar()==2
```

El constructor de la clase Pila1 reemplaza al constructor de la clase Pila. La instrucción super().__init__(self) invoca al constructor de la clase superior (Pila) y por lo tanto se define e inicializa el atributo self._L. Adicionalmente el constructor de la clase Pila1 agrega el atributo self._h para guardar la altura máxima permitida para una pila. De hecho, el atributo self._h es utilizado en la redefinición del método poner para evitar agregar valores en una pila que ya está llena.

Herencia sucesiva y herencia múltiple

La herencia puede ser sucesiva o transitiva, es decir, una clase puede heredar de una clase que a su vez hereda de otra clase. El siguiente ejemplo muestra esta posibilidad:

```
class A:
  def __init__(self,x):
    self._a=x
  def f(self): return self._a
class B(A): #hereda de A
  def __init__(self,x,y):
    A.__init__(self,x)
    self._b=y
  def g(self): return self._b
class C(B): #hereda de B
  def __init__(self,x,y,z):
    B.__init__(self,x,y)
    self._c=z
  def h(self): return self._c
c=C(1,2,3)
assert c.f()==1
assert c.g()==2
assert c.h()==3
```

En el constructor de la clase B (que hereda de A) la instrucción A.__init__(self,x) invoca al constructor de la clase A y por lo tanto es equivalente a la notación super().__init__(x). A su vez, el constructor de la clase C invoca al constructor de su clase superior B.

Finalmente, la herencia múltiple permite que una clase herede de dos o más clases:

```
class A:
  def __init__(self,x):
    self._a=x  #una variable
  def f(self): return self._a
class B:
  def __init__(self,x):
    self._b=x  #una variable
  def g(self): return self._b
class C(A,B): #C hereda de clases A y B
  def __init__(self,x,y,z):
    A.__init__(self,x)
    B.__init__(self,y)
    self._c=z
  def h(self): return self._c
c=C(1,2,3)
assert c.f()==1
assert c.g()==2
assert c.h()==3
```

En resumen, las clases se pueden relacionar en jerarquías de herencia sucesivas y múltiples.