Curso Básico de Python

9. Clases y objetos

Contenido

La *Programación Orientada a Objetos* define estructuras llamadas *objetos* que contienen a la vez datos y funciones. Los objetos se construyen en base a un modelo llamado *clase*, donde se define el tipo de información y las funciones asociadas a esos objetos. En este capítulo examinamos la forma en que Python soporta la Programación Orientada a Objetos.

Lecturas recomendadas:

1. Clases en el Tutorial de Python.

1 Espacios de nombres

En Python, una variable o una función pueden tener varios nombres, o *alias*. Así, varios nombres distintos pueden apuntar a un mismo objeto, como si fuera un puntero. También es posible colocar uno de estos nombres como argumento al invocar una función, lo que permite modificar el objeto dentro de la función recibiendo solo un puntero, el nombre de la variable.

Un *espacio de nombres* es un mapeo entre nombres y objetos. En un espacio de nombres, los nombres son exclusivos de ese espacio; Esto permite usar los mismos nombres en espacios distintos. Se denomina *alcance* (*scope*) a la región de un programa donde un espacio de nombres es directamente accesible. El alcance de una variable es la región del programa donde esa variable se puede consultar y modficar.

Espacios de nombres y alcances:

- el módulo `__main__` es un espacio de nombres; aparece al invocar el intérprete de comandos o al ejecutar un script; permanece hasta que termina la ejecución.
- los nombres definidos en un módulo son variables globales, disponibles en todo el módulo.
- las variables y funciones propias de Python (*builtin functions*) forman un espacio de nombres siempre presente.
- una función genera su propio espacio de nombres, que desaparece al finalizar su ejecución o si se levanta una excepción.
- una variable definida dentro de una función es local a esa función, está en el espacio de nombres de esa función.
- una variable declarada en el módulo se puede ver dentro de una función definida en ese módulo, pero no se puede modificar. Para poder modificar una variable del módulo dentro de una función debe declararse `global` dentro de esa función.

- la invocación recursiva de funciones genera un espacio de nombre distinto en cada invocación.

Una variable `m2_var1_m` en el módulo `alcance_2` es distinta de una variable `m2_var1_m` en otro módulo. Si desde el módulo `alcance_2` se importa la variable `m2_var1_m`, deberá hacerse con un alias o calificando la variable.
Importación con alias:

```
>>> m2_var1_m ="m2_var1_m en este módulo"
>>> from alcance_2 import m2_var1_m as var_alias
>>> print(m2_var1_m, var_alias, sep=" ; ")
m2_var1_m en este módulo ; m2_var_1_m
```

Importación calificando la variable:

```
>>> import alcance_2
>>> print(m2_var1_m, alcance_2.m2_var1_m, sep=" ; ")
m2_var1_m en este módulo ; m2_var_1_m
```

Variables definidas dentro y fuera de una función:

```
>>> var_mod ="VarMod"    # define una variable de módulo
>>> def fn():
    var_mod += "en fn"   # intenta modificar variable de módulo

>>> fn()                 # la variable de módulo es inaccessible
...
UnboundLocalError: local variable 'var_mod' referenced before assignment
>>> def fng():
   global var_mod        # declara la variable del módulo
   var_mod += " modificada en fng"

>>> fng()
>>> var_mod              # la función midificó la variable de módulo
'VarMod modificada en fng'
```

2 Clases y objetos

La *Programación Orientada a Objetos* o *POO* (*Object Oriented Programming*, *OOP*) es una forma de construir programas basada en objetos. Un *objeto* es una estructura capaz de contener información, en forma de variables, y comportamiento, en forma de funciones. Los objetos se construyen en base a un modelo llamado *clase*, equivalente a un tipo de dato. Un objeto es siempre una *instancia* de una clase, una realización particular de esa clase o tipo de dato.

Este ejemplo define una clase `Rectangulo`, y construye dos objetos, `r1` y `r2`, de tipo `Rectangulo`.

```
class Rectangulo:
    '''Clase para para definir objetos de tipo Rectangulo.
    '''
    def __init__(self, nom, a, b):
        '''Constructor, crea un objeto Rectangulo.
        '''
        self.nombre = nom    # nombre, para identificar cada rectángulo
        self.lado_a = a      # longitud de un lado
        self.lado_b = b      # longitud del otro lado
    def area(self):
        '''Calcula el área del rectángulo.
        '''
        return self.lado_a * self.lado_b
    def __str__(self):
```

```
        '''Devuelve una cadena para imprimir con "print".
        '''
        mens = "Rectangulo " + self.nombre
        mens += ", lado a: " + str(self.lado_a)
        mens += ", lado b: " + str(self.lado_b)
        mens += ", área: " + str(self.area())
        return mens
```

Observamos los siguientes elementos:

- `class` : indica el nombre de la clase; los objetos se constuirán en base a este patrón.
- `__init__(self, nom, a, b)` : método constructor, devuelve un objeto de tipo `Rectangulo`. El primer parámetro `self` es obligatorio, alude al propio objeto; recibe además un nombre para identificar el objeto, y las longitudes de los lados.
- `nombre`, `lado_a`, `lado_b` son *atritbutos de datos* o simplemente *atributos*, variables asociadas al objeto donde se retiene información relevante para este tipo de objeto.
- `self` hace referencia al propio objeto, `self.lado_a` indica "variable de nombre `lado_a` de este objeto"; puede haber otra variable del mismo nombre no propia de este objeto.
- `area(self)` es un *atributo de función* o *método*, una función propia de este tipo de objetos. Todos los métodos deben contener como primer parámetro `self`, para tener acceso a los atributos y funciones del propio objeto.
- `__str__` es un método especial (indicado por el nombre con infra guiones) que devuelve una cadena de caracteres para mostrar con el comando `print`.

Desde el intérprete de comandos podemos importar este código desde `rectangulo.py` y construir un par de rectángulos (para asegurar el acceso al módulo verificar o actualizar `sys.path`).

```
>>> from rectangulo import Rectangulo
>>> r1 = Rectangulo("R-uno", 2.4, 3.6)   # invoca __init__, objeto r1
>>> r2 = Rectangulo("R-dos", 6, 3)       # crea objeto r2
>>> print(r1); print(r2)  # muestra ambos rectángulos
Rectangulo R-uno, lado a: 2.4, lado b: 3.6, área: 8.64
Rectangulo R-dos, lado a: 6, lado b: 3, área: 18
```

Los objetos `r1` y `r2` son ambos de tipo `Rectangulo`, pero objetos diferentes, como muestran los identificadores de cada uno (los números hexadecimales 0x...):

```
>>> type(r1); type(r2)
<class 'rectangulo.Rectangulo'>
<class 'rectangulo.Rectangulo'>
>>> r1
<rectangulo.Rectangulo object at 0x7fe5dbb95490>
>>> r2
<rectangulo.Rectangulo object at 0x7fe5da9e81c0>
```

Es posible acceder y aún modificar los elementos de un objeto, usando la notación de punto `<objeto>.<atributo>`, donde el atributo puede ser una variable o una función:

```
>>> r1.nombre, r1.lado_a, r1.lado_b, r1.area()
('R-uno', 2.4, 3.6, 8.64)
>>> r1.nombre = "Mi nuevo Rectángulo"; r1.lado_a = 10; r1.lado_b = 5
>>> print(r1)
Rectangulo Mi nuevo Rectángulo, lado a: 10, lado b: 5, área: 50
```

Una clase, y por lo tanto cada objeto, definen su propio espacio de nombres. Los atributos de datos o de función (métodos) definidos en el cuerpo de una clase solo son accesibles dentro de esa clase, su alcance está limitado a la clase.

3 Variables de clase y de instancia

Un atributo de dato es una *variable de instancia*, está asociada a un objeto en particular, y otro objeto puede tener otro valor guardado en una variable del misma nombre. Una variable de instancia va precedida por `self`. Un atributo o *variable de clase* está asociada a la clase, y es compartida por todos los objetos de esa clase; el cambio en una variable de clase es visible para todos los objetos de esa clase. Una variable de clase se define dentro de la clase sin anteponer la palabra `self`; para referenciarla, se antepone el nombre de la clase, `<nombre_clase>.<variable_de_clase>`.

El siguiente código define una clase `Paloma` con dos variables de clase, `especie` y `ave_cnt`. Esta última variable es un contador de objetos, se incrementa en la función `__init__` al crear un objeto y se decrementa en la función `__del__` al destruirse ese objeto. Un objeto se destruye cuando desaparece la última referencia hacia él, y en ese momento se ejecuta la función `__del__`. Un objeto puede destruirse explícitamente con el comando `del`.

```
class Paloma:
    especie = "Ave"  # variable de clase, común a todos los objetos Paloma
    ave_cnt = 0      # variable de clase, contador de objetos existentes
    def __init__(self, p_nombre):  # constructor, crea objeto Paloma
        self.nombre = p_nombre     # variable de instancia
        Paloma.ave_cnt += 1        # incrementa contador de palomas
    def __del__(self):             # destructor, borra objeto Paloma
        Paloma.ave_cnt -= 1        # decrementa contador de palomas
    def mostrar(self):      # muestra especie, nombre y cantidad de objetos
        print(Paloma.especie, self.nombre, Paloma.ave_cnt)
```

El siguiente código crea dos objetos y muestra el total de objetos creados.

```
>>> from paloma import Paloma
>>> plm_1 = Paloma("Gertrudis")
>>> plm_2 = Paloma("Milena")
>>> plm_1.mostrar()               # total 2 objetos
Ave Gertrudis 2
>>> plm_2.mostrar()
Ave Milena 2
```

Si se cambia una variable de clase, todos los objetos de esa clase ven el mismo valor:

```
>>> Paloma.especie = "Paloma de Monte"  # cambia variable de clase
>>> plm_1.mostrar()                    # cambia en todos los objetos
Paloma de Monte Gertrudis 2
>>> plm_2.mostrar()                    # cambia en todos los objetos
Paloma de Monte Milena 2
```

Al borrar un objeto, se actualiza el contador de objetos existentes:

```
>>> del plm_1                          # borra un objeto de 2
>>> plm_2.mostrar()                    # ... queda 1
Paloma de Monte Milena 1
```

4 Herencia

En la Programación Orientada a Objetos, la *herencia* es un mecanismo que permite definir una clase en base a otra preexistente, de la cual recibe todos sus atributos y métodos. Una clase Perro puede heredar de una clase Mamífero, porque un perro es un mamífero, tiene todas sus características. También un gato es un mamífero, por lo que una clase Gato también heredaría de Mamifero. Así, es posible reunir en Mamifero el comportamiento común de todos los mamíferos, y agregar en cada mamífero específico (perro, gato, humano) la información de estado (atributos) y comportamiento (métodos) propios de cada especie. Diríamos entonces:

- Mamifero es una clase superior, o *superclase*, de la cual heredan Perro y Gato.
- Perro y Gato son clases descendientes o *subclases* de Mamífero.
- Perro y Gato tienen todos los atributos y métodos de Mamífero, además de los suyos propios que puedan definirse.
- La herencia es una relación "es-un-tipo-de": un Perro es un tipo de Mamífero.

La herencia permite construir una jerarquía de clases: SanBernardo, Chihuahua y GranDanes serían subclases de Perro; SanBernardo tendría, además de los suyos propios, todos los atributos de Perro, y también todos los de Mamifero, heredados a través de Perro.
En el siguiente código se define una clase `Figura`, y dos subclases, `Triangulo` y `Rectangulo`. Cada figura tiene un identificador, y se cuenta la cantidad de figuras creadas, sean triángulos o rectángulos.

```
class Figura():                     # clase superior
    nom_fig = "Figura"              # variable de clase Figura
    fig_cnt = 0                     # variable de clase, cuenta instancias
    def __init__(self, id_fig):     # constructor de Figura
        self.id_fig = id_fig        # identificador, atributo de instancia
        Figura.fig_cnt += 1         # incrementa variable de clase
        print("   [ejecuta constructor en Figura]")
    def __del__(self):              # destructor de Figura
        Figura.fig_cnt -= 1         # decrementa variable de clase
        print("   [ejecuta destructor en Figura]")
    def area():                     # método a sobreescribir en subclase
        return 0
    def __str__(self):              # devuelve cadena para imprimir Figura
        return "Figura " + self.id_fig + ", cantidad: " +
str(Figura.fig_cnt)

class Triangulo(Figura):              # subclase de Figura
    nom_fig = "Triángulo"             # variable de clase Triangulo
    def __init__(self, id_fig, b, h): # constructor de Triangulo
        super().__init__(id_fig)      # constructor de clase superior
        self.base = b
        self.altura = h
        print("     [ejecuta constructor en Triangulo]")
    def __del__(self):                # destructor de Triangulo
        super().__del__()             # destructor de clase superior Figura
        print("     [ejecuta destructor en Triangulo]")
    def area(self):                   # sobreescribe método de Figura
        return self.base * self.altura / 2
    def __str__(self):                # cadena para imprimir Triangulo
        mens = super().__str__()      # invoca método de la clase superior
        mens += " - " + self.nom_fig + ", área = " + str(self.area())
        return mens                   # devuelve cadena de caracteres

class Rectangulo(Figura):             # subclase de Figura
    nom_fig = "Rectángulo"            # variable de clase Rectangulo
    def __init__(self, id_fig, a, b): # constructor de Rectangulo
        super().__init__(id_fig)      # constructor de clase superior
        self.lado_a = a
        self.lado_b = b
        print("     [ejecuta constructor en Rectangulo]")
    def __del__(self):                # destructor de Rectangulo
        super().__del__()             # destructor de clase superior Figura
        print("     [ejecuta destructor en Rectangulo]")
    def area(self):                   # sobreescribe método de Figura
        return self.lado_a * self.lado_b
    def __str__(self):                # cadena para imprimir Rectangulo
        mens = super().__str__()      # invoca método de la clase superior
        mens += " - " + self.nom_fig + ", área = " + str(self.area())
        return mens                   # devuelve cadena de caracteres
```

Observamos los siguientes elementos:

- `class Rectangulo(Figura)` : la clase `Rectangulo` hereda de `Figura`; se coloca entre paréntesis la superclase de la cual hereda esta clase.
- `nom_fig` es una variable de clase propia de cada clase, se sobreescribe en las subclases.
- `fig_cnt` es una variable de la clase `Figura`, se actualiza en las subclases con la notación `Figura.fig_cnt`.
- las subclases sobreescriben la función `area()`, cada una según su geometría.
- las subclases pueden invocar métodos de la superclase usando la notación `super()`. Por ejemplo `super().__init__()` invoca el constructor de la superclase. Esta invocación va siempre al principio del método.

En el intérprete de comandos:

```
>>> from figura import Figura, Triangulo, Rectangulo
>>> r1 = Rectangulo("RR-1", 1, 2)
>>> r2 = Rectangulo("RR-2", 5, 4)
>>> t1 = Triangulo("TR-1", 5, 4)
>>> for fig in [r1, r2, t1]:
...     print(fig)

Figura RR-1, cantidad: 3 - Rectángulo, área = 2
Figura RR-2, cantidad: 3 - Rectángulo, área = 20
Figura TR-1, cantidad: 3 - Triángulo, área = 10.0
```

Mostrar la cantidad de figuras, borrar una, verificar actualización de cantidad de figuras:

```
>>> Figura.fig_cnt
3
>>> del r1
>>> Figura.fig_cnt
2
```

Verificar la clase a que corresponden los objetos (instancias de qué clase son):

```
>>> isinstance(r2, Rectangulo)         # objeto en r1 es un Rectangulo
True
>>> isinstance(t1, Triangulo)          # objeto en t1 es un Triangulo
True
>>> isinstance(t1, Figura)             # objeto en t1 es también una Figura
True
>>> issubclass(Rectangulo, Figura)     # Rectangulo hereda de Figura
True
```

La ejecución del módulo figura,

```
$ python3 figura.py
```

muestra mensajes de creación y destrucción de objetos (omitidos en la transcripción anterior).

5 Herencia múltiple

Python soporta *herencia múltiple*: una clase puede heredar de varias clases a la vez. La sintaxis es:

```
class ClaseDerivada ( Superclase1, Superclase2, ...):
```

Se heredan primero los atributos de la primera superclase, heredando luego ordenadamente de los ancestros en forma ascendente; se sigue igual con la segunda superclase ascendiendo toda su jerarquía, y así hasta tratar todas las superclases. La herencia múltiple requiere un diseño cuidadoso de la jerarquía de clases, para evitar posibles ambigüedades. Más información en el Tutorial de Python, Herencia Múltiple.

6 Ejercicios

1. En el intérprete de comandos, importar el módulo `rectangulo`. Crear algunos rectángulos y mostrarlos; modificar alguno de sus atributos, mostrarlos.

2. Estudiar y ejecutar el módulo `paloma.py`, un ejemplo simple de variables de clase y de instancia.

3. Estudiar y ejecutar el módulo `figura.py`. Verificar la herencia, observando los mensajes de construcción y destrucción de objetos. En el análisis del código, prestar especial atención a:
   1) la invocación de métodos de la superclase, con `super()`.
   2) la sobreescritura de métodos en la subclase, e.g. `area()` y `__str__()`.
   3) la presencia y el código de constructores y destructores.
   4) el uso de las variables de clase, `fig_cnt` y `nom_fig`, así como su sobre escritura.

4. Estudiar y ejecutar el módulo `alcance_1.py`, que usa el módulo `alcance_2.py`. La ejecución de `alcance_1.py` muestra la accesibilidad de variables locales, de módulo y globales. Visualizar el código y la ejecución en dos ventanas distintas, para poder interpretar más fácilmente la ejecución. Si se usa IDLE, abrir el archivo `alcance_1.py` y ejecutarlo con F5, o Run / Run module; el resultado de la ejecución se visualiza en otra ventana.

5. Herencia múltiple. Estudiar y ejecutar el módulo `chofer.py`, un ejemplo muy simple.