Herramientas Computacionales para la Astroinformática

Cristian A. Vega-Martínez (oficina: IMIP, Académicos 2)
Facundo A. Gómez

Introducción a la Programación Orientada a Objetos

usando Python

Programación Orientada a Objetos

- La programación orientada a objetos (POO) es un paradigma de programación que utiliza **objetos** y **clases** para organizar el código de manera estructurada y modular.
- Python es un lenguaje versátil que soporta tanto programación procedimental como orientada a objetos. No obstante, **los objetos cumplen un rol central** en su estructura y forma de programación.

¿Qué tan versátil y orientado a objetos es Python?

Analizar objetos en ejemplos Clases/2.5/ejemp:
- ex_calc.py y ex_mult.py (prints básicos)
- ex_histogram.py (trabajo de datos)
- ex_ball.py ???
- ex_slider.py ???

Objetos, *scopes* y *namespaces*

Un **namespace** en Python es una lista de referencias asociadas a objetos. Ejemplos:

- el conjunto de nombres built-in (`len`, `print`, `ValueError`, etc),
- nombres globales de un módulo
- nombres locales: al invocar funciones o instanciar clases.

- Los namespaces son independientes entre sí: pueden tener referencias del mismo nombre asociadas a objetos distintos (e.g. `sum != np.sum`)
- Se crean en diferentes momentos y se olvidan siguiendo el flujo del código.

Entonces: un **scope** (ámbito) es una región textual de un programa en Python donde un namespace es accesible directamente.

Las instrucciones ejecutadas en el **scope superior** del intérprete (script o interactivo), se consideran parte del **módulo __main__** , el cual define su propio espacio de nombres global.

Declaraciones en diferentes scopes / namespaces

- Por defecto, toda declaración crea una referencia en el namespace más anidado (local).
- Este comportamiento se puede cambiar usando **`global`** y/o **`nonlocal`**.

```
def scope_test():
      def do_local():
      spam = "local spam"

      def do_nonlocal():
      nonlocal spam
      spam = "nonlocal spam"

      def do_global():
      global spam
      spam = "global spam"

      spam = "test spam"
      do_local()
      print("After local assignment:", spam)
      do_nonlocal()
      print("After nonlocal assignment:", spam)
      do_global()
      print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Clases

- Proveen una forma de empaquetar datos y funcionalidad.
- Cada clase nueva define un tipo de objeto nuevo, permitiendo crear nuevas instancias de ese tipo.
- Cada instancia de puede tener atributos adjuntos para mantener su estado. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

  Sintaxis:

```
# definición
class ClassName:
      <instrucciones>
      …

# Instanciación:
x = ClassName( )
```

Ejemplo:

```
class MyClass:
    """A bad example class"""
    i = 12345   # don't do this

    def f(self):
        return 'hello world'
```

Esta definición creará referencias `MyClass.i` y `MyClass.f` para acceder al atributo o método respectivo.

Clases: instancias

Para instanciar una clase se utiliza la notación de función con el nombre de la clase. Por ejemplo:

```
x = MyClass( )
```

Crea una nueva instancia de la clase `MyClass`, y asigna este objeto a la referencia `x`.

La operación instanciación (*llamado* de un objeto *class*) suele crear objetos **vacíos**.
Para que las instancias tengan atributos declarados en un estado inicial, se utiliza el método especial `__init__(...)`, de la forma:

```
def __init__(self):
    self.i = 12345
    self.data = []
    ...
```

```
#Ejemplo:

class Complex:
    """A better-defined class"""

    def __init__(self, realpart, \
                 imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
print(x.r, x.i)
```

En los lenguajes con POO, este método se suele llamar **constructor**, de modo que el **destructor** es aquel con las instrucciones de eliminación de un objeto de la clase.

Clases: atributos (de datos)

Un atributo de datos de una clase corresponde a una "variable" miembro de la clase (como `r` e `i` del ejemplo `Complex`).

Como toda "variable" de Python, un atributo puede ser definido **en cualquier momento** (incluso luego de creado el objeto).

Existen dos tipos de atributos (de datos) en una clase:

- **Atributos de instancia:** todos aquellos que son creados **para cada instancia** de la clase de forma independiente.
  Estos se definen y usan mediante la referencia `self`.
- **Atributos de clase:** son aquellos que son compartidos por todas las instancias de la clase. No es recomendable utilizar objetos mutables en esta categoría.

  Si la misma referencia (nombre) es usado en ambos tipos, se **priorizará el de instancia**.

```
# Ejemplo:
class Dog:

    familia = 'canine'    # atributo de clase

    def __init__(self, name):
        self.name = name  # atributo de instancia

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.familia       # compartido por todo Dog
'canine'
>>> e.familia       # compartido por todo Dog
'canine'
>>> d.name          # unico de d
'Fido'
>>> e.name          # unico de e
'Buddy'
```

En muchos lenguajes con POO los atributos de las clases pueden definirse *públicos* o *privados* (no visibles desde otros scopes).
En Python todo es **público por defecto**, aunque las referencias se pueden ocultar parcialmente usando guiones inferiores _.

Clases: métodos

Un método es una **función** que **pertenece** a un objeto (de clase u otros objetos del lenguaje).

Se definen dentro de la clase como una función, pero utilizando `self` como primer argumento.

Se llaman al igual que una función, pero accediendo desde una instancia de la clase mediante el punto. Por ejemplo:

```
x.f( )  # con x instancia de MyClass
```

Como las funciones (y métodos) también son objetos en Python, sus referencias pertenecen al namespace del objeto al igual que los atributos de datos.

> Los **métodos** también pueden ser considerados **atributos**.

```
#Ejemplo:
class Dog:

    def __init__(self, name):
        self.name = name
        # lista vacia nueva para cada Dog:
        self.tricks = []

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```