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 puede tener atributos adjuntos para definir su estado. Las clases también pueden tener métodos que permiten modificar el estado de las instancias.

  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 ser instanciado 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 (de instancia)

Un método es una **función** que **pertenece** a un objeto (instancia 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']
```

Clases: otros métodos

Con usabilidad más específica, en una clase también se pueden definir:

- **Métodos de clase**
  Aquellos que únicamente operan sobre los atributos de clase, no de instancia.
  Se definen anteponiendo `@classmethod`, y su primer parámetro es `cls` en vez de `self`.

- **Métodos estáticos**
  Aquellos que no operan sobre atributos de clase ni de instancia. Son esencialmente funciones dentro de clases.
  Se definen anteponiendo `@staticmethod` y no incluyen argumentos `self` ni `cls`.

```python
class Estudiante:
    # Atributo de clase:
    universidad = "Universidad de La República"

    def __init__(self, nombre, edad):
        # Atributos de instancia:
        self.nombre = nombre
        self.edad = edad

    # Método de instancia
    def presentarse(self):
        return f"Soy {self.nombre}, “+
          “tengo {self.edad} años y “+
          “estudio en {Estudiante.universidad}"

    @classmethod
    def cambiar_universidad(cls, nueva_univ):
        cls.universidad = nueva_univ

    @staticmethod
    def es_mayor_de_edad(edad):
        return edad >= 18
```

Clases: otros métodos

Getters y Setters

En lenguajes POO que utilizan **atributos privados** generalmente se insta a definir métodos específicos para:
-consultar (getter: `get_attname( )`) y
-modificar (setter: `set_attname( )`)
atributos internos de los objetos.

Como en Python es todo público, estos no son necesarios. Igualmente, incluye **built-ins**:

- `getattr(objeto, attname)`
  `object.__getattribute__( )`
- `setattr(objeto, attname, value)`
  `object.__setattr__( )`

Métodos especiales

Así como el constructor `__init__( )`, todas las clases incluyen métodos especiales que están asociados a acciones de Python y pueden ser definidos según necesidad. Por ejemplo:

`__del__( )`: destructor del objeto.

`__str__( )`: se invoca cuando se llama a `print( )` o `str( )` con el objeto.

`__repr__( )`: representación “oficial” del objeto, análoga a la sintaxis de creación.

`__len__( )`: se invoca cuando se llama `len( )` con el objeto.

y muchos más!

Clases: más métodos especiales

Entre los métodos especiales también se incluyen los que nos permiten definir el comportamiento de un objeto con un operador (e.g: +, -, /, *, **, [ ], etc).

Por ejemplo, para habilitar la suma (+) de objetos de nuestra clase Complex, se puede agregar a la clase:

```
def __add__(self, x):
    return Complex(self.r+x.r, self.i+x.i)
```

Una vez definida, las siguientes operaciones pasan a ser equivalentes:

```
a + b      # a,b del mismo tipo
a.__add__(b)
type(a).__add__(a, b)
```

El proceso de redefinir una instrucción ya conocida por el lenguaje (método u operador) se denomina **sobrecarga**.

Operator	Method	Expression
+ Addition	`__add__(self, other)`	`a1 + a2`
- Subtraction	`__sub__(self, other)`	`a1 - a2`
* Multiplication	`__mul__(self, other)`	`a1 * a2`
@ Matrix Multiplication	`__matmul__(self, other)`	`a1 @ a2` (*Python 3.5*)
/ Division	`__div__(self, other)`	`a1 / a2` (*Python 2 only*)
/ Division	`__truediv__(self, other)`	`a1 / a2` (*Python 3*)
// Floor Division	`__floordiv__(self, other)`	`a1 // a2`
% Modulo/Remainder	`__mod__(self, other)`	`a1 % a2`
** Power	`__pow__(self, other[, modulo])`	`a1 ** a2`
<< Bitwise Left Shift	`__lshift__(self, other)`	`a1 << a2`
>> Bitwise Right Shift	`__rshift__(self, other)`	`a1 >> a2`
& Bitwise AND	`__and__(self, other)`	`a1 & a2`
^ Bitwise XOR	`__xor__(self, other)`	`a1 ^ a2`
| (Bitwise OR)	`__or__(self, other)`	`a1 | a2`
- Negation (Arithmetic)	`__neg__(self)`	`-a1`
+ Positive	`__pos__(self)`	`+a1`
~ Bitwise NOT	`__invert__(self)`	`~a1`
< Less than	`__lt__(self, other)`	`a1 < a2`
<= Less than or Equal to	`__le__(self, other)`	`a1 <= a2`
== Equal to	`__eq__(self, other)`	`a1 == a2`
!= Not Equal to	`__ne__(self, other)`	`a1 != a2`
> Greater than	`__gt__(self, other)`	`a1 > a2`
>= Greater than or Equal to	`__ge__(self, other)`	`a1 >= a2`
[index] Index operator	`__getitem__(self, index)`	`a1[index]`
in In operator	`__contains__(self, other)`	`a2 in a1`
(*args, ...) Calling	`__call__(self, *args, **kwargs)`	`a1(*args, **kwargs)`

Clases *de datos*

A veces es conveniente definir un tipo de clase más simple que sólo sirva de empaquetador de datos de tipos conocidos con sus nombres (al estilo **`struct`** de C). En Python se incluyen las dataclasses para este propósito:

```
from dataclasses import dataclass

@dataclass
class Estudiante:
    nombre: str
    RUT: str
    edad: int
    nivel: int

>>> est = Estudiante('Jack Torrance', '12.345.678-9', 25, 4)
>>> est.RUT
'12.345.678-9'
```

Herencia

La herencia es el proceso por la cual se crea una clase nueva (clase derivada, child o subclase) **a partir de otra** (clase base, parent o superclase), de modo que la nueva **hereda** atributos y métodos definidos en la original.

- La herencia establece una jerarquía entre las clases.
- La clase derivada puede tener atributos y métodos adicionales o puede **sobrescribir** los heredados de la clase base.
- Si se desea extender métodos de la clase base: es posible llamar a cualquier método o atributo de la clase base mediante:
  `BaseClassName.method(self, args)`
- La herencia establece una relación del tipo "es un" entre la clase base y la clase derivada, lo cual se puede consultar con las built-in:
  `isinstance(obj, type)`
  `issubclass(type, type)`

```python
### Ejemplo de herencia ###

# Clase base
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hablar(self):
        pass

# Clase derivada
class Perro(Animal):
    def hablar(self):
        return "Guau!"

# Clase derivada
class Gato(Animal):
    def hablar(self):
        return "Miau!"

# Creación de objetos
perro = Perro("Rex")
gato = Gato("Luna")

print(perro.hablar())  # Salida: Guau!
print(gato.hablar())   # Salida: Miau!
```

Tener métodos distintos pero con mismo nombre en diferentes objetos se conoce como **polimorfismo**.

Tópico de profundización autónoma.

Hasta el momento hemos visto una colección de conceptos complejos nuevos:

- objeto
- referencia
- instancia
- clase
- atributo (de dato)
- método
- operador
- namespace
- polimorfismo
- herencia

Actividad:

1. Crear un **esquema** tipo **mapa conceptual** que permita representar gráficamente (la mayor cantidad de) estos conceptos. Presentar este esquema en clases.

2. Crear un **diccionario de términos** que incluya estos y otros que Uds. consideren pertinentes.

Agregar estos a su repositorio personal del curso.

Herencia: la función super( )

La función super( ) permite a la clase derivada llamar a métodos de la clase base. Es especialmente útil cuando se desea **extender** o **modificar** el comportamiento de un método heredado sin reescribirlo por completo.

El uso de esta función nos permite evitar el uso del nombre de la clase base, lo cual soluciona problemas asociados con la resolución de nombres y facilita modificaciones del código (e.g. cambio de nombre de la clase base).

Ejemplos de uso:

```
# Uso de super() en el constructor:

class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

class Estudiante(Persona):
    def __init__(self, nombre, edad, nivel):
    #  constructor clase base:
        super().__init__(nombre, edad)
      self.nivel = nivel

>>> e = Estudiante("Ana", 25, 7)
>>> print(e.nombre, e.edad, e.nivel)
"Ana 25 7"
```

```
# Uso de super() en un método de la clase:

class Rectangulo:
    def area(self, largo, ancho):
        return largo * ancho

class Cuadrado(Rectangulo):
    def area(self, lado):
        # usamos método de la clase base:
        return super().area(lado, lado)

>>> c = Cuadrado()
>>> print(c.area(4))
16
```

Encapsulamiento

Encapsular es la práctica de *restringir* el acceso a componentes de un objeto, lo cual permite ocultar los detalles de implementación y proteger su integridad.
Python define todas las componentes de un objeto como públicas, pero sigue una convención de nombres para emular encapsulamiento:

1) **Atributos Protegidos:**
   Se usa para sugerir que el atributo (de datos o método) no debería ser accedido directamente desde fuera de la clase, pero si puede ser utilizado por las subclases. Se usa para definir atributos de manejo internos de la clase, que podrían modificarse en alguna versión futura.
   Se indican con un guión bajo previo (_).
   Por ejemplo: `_atr_protegido`

2) **Atributos Privados:**
   Se utilizan para definir atributos que no deberían ser accedidos desde fuera de la clase. Python automáticamente **altera el nombre** (*name mangling*) de estos atributos para resaltar esta idea.
   Se indican con dos guiones bajos previos (__).
   Por ejemplo: `__atr_privado`

```python
# Ejemplos de encapsulamiento:
class MiClase:
    def __init__(self):
        self.__privado = "atprivado"

    # Accedemos al atributo con su nombre
    # original dentro de la clase:
    def mostrar(self):
          return self.__privado

obj = Ejemplo()
print(obj.mostrar())  # Esto funciona


# Esto generaría un error:
# print(obj.__privado)

# Accedemos al atributo usando
# su nombre "mangled":
print(obj._Ejemplo__privado)
```

El ***name mangling*** agrega el nombre de la clase a la referencia del atributo, de modo que su acceso desde instancias cambia.

Instancias de subclases usarán el nombre de la clases donde se definió el atributo privado.

Polimorfismo

Alude a la capacidad de los objetos de tomar muchas formas: en Python, un objeto de una clase puede ser tratado como de otra clase relacionada.

Permite que una **función** o **método** se aplique a **objetos de diferentes clases**, siempre que esas clases compartan una interfaz común (por ejemplo, un método con el mismo nombre).
Reconocemos dos tipos de polimorfismo:

1) **Polimorfismo con Funciones y Objetos**: Capacidad de una función de aceptar objetos de diferentes clases y realizar operaciones, siempre que esas clases implementen un método específico.

2) **Polimorfismo con Métodos de Clase**: Las clases derivadas pueden tener sus propias implementaciones de métodos de la clase base.

El polimorfismo permite la reutilización de código y reduce la necesidad de **comprobaciones de tipo** en el código. Facilita la extensibilidad y modularidad del código.

Polimorfismo: ejemplos

```python
# con funciones y objetos:
class Gato:
    def sonido(self):
        return "Miau!"

class Perro:
    def sonido(self):
        return "Guau!"

def hacer_sonido(animal):
    return animal.sonido()

mi_gato = Gato()
mi_perro = Perro()

print(hacer_sonido(mi_gato))
print(hacer_sonido(mi_perro))

### Salida:
# Miau!
# Guau!
```

```python
class Animal:
    def sonido(self):
        return "carf carf..."

class Gato(Animal):
    def sonido(self):
        return "Miau!"

class Perro(Animal):
    def sonido(self):
        return "Guau!"

animales = [Gato(), Perro(), Animal()]

for animal in animales:
    print(animal.sonido())

### Salida:
# Miau!
# Guau!
# carf carf...
```

Herencia múltiple

En Python una subclase puede heredar los atributos y métodos de más de una clase base. Esto permite crear clases más versátiles y flexibles, pero no es una característica común en la POO.

Sintaxis:

```
class DerivedClassName(Base1, Base2, …):
    <instrucciones>
    …
```

Esta flexibilidad agrega mayor complejidad a los códigos, de modo que se debe utilizar con cautela..

Method Resolution Order (MRO)

Es el orden seguido por Python para buscar una referencia a un método dentro de los namespaces definidos por la jerarquía de clases heredadas.

Se puede consultar con la función `mro( )` que se **incorpora automáticamente** a las definiciones de las (sub)clases.

```
# ejemplo llamado a mro():
class A:
    pass
>>> A.mro( )
```

La función `super()` sigue el MRO para definir a qué clase invocar.

Herencia múltiple: potenciales problemas

El uso de herencia múltiple tiene problemas potenciales que se deben considerar:

- **Conflicto de nombres**
  En el caso que dos clases base tenga algún atributo (de datos o método) que tienen el mismo nombre.
- **Herencia en diamante**
  Ocurre cuando una clase hereda de dos clases que tienen una clase base común.

Implementar clases con herencia múltiple demanda conocer el MRO y ser cauteloso con el orden que define la herencia.

```python
# Ejemplo herencia en diamante:
class A:
    def mostrar(self):
        print("Método de clase A")

class B(A):
    def mostrar(self):
        print("Método de clase B")

class C(A):
    def mostrar(self):
        print("Método de clase C")

# Clase derivada con herencia múltiple
class D(B, C):
    pass

obj = D()
obj.mostrar()  # ¿Qué método se invocará?

print(D.mro())  # Muestra el MRO
```

Objetos compuestos (composición)

Se refiere a la construcción de clases complejas utilizando otras **clases como componentes** (atributos), en lugar de utilizar herencia.

Se utiliza cuando queremos representar una relación del tipo "***tiene***" o "***usa***" entre clases. Por ejemplo, un "Auto" tiene un "Motor".

Esta forma de implementación es más flexible y modular que la herencia, pero requiere mayor planificación y diseño de código.

```python
class Motor:
    def arrancar(self):
        return "El motor arranca"

class Auto:
    def __init__(self):
        self.motor = Motor()

    def arrancar(self):
        return self.motor.arrancar()
```

Los 4 pilares de la POO

Programación Orientada a Objetos

- Abstracción (clases, objetos)
- Encapsulamiento
- Herencia
- Polimorfismo

Tópico de profundización autónoma:

Estudiar el concepto de **Iterador** (y **Generador**) en Python.
¿Qué son?
¿Para qué/Cuándo se usan?
¿Cómo se funcionan en las clases?