

Programación II

Programación Orientada a Objetos

Universidad Nacional de Rosario.
Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Paradigmas de programación

- Los paradigmas de programación son diferentes estilos de construir programas.
- Al momento de diseñar un lenguaje de programación se incluyen en él construcciones que faciliten un estilo u otro de programación.
- Luego, en base a esas construcciones los programas siguen uno u otro estilo de programación.

Paradigma Imperativo

- En el paradigma imperativo, los programas se construyen como una serie de órdenes o instrucciones concretas.
- Como programadores, en un estilo imperativo lo que hacemos es pensar paso a paso que es lo que queremos que nuestro programa haga.
- La computación se lleva a cabo mediante el concepto de **estado**, que es la colección de variables y sus valores actuales en un momento dado. Las instrucciones que escriben describen como alterar ese estado.
- Algunos lenguajes imperativos son: **FORTRAN**, **C** o **C++**.

Paradigma Orientado a Objetos

- La programación orientada a objetos es una evolución del paradigma imperativo.
- En este paradigma, los programadores pensamos los programas como una serie de objetos que interactúan unos con otros.
- La computación también se lleva a cabo mediante el **estado**, pero la diferencia es que, además del estado global, cada objeto tiene un estado propio.
- Algunos lenguajes orientados a objetos son: **Java**, **C Sharp** o **JavaScript**.

... y Python?

Puede ser imperativo...

```
my_list = [1, 2, 3, 4, 5]
sum = 0

for x in my_list:
    square = x * x
    doble = 2 * square
    sum += doble

print(sum)
```

Puede ser orientado a objetos...

```
class MyList():
    def __init__(self, any_list):
        self.any_list = any_list

    def square(self):
        return [ x * x for x in self.any_list ]

    def doble(self):
        return [ 2 * x for x in self.square() ]

    def sum(self):
        return sum(self.doble())

my_list = MyList([1, 2, 3, 4, 5])

print(my_list.sum())
```

- Python es **multiparadigma**, combina propiedades de distintos lenguajes de programación.
- Lo podemos usar con distintos estilos y usar distintos estilos según sea lo mas apropiado para cada problema.
- Incluso podemos mezclar estilos dentro del mismo programa, pero debemos intentar siempre que el resultado sea un programa claro, limpio y legible.

- Una idea de un objeto es representar una entidad de la vida real.
- La otra idea es que el objeto agrupe todas las variables y funciones asociadas con la misma entidad real.

Cosas de lo más cotidianas pueden pensarse como un objeto, desde un gato o un auto.

- Cada uno de estos objetos tiene ciertas características. Por ejemplo, para el caso de un gato podemos decir cual es su tamaño, su color de pelo o su mestizaje. A estas características las llamamos **atributos**.
- Por otro lado, cada objeto tiene una serie de comportamientos que lo distingue, por ejemplo en el caso de perro podrían ser caminar o maullar. A estos comportamientos o funcionalidad propia de cada objeto los llamaremos **métodos**.

Un **objeto** es un ente que consta de *identidad, estado y de un comportamiento*.

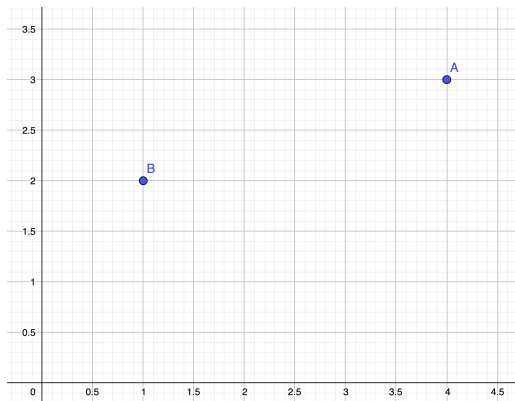
Una **clase** es una *plantilla para la creación de nuevos objetos*.

- Las clases nos permiten modularizar los datos y la funcionalidad asociada a un tipo de objeto.
- Al definir una clase debemos definir qué atributos y qué métodos contendrán los objetos pertenecientes a esta clase.
- Cuando un objeto pertenezca a una clase, diremos que ese objeto es una **instancia** de la clase.

Hemos trabajado hasta ahora con clases y objetos sin saberlo, pues en Python los tipos de datos integrados (como por ejemplo `int`, `list`, etc.) están implementado como clases y sus instancias concretas son objetos.

Lo único que nos falta aprender es a crear tipos definidos por nosotros.

Representando Entidades



Definición de clases

```
class Point:
    """representación de un punto
    en un plano cartesiano 2D"""

    def __init__(
        self, x: float, y: float
    ) -> None:
        self.x = x
        self.y = y

    def __str__(self) -> str:
        return (
            "(" + str(self.x)
            + ", "
            + str(self.y) + ")"
        )
```

Instanciando objetos

Podemos crear objetos pertenecientes a la clase del siguiente modo

```
A = Point(3, 4)
B = Point(1, 2)
```

Al realizar la llamada `Point(3, 4)`:

- 1 Se crea una nueva instancia de la clase `Point`.
- 2 Se ejecuta el constructor `__init__`, con `self` \rightarrow la instancia nueva, `x` \rightarrow 3, `y` \rightarrow 4.
- 3 El constructor asigna los atributos `self.x` \rightarrow 3, `self.y` \rightarrow 4.
- 4 Cuando finaliza la ejecución del constructor, se asigna `mi_punto` \rightarrow la instancia de `Point` recién creada.

Instanciando objetos

El siguiente método, `__str__` devuelve una representación como *string* de un objeto de tipo `Point`. Si la clase provee este método, **sobrecarga** el comportamiento por defecto de la función *built-in* `str`

```
>>> str(A)
'(3, 4)'
```


Podemos cambiar los valores de los atributos de una instancia utilizando la notación punto:

```
>>> A.x = 3.4
>>> A.y = 4.4
>>> A
(3.4, 4.4)
```

x e y se llaman **atributos** del objeto y decimos que el atributo x del objeto referenciado en la variable A posee el valor 3,4. Cada atributo es un número de punto flotante. Podemos leer el valor de cada atributo del mismo modo:

```
>>> A.x
3.4
```

Atributos de clase y Atributos de instancia

En Python se distingue entre dos tipos de atributos:

- Los **atributos de clase** (o variables de clase) son compartidos por todas las instancias de la clase.
- Los **atributos de instancia** (o variables de instancia) son particulares para cada objeto creado con esa clase.

Por ejemplo, en la clase `Point`, ambos atributos son de instancia, dado que representan coordenadas que varían según el punto que estamos representando.

Normalmente, debe evitarse el uso de atributos de clase, excepto para definir constantes.

El significado de las palabras “igual”, “mismo”, parece perfectamente claro pero contiene sutilezas propias del lenguaje natural.

Por ejemplo, cuando alguien dice “Cristian tiene un auto *igual* al mío”, seguramente signifique que ambos autos son del mismo fabricante y modelo, pero que son dos autos diferentes.

Sin embargo, si alguien dice “Cristian y yo tenemos el *mismo* profesor” seguramente se refieran ambos a la misma persona, no a dos personas que son iguales.

Cuando hablamos de objetos, hay una ambigüedad similar. Por ejemplo, si dos objetos de tipo `Point` son iguales, ¿nos referimos a que contienen los mismos atributos o que son realmente el mismo objeto?

Es por esto que en Python se diferencia entre **igualdad** e **identidad**.

El operador igualdad (`==`) es un operador de comparación que compara los valores de ambos objetos.

Mientras que el operador identidad (`is`) decide si ambos operandos se refieren exactamente al mismo objeto (es decir si están en el mismo lugar en la memoria).

Igualdad e Identidad

Por ejemplo:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> p1 is p2
False
```

Aunque p1 y p2 contengan los mismos atributos, no son el mismo objeto. En cambio, si asignamos p1 a p2, entonces ambas variables son alias para el mismo objeto

```
>>> p1 = p2
>>> p1 is p2
True
```

Para comprobar si dos objetos son iguales en el sentido de que contienen los mismos atributos, podemos sobrecargar el operador (==) utilizando el método `__eq__` en la definición de la clase.

Igualdad e Identidad

```
class Point:
    def __init__(
        self, x: float, y: float
    ) -> None:
        self.x = x
        self.y = y

    ...

    def __eq__(
        self, other: "Point"
    ) -> bool:
        return (
            self.x == other.x
            and self.y == other.y
        )
```

Una vez que hicimos esto, podemos comparar por igualdad los objetos de tipo `Point`:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(3, 4)
>>> p1 == p2
True
```

Supongamos que necesitamos en nuestro programa representar rectángulos. La pregunta es: ¿qué información necesitamos para representar un rectángulo?

Una forma de representarlo es, entonces, utilizar tres atributos, el ancho, el alto y la esquina superior izquierda.

Definimos una nueva clase:

```
class Rectangle:
    def __init__(
        self,
        width: float,
        height: float,
        corner: Point,
    ) -> None:
        self.width = width
        self.height = height
        self.corner = corner
```

Esta clase la podemos usar del siguiente modo:

```
>>> corner = Point(400, 500)
>>> rectangle = Rectangle(100, 200, corner)
```

Esto nos muestra que podemos incluir un objeto dentro de otro objeto. El operador punto que utilizamos para acceder a los atributos es **composicional**:

```
>>> rectangle.corner.x  
400
```

Ejercicio Definir métodos `__str__` y `__eq__` apropiados para rectángulos.

Instancias como valores de retorno

Una función puede devolver una instancia de un objeto.

```
def find_center(  
    rectangle: Rectangle,  
) -> Point:  
    medio_x = (  
        rectangle.corner.x  
        + rectangle.width / 2  
    )  
    medio_y = (  
        rectangle.corner.y  
        - rectangle.height / 2  
    )  
    return Point(medio_x, medio_y)
```

Instancias como valores de retorno

Para utilizar esta función, pasamos como argumento un rectángulo y asignamos el resultado a una variable:

```
>>> corner = Point(400, 500)
>>> rectangle = Rectangle(100, 200, corner)
>>> centro = find_center(rectangle)
>>> print(centro)
(450.0, 400.0)
```

Como ya dijimos, en Python los objetos son **mutables**, es decir, podemos cambiar sus atributos asignando otros valores.

Ejemplo:

```
>>> rectangle.width = rectangle.width + 50
>>> rectangle.height = rectangle.height + 100
```

Mutabilidad

Podríamos encapsular esta lógica utilizando una función que nos permita agrandar un rectángulo por cualquier monto:

```
def grow_rectangle(  
    rect: Rectangle, dw: float, dy: float  
) -> None:  
    rect.width = rect.width + dw  
    rect.height = rect.height + dy
```

Por ejemplo, creamos un rectángulo llamado bob y lo pasamos por la función.

```
>>> corner = (0, 0)  
>>> bob = Rectangle(100, 200, corner)  
>>> grow_rectangle(bob, 50, 100)
```

Mientras la función `grow_rectangle` està ejecutándose, el parámetro `rect` es un alias para `bob`, es decir, se refieren al mismo objeto. Es por eso que cualquier cambio hecho a `rect` se reflejará también en `bob`.

Funciones Puras vs. Modificadoras

A partir de ahora haremos, entonces, una diferencia entre dos tipos de funciones:

- Las **funciones puras** no modifican ningún objeto pasado a las mismas.
- Las **funciones modificadoras** que realizan cambios en los objetos pasados a las mismas.

Por ejemplo, la función `grow_rectangle` es una función modificadora. Una función pura con un efecto similar hubiera creado - dentro del cuerpo de la función - un nuevo objeto de tipo `Rectangle` y aplicado las modificaciones a este, devolviendo el nuevo objeto y dejando el rectángulo original que pasamos como parámetro sin tocar.

Ejercicio Defina una función llamada `mover_rectangulo` que tome un rectángulo y dos parámetros `dx` y `dy`. Esta función debería cambiar de posición el rectángulo sumando `dx` a la coordenada `x` de la esquina superior izquierda y del mismo modo sumar `dy` a la coordenada `y` de la esquina superior izquierda. Haga las dos versiones, una función pura y una función modificadora.

Copiando objetos

Supongamos que tengo un objeto tipo `Point` y uno de tipo `Rectangle`.

```
>>> p = Point(3, 4)
>>> r = Rectangle(10, 2, p)
```

Si quisiera crear un objeto de tipo `Point` igual al objeto `p`, intuitivamente podríamos suponer que tenemos que hacer lo siguiente:

```
>>> q = p
```

El problema es que aquí no estamos realmente instanciando un nuevo objeto, si no que hay un solo objeto y dos nombres distintos para acceder a él. Eso se comprueba sencillamente cambiando los valores en un objeto y viendo que los cambios se reflejan en ambas variables:

```
>>> q.x = 2
>>> print(p.x)
2
```

Copiando Objetos

Podemos usar el operador `is` para confirmar si dos variables se refieren al mismo objeto:

```
>>> p is q
True
```

Usualmente, necesitamos crear objetos que sean iguales a uno que tenemos, pero que sean *independientes* de este. Podemos usar para ello las funciones del modulo de Python `copy`.

```
>>> import copy
>>> q = copy.copy(p)
>>> p is q
False
```

El módulo `copy` es genérico y lo podemos utilizar con cualquier objeto.

Algo importante es que cuando tenemos un objeto dentro de otro (composición), los objetos embebidos no se copian, si no que siguen siendo referencias al mismo objeto subyacente.

Copiando Objetos

Si necesitamos copiar el objeto y todos los objetos contenidos dentro de forma recursiva, hay que utilizar una **copia profunda** o *deep copy*.

```
>>> import copy
>>> r2 = copy.deepcopy(r)
>>> r is r2
False
>>> r.corner is r2.corner
False
```

La principal motivación de la programación orientada a objetos es agrupar la declaración de variables (atributos) y de funciones relacionadas (métodos).

Hemos utilizado métodos anteriormente, en objetos *built-in*. Por ejemplo, `.keys()` y `.values()` son métodos para diccionarios.

Cada método está asociado con una clase y pueden ser invocados en instancias de dicha clase. Los métodos son muy parecidos a las funciones, pero tienen dos diferencias:

- Son definidos dentro de una definición de clase, para hacer la relación entre ambos explícita.
- La sintaxis para invocar un método es distinta a la sintaxis para llamar a una función.

Veamos un ejemplo, definiendo una clase `Time` que nos permita representar una hora del día, utilizando horas, minutos y segundos, y que incluya un método `increment` para aumentar la hora actual en cierta cantidad de segundos.

Ejemplo de Clase

```
class Time:
    def __init__(
        self, hh: int, mm: int, ss: int
    ) -> None:
        self.hours = hh
        self.minutes = mm
        self.seconds = ss

    def increment(
        self, seconds: int
    ) -> None:
        self.seconds = (
            seconds + self.seconds
        )

        while self.seconds >= 60:
            self.seconds = self.seconds - 60
```

La sintaxis para definir métodos es muy similar a la de funciones, solo que se definen dentro de la clase (por eso es importante la indentación) y además tiene un primer parámetro, sin tipo, llamado `self`. Este primer parámetro puede llamarse como uno desee, pero es una convención utilizar `self`. El objetivo de «`self`» es hacer referencia al objeto que se está manipulando cuando se llama al método.

Este método lo llamamos como haríamos con cualquier otro:

```
>>> t = Time(19, 20, 40)
>>> t.increment(19)
```

Vemos que el parámetro `self` nunca lo pasamos dentro del paréntesis, ya que este se refiere realmente al objeto que estamos manipulando (el que pusimos delante del punto).

Ejercicio Agregar un método `convert_to_seconds` que convierta el objeto de tipo `Time` a un número que representa cuantos segundos han pasado desde la medianoche. ¿El método que escribió, es una pura o un modificador?

Como ya presentamos en secciones anteriores, utilizando ciertos métodos podemos conseguir aplicar operadores incluidos en el lenguaje a tipos de datos propios, como hicimos con la definición del igual en la clase `Point`. Esto lo podemos hacer para cualquier operador. Por ejemplo, retomando el ejemplo de los puntos, podemos definir la suma de dos puntos (como la suma de ambas componentes) del siguiente modo:

Sobrecarga de operadores

```
class Point:
    def __init__(
        self, x: float, y: float
    ) -> None:
        self.x = x
        self.y = y

    ...

    def __add__(
        self, other: "Point"
    ) -> bool:
        return Point(
            self.x + other.x,
            self.y + other.y,
        )
```

Sobrecarga de operadores

El método `__add__` que acabamos de definir recibe como parámetro otro objeto de tipo punto y devuelve un nuevo objeto que representa la suma entre ambos. Este podemos utilizarlo como cualquier otro método:

```
>>> p = Point (2 , 3)
>>> q = Point (3 , 4)
>>> p.__add__(q)
(5 , 7)
```

Sin embargo, Python también nos permite llamar a esto de forma más elegante:

```
>>> p + q
(5 , 7)
```

Cuando se usa el operador `+`, Python busca en el primer operando un método llamado `__add__` y lo invoca utilizando el segundo operando como argumento. De esta forma, podemos sobrecargar cualquier operador aritmético o de comparación.

En el apunte puede encontrarse una lista incompleta de operadores de Python junto con el método asociado que puede usarse para sobrecargarlo.

La herencia permite añadir nuevos métodos a una clase sin modificar la clase ya existente. Es llamada “herencia” porque la nueva clase hereda todos los métodos de la clase que ya existe. Esta metáfora se extiende y la clase que ya existía se suele llamar **clase padre**. La nueva clase se llama, entonces **clase hija o subclase**.

Ejemplificaremos el uso de la herencia utilizando la clase Square. Un cuadrado no es mas que un caso especial de un rectángulo, por lo que podemos reutilizar las funcionalidades que ya hubiéramos implementado para rectángulos.

```
class Square(Rectangle):
    def __init__(
        self, size: float, corner: Point
    ) -> None:
        self.width = self.size
        self.height = self.size
        self.corner = corner
        self.size = size

    def get_area(self) -> float:
        return self.size**2
```

Vemos que para declarar una herencia, agregamos una clase hija y entre paréntesis, indicamos cuál es su clase padre. En este caso agregamos un método `get_area` que calcula el área del cuadrado.

Además, podemos modificar cualquier método que ya tuviera la clase simplemente volviéndolo a definir. En el ejemplo esto es lo que hacemos con el método constructor. Cuando instanciamos una clase, Python buscará si la clase provee el método constructor. Si no lo hace, buscará en su clase padre y así sucesivamente. Lo mismo ocurre para cualquier método, muchas veces ocurrirá que los métodos son casi los mismos, pero no exactamente iguales.

Importante: Cuando hacemos redefinimos el método constructor, es importante que definamos de forma apropiada los atributos de la clase padre para que todos los métodos puedan trabajar de forma correcta. Es por eso que además de agregar el atributo `size`, también volvemos a definir `width` y `height`.


La definición anterior de la clase Square tiene un problema, al redefinir el método constructor, estamos repitiendo parte del código que ya habíamos implementado en una clase anterior. Para solucionar esto, podemos hacer uso de la función `super`, que ocasiona que temporalmente un método se comporte como en la clase padre. Así, podemos pasarle los argumentos que recibía nuestro constructor en la clase padre y así nos aseguramos de no estar olvidándonos nada y reutilizando el código que ya escribimos.

Herencia - La función super

```
class Square(Rectangle):
    def __init__(
        self, size: float, corner: Point
    ) -> None:
        super().__init__(size, size, corner)
        self.size = size
```

¿PREGUNTAS?

 Apunte de Cátedra
Elaborados por el staff docente.
Será subido al campus virtual de la materia.

 A. Downey et al, 2002.
How to Think Like a Computer Scientist. Learning with Python.
Capitulos 12 a 16

 <https://docs.python.org/es/3/tutorial/classes.html>