



Programación II

Tecnicatura Universitaria en Inteligencia Artificial

2022

Programación Orientada a Objetos

1. Paradigmas de programación

Los paradigmas de programación son modelos para el desarrollo estructurado de algoritmos. Distintos lenguajes de programación adhieren a uno u otro modelo.

1.1. Paradigma imperativo

Un lenguaje imperativo permite escribir programas como un conjunto de instrucciones que se ejecutan una por una, de principio a fin, de modo secuencial excepto cuando intervienen instrucciones de salto de secuencia o control. En los lenguajes imperativos, el concepto central para la ejecución de un programa es el concepto de **estado**, que va modificándose a medida que asignamos y cambiamos valores a **variables**.

Algunos ejemplos de lenguajes que adhieren a este modelo son:

- El tradicional lenguaje **C**, diseñado por Kernighan y Ritchie en el año 1969 y usado ampliamente en la implementación de sistemas operativos.
- **FORTRAN** un lenguaje de programación de propósito general especialmente adaptado al cálculo numérico y a la computación científica.

Este es el paradigma en el que venimos trabajando hasta ahora.

1.2. Paradigma Orientado a Objetos

Este paradigma evolucionó de los lenguajes imperativos.

No es fácil definir la programación orientada a objetos, pero algunas de sus características son:

- Los programas se componen de definiciones de objetos y definiciones de funciones, y la mayor parte de la computación está definida en términos de cambios o modificaciones a dichos objetos.
- Cada definición de objeto se corresponde con algún objeto o concepto del mundo real, y las funciones que operan en dicho objeto se corresponden con alguna interacción entre objetos en el mundo real.

En general, la programación orientada a objetos no nos entrega más poder de cómputo: no podremos hacer con ella nada que no pudiéramos hacer antes. Podemos pensarlo, si se quiere, como una sintaxis alternativa, que presenta de forma más concisa la estructura de nuestro programa.

La idea central de este paradigma fue primero distinguir en el paradigma imperativo dos tipos de instrucciones muy distintas, por un lado la asignación de valores a variables y por otro lado la definición de funciones. Luego, se trató de forzar un diseño de programas donde la asignación de variables y la definición de funciones relacionadas semánticamente con estas variables estuvieran también relacionadas sintácticamente. Es decir, se intenta lograr que las funciones que están fuertemente relacionadas con un conjunto de variables aparezcan “cerca” de estas en el código. El concepto central sigue siendo el de **estado**, pero la diferencia es que ahora, además del estado global del programa, cada objeto tiene un estado propio.

2. Python como lenguaje multiparadigma

El lenguaje de programación Python es ampliamente usado y su popularidad solo va en ascenso, desplazando incluso a lenguajes tradicionales de algunas áreas de dominio específico.

- En el área de cálculo numérico y computación científica, ha comenzado a desplazar al tradicional FORTRAN.
- En el área de desarrollo de aplicaciones para la web, comienza a ser cada vez más utilizado, junto con lenguajes como Java, Node o PHP.
- Para aplicaciones de estudio de grandes conjuntos de datos y estadísticas, se ha cementado como uno de los lenguajes principales, junto al lenguaje R.

Entre las razones de que se pueda usar el mismo lenguaje para desarrollar aplicaciones tan disímiles se encuentran la versatilidad de Python como lenguaje de programación y su relativa simplicidad. En este sentido, Python fue diseñado para soportar múltiples paradigmas de programación. En mayor o menor medida, Python soporta los paradigmas imperativo, funcional, orientado a objetos y procedural.

Problema: Dada una lista de números, calcular la suma del doble de los cuadrados de los números en la lista. ¹

Solución: Suponiendo que la lista es A y tiene n elementos, la solución es calcular la siguiente sumatoria:

$$sum = \sum_{i=0}^n 2 * (A[i]^2)$$

2.1. Como lenguaje imperativo

En una solución en estilo imperativo, el foco estará en *cómo* obtenemos el resultado. El programa tendrá un estado que irá cambiando a medida que el programa progresa, con el fin de encontrar una solución.

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

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

print(sum)
```

¹Contenido adaptado del blog <https://newrelic.com/blog/nerd-life/python-programming-styles>

En este ejemplo, el estado del programa está íntimamente ligado a las variables `sum` que lleva la suma parcial de los elementos vistos hasta ahora, y las variables `x`, `square` y `doble` que computan los resultados parciales para cada uno de los elementos vistos.

2.2. Como lenguaje orientado a objetos

En una solución orientada a objetos, el foco estará en conseguir que el programa sea lo más reutilizable posible. Usaremos un objeto propio para definir la funcionalidad que buscamos.

```
class MyList():
    def __init__(self, any_list: list[int]) -> None:
        self.any_list = any_list

    def square(self) -> list[int]:
        return [ x * x for x in self.any_list ]

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

    def sum(self) -> int:
        return sum(self.doble())

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

print(my_list.sum())
```

Veremos la sintaxis de esta solución en detalle en secciones futuras, pero por ahora podemos adelantar que diremos que el *objeto* `my_list` es una *instancia* de la clase `MyList`, y que el *método* `sum` realiza los cálculos, sin necesidad de conocer desde fuera de la clase los métodos `square` y `doble`.

3. Programación orientada a objetos

3.1. Historia

El lenguaje de programación **Simula** sentó en la década del '60 las bases conceptuales de clase y objeto. Estos conceptos se trasladaron a los lenguajes **Smalltalk** y **Lisp** en la década de los '70. Fue en los años '80 que el interés por estos conceptos se extendieron en el público general de programadores, y se comenzó a trabajar en extensiones del lenguaje C para soportar clases.

Estos esfuerzos dieron nacimiento a los lenguajes **Objective-C** y **C++**, el último de los cuales sigue siendo popular hoy en día. También nace en esta época el lenguaje **Eiffel**, el primero explícitamente diseñado para ser orientado a objetos. Durante la década del '90 la programación orientada a objetos se convirtió en el paradigma dominante para el desarrollo de software industrial y surgieron decenas de lenguajes.

La popularidad se incrementó a medida que también lo hacía la necesidad de programar interfaces gráficas. Al mismo tiempo, investigadores académicos buscaban formalizar la teoría de la programación orientada a objetos mediante la investigación de tópicos como la abstracción de datos y la modularización de programas. El científico de la computación Alan Kay fue pionero en esta investigación y fue el quien acuñó los términos *objeto* y *clase*. Alan Kay recibió el premio Turing, honor equivalente a un premio Nobel en el campo de la computación, "por ser pionero en muchas de las ideas en la raíz de los lenguajes orientados a objetos contemporáneos [...] y por sus contribuciones fundamentales a la computadora personal".

En 1996 surge un desarrollo llamado **JAVA**, en principio pensado como una extensión de C++, que se transformó en un lenguaje independiente. Es considerado por muchos como el lenguaje orientado a objetos definitivo, debido a su explícita cohesión con este paradigma.

Actualmente, el paradigma de programación orientada a objetos sigue siendo el paradigma principal utilizado en la industria del desarrollo de software.

Más recientemente comenzaron a popularizarse lenguajes que soportan un enfoque no purista o mixto de la orientación a objetos, entre ellos **Python**, **Ruby** y **Javascript**.

3.2. Introducción

La programación orientada a objetos es un paradigma de programación que nos permite organizar el código de manera que se asemeje a como pensamos en la vida real. Así, pensamos en los problemas como un conjunto de **objetos** que se relacionan entre sí. Estos **objetos** nos permiten agrupar un conjunto de variables y funciones relacionadas en un mismo espacio de nombres, facilitando la abstracción de pensamiento y la modularización del programa.

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 gato podrían ser caminar o maullar. A estos comportamientos o funcionalidad propia de cada objeto los llamaremos **métodos**.

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

3.3. Clases como tipos de datos

Las **clases** proveen una forma de empaquetar datos y funcionalidad juntos. Al crear una nueva clase, se crea un nuevo tipo de objeto, permitiendo crear nuevas instancias de ese tipo. Cada instancia de clase puede tener atributos adjuntos para mantener su estado. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

Podemos pensar a la clase como una plantilla para la creación de nuevos objetos.

Las clases nos permiten modularizar los datos y la funcionalidad asociada a un tipo de objeto. Una forma sencilla de pensar a las clases es pensarlas como un nuevo tipo de dato. Así, mientras representaremos a distintos gatos con distintos objetos, representaremos la idea abstracta de un gato mediante una clase Gato. 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. Cada objeto individual tiene sus propios atributos y métodos, la clase sólo nos provee una vista unificada de lo que tienen en común todos los objetos de este tipo.

4. POO en Python

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 implementados como clases y sus instancias concretas son objetos. Por lo tanto, lo único que nos falta aprender es a crear tipos definidos por nosotros. Esto lo haremos trabajando con un tipo de datos proveniente de la matemática, el Punto. En notación matemática, un punto usualmente se escribe entre paréntesis con una coma separando las coordenadas. Por ejemplo, $(0, 0)$ representa el origen y (x, y) representa el punto x unidades a la derecha e y unidades hacia arriba (partiendo desde el origen y para el caso de x e y positivos).

Una forma natural de representar un punto en Python es con dos números de punto flotante. La pregunta es, entonces, ¿cómo agrupar estos dos valores en un objeto? La forma fácil (y ~~sueia~~) es utilizar una lista o una tupla para agrupar los valores. Alternativamente, se puede utilizar una **clase** para implementar un tipo de datos definido por el usuario.

Una definición básica es como sigue:

```
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) + ')'
```

Al crear la clase `Point`, hemos creado un nuevo tipo de datos, también llamado `Point`. Los miembros de este tipo se llaman también **instancias** del tipo o bien **objetos** del tipo. El proceso de crear una nueva instancia de un tipo se llama **instanciación**.

Nota Por convención, nombraremos a las clases utilizando la primera letra en mayúscula, de forma de poder distinguirlas rápidamente.

El **método constructor** `__init__` es un método especial que es invocado cuando un objeto es creado. El método `__init__` establece un primer parámetro especial que se suele llamar `self` (veremos qué significa este nombre en la siguiente sección). Pero puede especificar otros parámetros siguiendo las mismas reglas que cualquier otra función. ²

Para crear un objeto de una clase determinada, es decir, instanciar una clase, se usa el nombre de la clase y a continuación se añaden paréntesis (como si se llamara a una función).

```
mi_punto = Point(3, 4)
```

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` → la instancia nueva, `x` → 3, `y` → 4.
3. El constructor asigna los atributos `self.x` → 3, `self.y` → 4.
4. Cuando finaliza la ejecución del constructor, se asigna a `mi_punto` → la instancia de `Point` recién creada.

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` ³:

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

4.1. Atributos

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

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

²Un constructor puede recibir 0, 1 o más parámetros. Es importante notar que `self` normalmente no se considera un parámetro. También es importante notar que los parámetros recibidos por el constructor y los atributos del objeto poco tienen que ver.

³Si no se sobrecarga la función `print`, se obtiene un resultado similar a `<__main__.Point instance at 80f8e70>`. Este resultado indica que `mi_punto` es una instancia de la clase `Point` (y el módulo en el que fue definida). `80f8e70` es el identificador único para este objeto, escrito en hexadecimal (base 16). Por lo general todas las clases deben redefinir este método para obtener una representación útil.

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

```
>>> mi_punto.x
3.4
```

La expresión `mi_punto.x` significa, "ve al objeto `mi_punto`, y obtén el valor del atributo `x`". Incluso podemos asignar este valor a una variable, por ejemplo:

```
x = mi_punto.x
```

No hay ningún conflicto entre la variable `x` y el atributo `x`. El propósito de la notación punto es identificar sin ambigüedades a que variable nos estamos refiriendo.

4.1.1. 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.

Es decir, las variables de instancia son para datos únicos y propios para cada objeto y las variables de clase son para atributos que deban ser compartidos por todas las instancias de 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.

Los **atributos de instancia** son creados usando la sintaxis `instance.attr` o `self.attr`. Como se ha dicho son locales para esa instancia y por tanto solo accesibles desde una de ellas. Es normal definir los atributos de instancia dentro del método constructor. Si no se hace así (aunque solo sea inicializarlos a un valor nulo) el código pierde legibilidad. En cuanto a su utilidad, genéricamente podemos decir que permiten definir un determinado “estado” a ese objeto particular y es lo que lo diferencia de otros objetos de su clase.

Los **atributos de clase** se definen fuera de cualquier método de la clase, normalmente justo debajo del docstring. Se pueden referenciar directamente desde la clase y también se puede hacer referencia a ellas desde cualquier instancia de la clase. Su uso más común es para crear una constante enumerada que utilizaremos en la clase.

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

4.2. Igualdad e Identidad

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).

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 `p2` a `p1`, 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.

```
class Point:
    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) + ')'

    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
```

4.3. Composición

Supongamos que necesitamos en nuestro programa representar rectángulos. La pregunta es: ¿qué información necesitamos para representar un rectángulo? Para mantener los ejemplos sencillos, supondremos que un rectángulo siempre esta orientado de forma vertical u horizontal, pero nunca en ángulo. Una forma de representarlo es, entonces, utilizar tres atributos, el ancho, el alto y la esquina superior izquierda. De nuevo, 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**, lo que significa que podemos acceder a los atributos de forma anidada. Por ejemplo, para obtener la coordenada x de la esquina superior izquierda del objeto `rectangle`, podemos simplemente utilizar:

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

Aquí, el significado es "ve al objeto `rectangle`, luego elige el atributo llamado `corner`, luego ve a ese objeto y elige el atributo llamado `x`".

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

4.4. Instancias como valores de retorno

Una función puede devolver una instancia de un objeto. Por ejemplo, la siguiente función recibe como argumento un rectángulo y devuelve un punto que contiene las coordenadas del centro del rectángulo:

```
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)
```

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)
```

4.5. Mutabilidad

Como ya dijimos, en Python los objetos son **mutables**, es decir, podemos cambiar sus atributos asignando otros valores. Por ejemplo, para modificar el tamaño de un rectángulo sin alterar su posición, podríamos asignar sus atributos:

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

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
```

Las variables `dw` y `dy` nos indican cuanto crece el rectángulo en cada dirección. Invocar a esta función no devuelve nada, pero tiene como efecto **modificar** el objeto de tipo `Rectangle` que le pasamos como primer argumento. Esto nos enseña algo importante: los objetos que pasamos a una función pueden ser alterados dentro de la función ⁴ y dichos cambios sobreviven, aún cuando el marco de ejecución de dicha función haya terminado. Por ejemplo, creamos un rectángulo llamado `bob` y lo pasamos por la función.

⁴A veces, se dice que los argumentos en Python se pasan por **referencia de objeto**, indicando que los objetos pueden mutar dentro de una función.


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

Mientras la función `grow_rectangle` está 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`.

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.

Cualquier cosa que pueda hacer con funciones modificadoras también puede hacerse con funciones puras ⁵. Hay alguna evidencia de que los programas que solo utilizan funciones puras son más rápidos de desarrollar y contienen menos errores, sin embargo, los modificadores son convenientes en algunos casos, e incluso, pueden ser más eficientes.

Ejercicio Como 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.

4.6. 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
```

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

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

⁵De hecho, ciertos lenguajes de programación solo admiten funciones puras

 Las asignaciones no copian datos, solamente asocian nombres a objetos.

Usualmente, necesitamos crear objetos que sean iguales a uno que tenemos, pero que sean *independientes* de este. Podemos usar para ello las funciones del módulo 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. Por ejemplo, podemos usarlo para copiar rectángulos:

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

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. Por eso decimos que `copy.copy` es una **copia superficial** (o *shallow copy*, en inglés).

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
```

4.7. Métodos

Recordemos que la principal motivación de la programación orientada a objetos es agrupar la declaración de variables y de funciones relacionadas. La parte de la declaración de variables se logra con los atributos, mientras que para definir funciones relacionadas a un objeto es necesario introducir el concepto de **método**.

Hemos utilizado métodos anteriormente, en objetos *built-in*. Por ejemplo, `.keys()` y `.values()` son métodos para diccionarios. Es por eso que ya conocemos la sintaxis para invocar métodos, debemos escribir el objeto al cual vamos a aplicarlo, seguido de punto y nombre del método, incluyendo cualquier parámetro, si los hubiera, entre paréntesis. Por ejemplo:

```
>>> d = {'a': 1, 'b': 2}
>>> d.values()
dict_values([1, 2])
```

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.

En las secciones anteriores, también hemos utilizado métodos. `__init__`, `__str__` y `__eq__` son también métodos. Podemos definir dentro de una clase tantos métodos como sean necesarios. 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.

```
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
            self.minutes = self.minutes + 1

        while self.minutes >= 60:
            self.minutes = self.minutes - 60
            self.hours = self.hours + 1
```

Podemos ver que en esta definición de clase, además del método constructor, está definido el método que queríamos. 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 En la definición del método constructor de `Time`, falta validar que los parámetros sean correctos. Reescribirlo para agregar dichas validaciones.

Para los métodos realizamos la misma distinción entre funciones puras y modificadoras. Por ejemplo, el método `increment` es un modificador, porque cambia el estado del objeto.

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 escribíó, es puro o un modificador?

Nota: Los métodos pueden utilizar tantos parámetros como deseen. Al igual que con las funciones, se pueden pasar parámetros se pueden pasar por posición o por nombre y se puede definir que tengan un valor por defecto si no se especifica uno.

4.8. Métodos constructores y sobrecarga de operadores

El método constructor `__init__`, como ya dijimos, será llamado internamente por Python cada vez que instanciamos un nuevo objeto de una clase. El constructor hace explícito que atributos maneja la clase. Se considera una buena práctica definir un constructor para cada clase que escribamos, ya que facilitan la lectura del código y documenta la forma apropiada de utilizar una clase.

Además, 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:

```
class Point:
    # definiciones anteriores ...
    def __add__(self, other: 'Point') -> 'Point':
        return Point(self.x + other.x, self.y + other.y)
```

Notar que cuando definimos métodos que reciban parámetros del mismo tipo que la clase que estamos definiendo, o devuelvan datos de este tipo, es necesario marcar la anotación de tipo entre comillas.

El método `__add__` que acabamos de definir recibe como parámetro otro objeto de tipo `Point` 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)
```

Lo que ocurre es que, cuando utilizamos 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.

Estos son los llamados métodos especiales, **métodos mágicos**, o *dunders*. Son métodos que, si están definidos para un objeto, Python se encargará de invocarlos cuando se realizan ciertas operaciones con ellos. ⁶

La siguiente tabla sirve como referencia (incompleta) de operadores y métodos mágicos:

| Operador | Método Mágico |
|--------------------|---------------------------------------|
| <code>+</code> | <code>__add__(self, other)</code> |
| <code>-</code> | <code>__sub__(self, other)</code> |
| <code>*</code> | <code>__mul__(self, other)</code> |
| <code>/</code> | <code>__truediv__(self, other)</code> |
| <code>%</code> | <code>__mod__(self, other)</code> |
| <code>**</code> | <code>__pow__(self, other)</code> |
| <code><</code> | <code>__lt__(self, other)</code> |
| <code><</code> | <code>__gt__(self, other)</code> |
| <code><=</code> | <code>__le__(self, other)</code> |
| <code>>=</code> | <code>__ge__(self, other)</code> |
| <code>==</code> | <code>__eq__(self, other)</code> |
| <code>!=</code> | <code>__ne__(self, other)</code> |

⁶El nombre *dunder* es un anglicismo proveniente de la contracción de *double underscore*, que significa "doble guión bajo", dado que muchos de estos métodos se escriben empezando y terminando en doble guión bajo.

5. Herencia

La funcionalidad que más comúnmente es asociada con la programación orientada a objetos es la herencia. La herencia es la habilidad de definir una nueva clase que es una versión modificada de una clase ya existente.

La principal ventaja de esta funcionalidad es que puedes 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**.

La herencia es una funcionalidad poderosa. Algunos programas que serían complicados sin herencia pueden escribirse de forma simple y concisa utilizando esta funcionalidad. Además, la herencia facilita la reusabilidad del código, ya que permite personalizar el comportamiento de una clase sin necesidad de modificar la clase. En algunos casos, la estructura de la herencia incluso refleja la naturaleza del problema, haciendo el programa más fácil de entender. Por otro lado, la herencia puede volver en algunos casos difícil de leer el programa. Cuando un método es invocado, puede ser difícil encontrar en que clase fue definido, ya que podría haber sido definido en la clase padre o en la clase hija. Así, el código relevante para un objeto dado puede incluso estar desperdigado en diferentes módulos. Al mismo tiempo, muchas de las cosas que realizamos utilizando herencia pueden realizarse también sin ella. Si la estructura del problema que estamos tratando involucra naturalmente una herencia, puede ser buena idea utilizarla, pero intentar aplicarla a cualquier tipo de problema puede traer más problemas de los que resuelve.

Ejemplificaremos el uso de la herencia utilizando la clase **Square**. Un cuadrado no es más 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
```

En este ejemplo vemos que, para declaración de una herencia, definimos la clase nueva e indicamos cuál será su clase padre entre paréntesis. Luego podemos agregar todos los métodos que querriamos. En este caso, agregamos un método `get_area` que nos permite obtener el área de nuestro 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.

El *override* ⁷ de métodos es una habilidad de cualquier lenguaje orientado a objetos que permite que una clase hija provea una implementación más específica de un método que ya provee una clase ancestro.

La versión o sabor del método que será ejecutado será determinado por la clase más específica que provea el método.

En el caso particular del constructor, debemos prestar especial atención. Podemos sumar nuevos atributos,

⁷En español, *override* puede traducirse como anulación, supresión, superación o sustitución; pero ninguna resulta del todo adecuada. Así, seguiremos la convención general y nos referiremos a este concepto directamente con su nombre en inglés.

pero es necesario 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 setear el atributo `size`, seteamos además los atributos `width` y `height` que definía su clase padre.

5.1. La función super

La definición anterior de la clase `Square` presenta un problema: al redefinir el método constructor, estamos repitiendo parte del código que ya habíamos implementado en `Rectangle`. Para evitar esto, Python nos provee una función llamada `super` con la cual podemos utilizar dentro de una definición de clase, los métodos de la clase padre correspondiente y así replicar el comportamiento del método en la clase padre.

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

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

La función `super` ocasiona que temporalmente un método se comporte como en la clase padre. Así, podemos pasarle los argumentos que recibía nuestro constructor de `Rectangle` y así nos aseguramos de no estar olvidándonos nada y reutilizando el código que ya escribimos.

6. Errores comunes

Cuando comienzas a trabajar con objetos, es probable que encuentres algunas excepciones nuevas. Si intentas acceder a un atributo que no existe, obtienes un `AttributeError`:

```
>>> p = Point(3, 4)
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

Si no sabes bien de que tipo es un objeto, puedes utilizar la función `type` para verificarlo:

```
>>> type(p)
<class '__main__.Point'>
```

Python también nos provee la función `isinstance` para verificar si un objeto es instancia de una clase. Por ejemplo:

```
>>> isinstance(5, int)
True
>>> isinstance(2.3, int)
False
>>> isinstance("foo", str)
True
>>> isinstance([1,2,3,4,5], list)
True
```

Si no sabes bien si un objeto tiene un atributo en particular, puedes utilizar la función incorporada `hasattr`⁸:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

⁸`hasattr` proviene del inglés *has attribute*.

Para la función `hasattr`, el primer argumento puede ser cualquier objeto, el segundo argumento es un string que contiene el nombre del atributo.

Referencias

- [1] A. Downey et al, 2002. How to Think Like a Computer Scientist. Learning with Python. Capítulos 12 a 16
- [2] <https://ellibrodepython.com/>
- [3] <https://docs.python.org/es/3/tutorial/classes.html>
- [4] Apunte de la Facultad de Ingeniería de la UBA, 2da Edición, 2016. Algoritmos y Programación I: Aprendiendo a programar usando Python como herramienta. Unidad 14