Tecnicatura Universitaria en Programación
Programación II

UNIDAD N° 3: Programación Orientada a Objetos

Índice

OOP **3**
Introducción 3
Conceptos básicos de POO 3
Propiedades fundamentales de la programación orientada a Objetos **5**
Abstracción 5
Encapsulación 5
Herencia 6
Polimorfismo 7
Clases y Objetos **8**
El método __init__() 9
Accediendo a los atributos 10
Ejecutando comportamiento 10
Visibilidad de atributos y métodos en Python 11
Trabajar con atributos y métodos privados y públicos 11
Getters y Setters 12
El decorador @property en Python 14
Métodos en Python: instancia, clase y estáticos 16
Atributos en Python: instancia y clase 17
Variables de instancia 17
Variables de clase 17
Herencia **17**
El método __init__() para una sub clase 18
Polimorfismo 19
Sobreescritura de métodos 20
Sobrecarga de métodos 21
Herencia Múltiple **21**
Clase Abstracta **21**
Características 22
raise NotImplementedError() 23
Mapeo de relaciones desde el UML a Python **23**
Modelado de Aplicaciones: UML - Diagrama de clase **24**

Diagramas de estructura 24
Diagrama de clases 25
Nomenclatura del diagrama de clases 25
Atributos 26
Multiplicidad 26
Comportamiento 27
Visibilidad 28
Variables de clase y métodos de clase 28
Relaciones 29
Asociaciones 29
Dependencia 30
Agregaciones 31
Composiciones 31
Herencia 32
Clases abstractas 32
Bibliografía **34**
Versiones **34**
Autores **34**

OOP

Introducción

Si queremos modelar en un estilo orientado a objetos, primero debemos aclarar qué significa Orientación a objetos. La introducción de la orientación a objetos se remonta a la década de 1960, cuando se presentó el lenguaje de simulación SIMULA, basado en un paradigma que era lo más natural posible para los humanos para describir el mundo.

No existe una única definición para la orientación a objetos. Sin embargo, existe un consenso general sobre las propiedades que caracterizan la orientación a objetos.

La programación orientada a objetos (POO) es uno de los enfoques más eficaces para escribir software. En la programación orientada a objetos, se escriben **clases** que representan cosas y situaciones del mundo real y se crean **objetos** basados en estas clases. Cuando escribes una clase, defines el comportamiento general que puede tener toda una categoría de objetos.

Cuando crea objetos individuales de la clase, cada objeto se equipa automáticamente con el comportamiento general; Luego puedes darle a cada objeto los rasgos únicos que desees. Crear un objeto a partir de una clase se llama creación de **instancias** y se trabaja con instancias de una clase.

También se pueden definir clases que amplíen la funcionalidad de clases existentes, de modo que clases similares puedan compartir una funcionalidad común y puedas hacer más con menos código.

Aprender sobre programación orientada a objetos le ayudará a ver el mundo como lo ve un programador. Le ayudará a comprender su código, no solo lo que sucede línea por línea, sino también los conceptos más amplios detrás de él. Conocer la lógica detrás de las clases lo capacitará para pensar de manera lógica, de modo que pueda escribir programas que aborden de manera efectiva casi cualquier problema que encuentre.

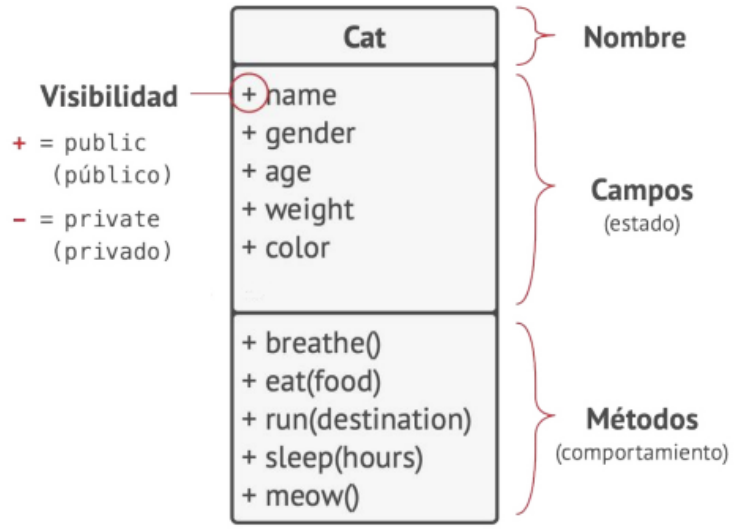
Conceptos básicos de POO

Las clases describen los atributos y el comportamiento de un conjunto de objetos de manera abstracta y así agrupa características comunes de los objetos del mundo real.

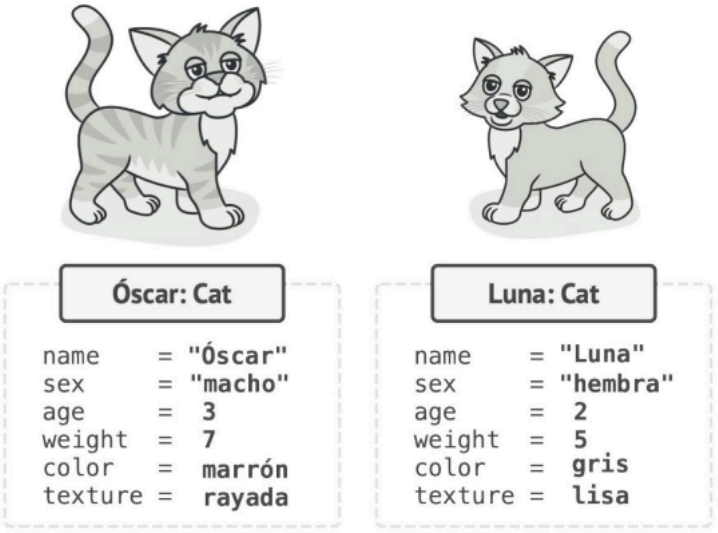
Digamos que tienes un gato llamado Felix. Felix es un **objeto**, una instancia de la **clase** Cat. Cada gato tiene varios **atributos** estándar: nombre, sexo, edad, peso, color, comida favorita, etc. Estos son los campos de la clase.
Además, todos los gatos se comportan de forma similar: respiran, comen, corren, duermen y maúllan. Estos son los **métodos** de la clase.

La información almacenada dentro de los campos del objeto suele denominarse **estado**. El estado de un objeto viene determinado por los valores que toman sus datos o atributos.

Colectivamente, podemos referirnos a los campos y los métodos como los **miembros** de una clase.



Esto es un diagrama de clases en UML. Más adelante se introducirá el tema.



Ejemplos de objetos de la clase Cat.

Por lo tanto, una clase es como una plantilla que define la estructura de los objetos, que son instancias concretas de esa clase.

La identidad permite diferenciar los objetos de modo no ambiguo independientemente de su estado. Es posible distinguir dos objetos en los cuáles todos sus atributos sean iguales. Cada objeto posee su propia identidad de manera implícita. Cada objeto ocupa su propia posición en la memoria de la computadora.

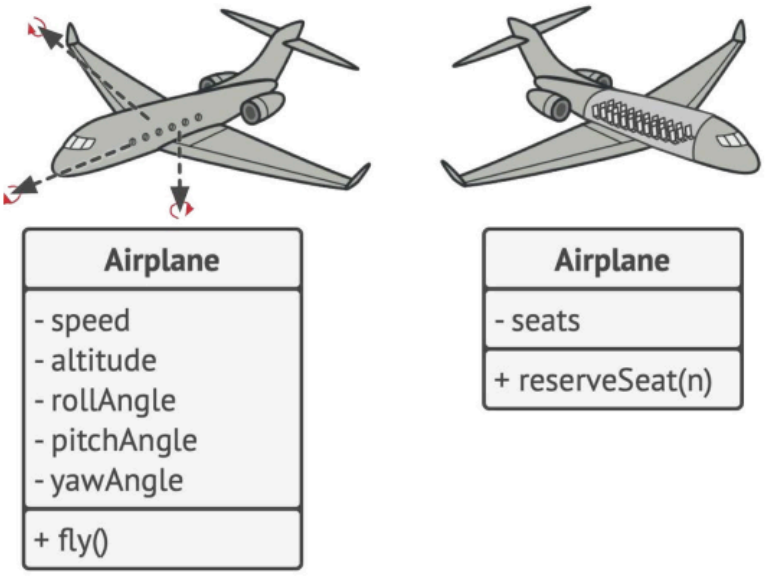
Propiedades fundamentales de la programación orientada a Objetos

Abstracción

La mayoría de las veces, cuando creas un programa con POO, das forma a los objetos del programa con base a objetos del mundo real. Sin embargo, los objetos del programa no representan a los originales con una precisión del 100 % (y rara vez es necesario que lo hagan).

En su lugar, tus objetos tan solo copian atributos y comportamientos de objetos reales en un contexto específico, ignorando el resto.

Por ejemplo, una clase Avión probablemente podría existir en un simulador de vuelo y en una aplicación de reserva de vuelos. Pero, en el primer caso, contendría información relacionada con el propio vuelo, mientras que en la segunda clase sólo habría que preocuparse del mapa de asientos y de los asientos que estén disponibles.



Distintos modelos del mismo objeto del mundo real.

La Abstracción es el modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los datos relevantes a este contexto con gran precisión, omitiendo el resto.

Encapsulación

Para arrancar el motor de un auto, tan solo debes girar una llave o pulsar un botón. No necesitas conectar cables bajo el capó, rotar el cigüeñal y los cilindros, e iniciar el ciclo de potencia del motor. Estos detalles se esconden bajo el capó del auto. Sólo tienes una **interfaz** simple: un interruptor de encendido, un volante y unos pedales. Esto ilustra el modo en que

cada objeto cuenta con una interfaz: una parte pública de un objeto, abierta a interacciones con otros objetos.

La encapsulación es la capacidad que tiene un objeto de esconder partes de su estado y comportamiento de otros objetos, exponiendo únicamente una interfaz limitada al resto del programa.

Encapsular algo significa hacerlo privado y, por ello, accesible únicamente desde dentro de los métodos de su propia clase.

La encapsulación oculta lo que hace un objeto de lo que hacen otros objetos y del mundo exterior, por lo que se denomina también ocultación de datos.

Herencia

La herencia es la capacidad de crear nuevas clases sobre otras existentes. La principal ventaja de la herencia es la reutilización de código. Si quieres crear una clase ligeramente diferente a una ya existente, no hay necesidad de duplicar el código.

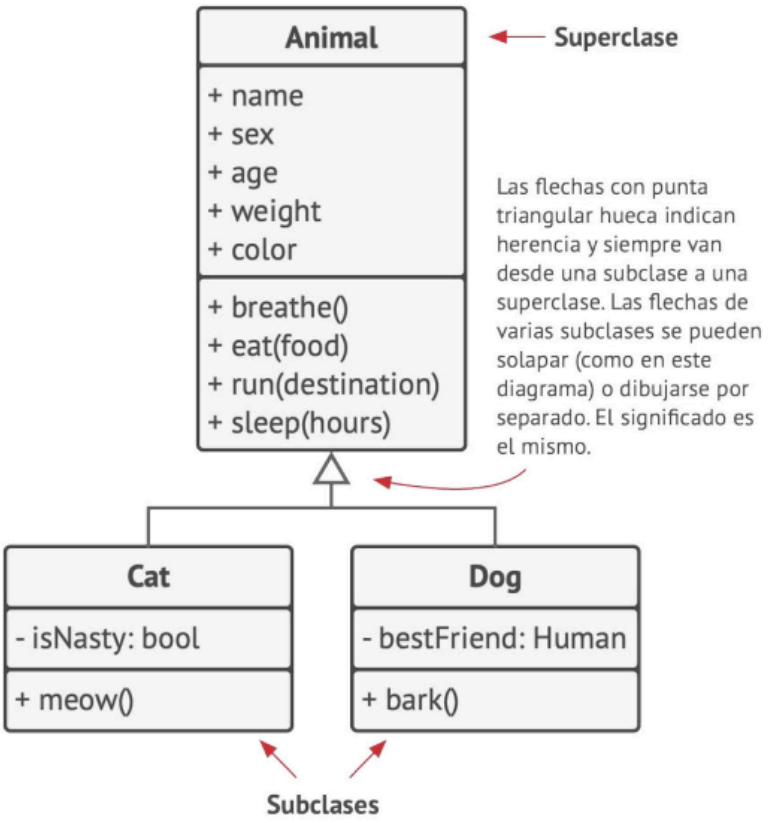
En su lugar, extiendes la clase existente y colocas la funcionalidad adicional dentro de una subclase resultante que hereda los campos y métodos de la superclase.

La consecuencia del uso de la herencia es que las subclases tienen la misma interfaz que su clase padre. No puedes esconder un método en una subclase si se declaró en la superclase.

En la mayoría de los lenguajes de programación una subclase puede extender una única superclase.

Por ejemplo, digamos que tu vecino tiene un perro llamado Fido. Resulta que perros y gatos tienen mucho en común: nombre, sexo, edad y color, son atributos tanto de perros como de gatos. Los perros pueden respirar, dormir y correr igual que los gatos, por lo que podemos definir la clase base Animal que enumerará los atributos y comportamientos comunes.

Una clase padre, como la que acabamos de definir, se denomina superclase. Sus hijas son las subclases. Las subclases heredan el estado y el comportamiento de su padre y se limitan a definir atributos o comportamientos que son diferentes. Por lo tanto, la clase Cat contendrá el método maullar y, la clase Dog, el método ladrar .



Asumiendo que tenemos una tarea relacionada, podemos ir más lejos y extraer una clase más genérica para todos los Organismo vivos, que se convertirá en una superclase para Animal y Planta . Tal pirámide de clases es una jerarquía. En esta jerarquía, la clase Cat lo hereda todo de las clases Animal y Organismo.

Las subclases pueden sobrescribir el comportamiento de los métodos que heredan de clases padre. Una subclase puede sustituir completamente el comportamiento por defecto o limitarse a mejorarlo con líneas de código extras.

Polimorfismo

El polimorfismo es la propiedad que permite tener el mismo nombre de método en clases diferentes y que actúe de modo diferente en cada una de ellas.

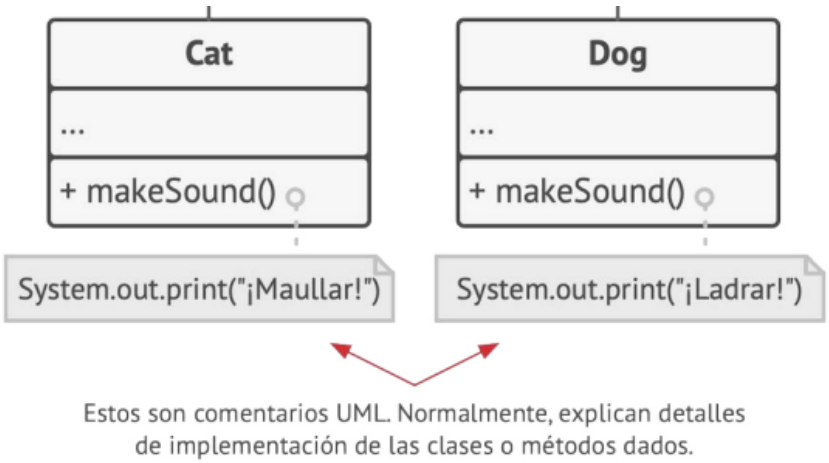
Veamos algunos ejemplos con animales. La mayoría de los animales pueden emitir sonidos. Podemos anticipar que todas las clases necesitarán un método emitirSonido.

Imagina que ponemos varios gatos y perros dentro de una gran bolsa. Después, con los ojos tapados, vamos sacando los animales de la bolsa, de uno en uno. Al sacar un animal, no sabemos con seguridad lo que es. Pero el animal emitirá un sonido específico, dependiendo de su clase concreta.

El programa no conoce el tipo concreto del objeto contenido dentro de la variable a , pero, gracias al mecanismo especial llamado polimorfismo, el programa puede rastrear la clase

del objeto cuyo método está siendo ejecutado, y ejecutar el comportamiento adecuado.

El polimorfismo es la capacidad que tiene un programa de detectar la verdadera clase de un objeto e invocar su implementación.



Clases y Objetos

Python es un lenguaje de programación orientado a objetos. Casi todo en Python es un objeto, con sus propiedades y métodos.

Comencemos escribiendo una clase simple, Dog que represente un perro, no un perro en particular, sino cualquier perro. ¿Qué sabemos sobre la mayoría de los perros domésticos? Pues todos tienen un nombre y una edad. También sabemos que la mayoría de los perros se sientan y se dan vuelta. Esos dos datos (nombre y edad) y esos dos comportamientos (sentarse y darse vuelta) se incluirán en nuestra clase Dog porque son comunes a la mayoría de los perros. A partir de esta clase se puede instanciar un objeto que represente a un perro.

dog.py > ...

```python
class Dog:
    def __init__(self, name, age):
        """Initialize attributes."""
        self.name = name
        self.age = age

    def sit(self):
        """Simulate a dog sitting in response to a command."""
        print(f"{self.name} is now sitting.")

```

Por convención, los nombres en mayúscula se refieren a clases en Python.

Importante: Codificaremos a cada clase en un archivo .py aparte con el nombre de clase en minúscula.

El método __init__()

Una función que es parte de una clase es un método . Todo lo que aprendiste sobre funciones se aplica también a los métodos; La única diferencia práctica por ahora es la forma en que llamaremos a los métodos. El método __init__() es un método especial que Python ejecuta automáticamente cada vez que creamos una nueva instancia basada en la clase Dog. Es un método de clase.

Este método tiene dos guiones bajos iniciales y dos guiones bajos finales, una convención que ayuda a evitar que los nombres de los métodos predeterminados de Python entren en conflicto con los nombres de sus métodos. Asegúrese de utilizar dos guiones bajos a cada lado de __init__(). Si usa solo uno en cada lado, el método no se llamará automáticamente cuando use su clase, lo que puede resultar en errores difíciles de identificar.

Definimos el método __init__() en este ejemplo para que tenga tres parámetros: self, name y age. El parámetro self es obligatorio en la definición del método y debe aparecer primero, antes que los demás parámetros que son opcionales.

Debe incluirse el parámetro self en la definición porque cuando Python llame a este método más adelante (para crear una instancia de Dog), la llamada al método pasará automáticamente el argumento self. Cada llamada a un método asociado con una instancia pasa automáticamente self, que es una referencia a la instancia misma; le da a la instancia individual acceso a los miembros de la clase. Cuando creamos una instancia de Dog, Python llamará al método __init__() de la clase Dog. Pasaremos a Dog() un nombre y una edad como argumentos; self se pasa automáticamente, por lo que no es necesario que lo pasemos. Siempre que queramos crear una instancia de la clase Dog, proporcionaremos valores solo para los dos últimos parámetros name y age.

app.py > ...

```
1   from dog import *
2
3   my_dog = Dog('Willie', 6)
4
5   print(f"My dog's name is {my_dog.name}.")
6   print(f"My dog is {my_dog.age} years old.")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\valon\Desktop\PythonEjercicios-master\oop> python app.py
My dog's name is Willie.
My dog is 6 years old.
```

La clase Dog que estamos usando aquí es la que acabamos de escribir en el ejemplo anterior. Aquí, le decimos a Python que cree un perro cuyo nombre es 'Willie' y edad sea 6 . Cuando Python lee esta línea, llama al método __init__() con los argumentos 'Willie' y 6. El método __init__() crea una instancia que representa a este perro en particular y establece los atributos name y age utilizando los valores que proporcionamos. Luego, Python devuelve una instancia que representa a este perro. Asignamos esa instancia a la variable my_dog.

La convención de nomenclatura resulta útil en este caso; Por lo general, podemos suponer que un nombre en mayúscula como Dog se refiere a una clase, y un nombre en minúscula como my_dog se refiere a una única instancia creada a partir de una clase.

Nota: *Al método __init__ se lo conoce cómo métodos mágicos en Python. Los métodos mágicos son métodos especiales que tienen doble guión bajo al principio y al final de sus nombres. Son también conocidos en inglés como dunders (de doble underscores). Son utilizadas para crear funcionalidades que no pueden ser representadas en un método regular.*

Accediendo a los atributos

Para acceder a los atributos de una instancia, se utiliza la notación de puntos. Accedemos al valor del atributo name de my_dog escribiendo:

```
5    print(f"My dog's name is {my_dog.name}.")
```

La notación de puntos se usa con frecuencia en Python. Esta sintaxis demuestra cómo Python encuentra el valor de un atributo. Aquí, Python mira la instancia my_dog y luego encuentra el atributo name asociado con my_dog. Este es el mismo atributo al que se hace referencia self.name en la clase Dog.

Ejecutando comportamiento

Después de crear una instancia de la clase Dog, podemos usar la notación de puntos para llamar a cualquier método definido en Dog. Hagamos que nuestro perro se siente.

app.py > ...

```
1   from dog import *
2
3   my_dog = Dog('Willie', 6, 'marron')
4
5   print(f"My dog's name is {my_dog.name}.")
6   print(f"My dog is {my_dog.age} years old.")
7   my_dog.sit()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\valon\Desktop\PythonEjercicios-master\oop> python app.py
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.
```

Para llamar a un método, proporcione el nombre de la instancia (en este caso, my_dog) y el método que desea llamar, separados por un punto. Cuando Python lee my_dog.sit(), busca el método sit()en la clase Dogy ejecuta ese código.

Nota: *Esta forma de acceder a los atributos y llamar a los métodos sigue el principio de acceso uniforme. Este principio establece que no debería haber diferencia sintáctica entre trabajar con un atributo , propiedad calculada, o método de un objeto.*

Visibilidad de atributos y métodos en Python

Python no distingue entre métodos o atributos públicos y privados, sino que todos los miembros dentro de una clase o módulo son públicos y pueden ser accedidos por fuera de ellos.

No obstante, como convención se prefiere un guión bajo antes del nombre de un miembro para interpretar el mismo como protected y dos guiones bajo para interpretarlo como privado.

Los modificadores de acceso cumplen con el propósito de Encapsulamiento. Su misión es hacer inaccesible los detalles internos de la clase, con el fin de evitar que otras entidades sepan de su existencia y accedan a ellos directamente produciendo comportamiento inesperado. Esto, con el fin de minimizar el coupling entre entidades.

mi_clase.py > MiClase

```
1  class MiClase:
2
3      def __init__(self):
4          self._atributo_protejido = 1
5          self.__atributo_privado = ''
6
7      def _metodo_protejido(self):
8          return self._atributo_protejido
9
10     def __metodo_privado(self):
11         return self.__atributo_privado
```

Trabajar con atributos y métodos privados y públicos

Puede utilizar clases para representar muchas situaciones del mundo real. Una vez que escriba una clase, pasará la mayor parte de su tiempo trabajando con instancias creadas a partir de esa clase. Una de las primeras tareas que querrás realizar es modificar los atributos asociados con una instancia en particular. Puede modificar los atributos de una instancia directamente o escribir métodos que actualicen los atributos de formas específicas.

Escribamos una nueva clase que represente un automóvil. Nuestra clase almacenará información sobre el tipo de automóvil con el que estamos trabajando y tendrá un método que resuma esta información:

car.py > Car > _get_short_descriptive

```python
class Car:

    def __init__(self, model: str, year: int):
        """Initialize attributes"""
        self.model = model
        self.year = year
        self._cost = 0

    def get_description(self):
        if self._cost > 0:
            return self._get_full_descriptive()
        else:
            return self._get_short_descriptive()

    def _get_full_descriptive(self):
        full = f"El auto es modelo {self.model} del año {self.year} y la valuación es de {self._cost}"
        return full.title()

    def _get_short_descriptive(self):
        short = f"El auto es modelo {self.model} del año {self.year}, se desconoce su valuación"
        return short.title()
```

app.py > ...

```python
from car import *

my_car = Car('Siena', 1998)
print(my_car.get_description())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\valon\Desktop\PythonEjercicios-master\oop> python app.py
El Auto Es Modelo Siena Del Año 1998, Se Desconoce Su Valuación
```

La clase auto tiene atributos y métodos públicos y protegidos.

Cuando se crea una instancia, los atributos se pueden definir sin pasarlos como parámetros. Estos atributos se pueden definir en el método __init__(), donde se les asigna un valor predeterminado. En este caso el atributo privado _cost le asignamos 0 por default.

Para un atributo público se puede cambiar el valor directamente a través de una instancia, cómo lo vimos anteriormente accediendo con punto: my_object.atribute.

Para un método público se lo llama de la misma manera a través de una instancia cómo sigue: my_object.method( ).

Si bien Python permite acceder más o menos directamente con punto a los tributos y métodos que definimos cómo privados o protegidos (ya que el que sean privados o protegidos es sólo una convención de nombres) los programadores respetan el hecho de que esto no es conveniente.

Getters y Setters

Digamos que la clase House es parte de su programa y que por el momento, la clase solo tiene definido un atributo de instancia de precio .

```
house.py > House > __init__
1   class House:
2
3       def __init__(self, price):
4           self.price = price
```

Este atributo de instancia es público porque su nombre no tiene un guión bajo inicial. Dado que el atributo es público actualmente, es muy probable que usted y otros desarrolladores de su equipo hayan accedido y modificado el atributo directamente en otras partes del programa usando notación de puntos, como esta:

```
9   # Access value
10  obj.price
11  # Modify value
12  obj.price = 40000
```

Pero digamos que se le pide que haga que este atributo esté protegido (no público) y valide el nuevo valor antes de asignarlo . Específicamente, debe verificar si el valor es un flotante positivo.

En este punto, si decide agregar getters y setters, usted y su equipo probablemente entrarán en pánico. Esto se debe a que cada línea de código que acceda o modifique el valor del atributo deberá modificarse para llamar al getter o al setter, respectivamente. De lo contrario, el código se romperá:

```
house.py > ...
1   class House:
2
3       def __init__(self, price):
4           self._price = price
5
6       # getter method
7       def get_price(self):
8           return self._price
9
10      # setter method
11      def set_price(self, price):
12          self._price = price

14  # Changed from obj.price
15  obj.get_price()
16  # Changed from obj.price = 40000
17  obj.set_price(40000)
```

Estos getters y setters no son más que métodos públicos de la clase House que permiten acceder o modificar el valor almacenado en el campo _price.

El decorador @property en Python

Una función decoradora es básicamente una función que agrega nueva funcionalidad a una función que se pasa como argumento. ¿Usar una función decorativa es como agregar chispas de chocolate a un helado? Nos permite agregar nueva funcionalidad a una función existente sin modificarla.

Con @property, usted y su equipo no necesitarán modificar ninguna de esas líneas porque podrán agregar getters y setters "entre bastidores" sin afectar la sintaxis que utilizó para acceder o modificar el atributo cuando era público.

house.py > ...

```python
class House:

    def __init__(self, price):
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if new_price > 0 and isinstance(new_price, float):
            self._price = new_price
        else:
            print("Please enter a valid price")

    @price.deleter
    def price(self):
        del self._price
```

Específicamente, puede definir tres métodos para una propiedad:

Un getter: para acceder al valor del atributo.
Un setter: para establecer el valor del atributo.
Un deleter: para eliminar el atributo de instancia.

El precio ahora está "protegido". Tenga en cuenta que el atributo de precio ahora se considera "protegido" porque agregamos un guión bajo a su nombre en self._price.

```python
@property
def price(self):
    return self._price
```

@property
Se utiliza para indicar que vamos a definir una propiedad. Observe cómo esto mejora inmediatamente la legibilidad porque podemos ver claramente el propósito de este método.
def price(self):
Observe cómo el getter tiene el mismo nombre que la propiedad que estamos definiendo: price. Este es el nombre que usaremos para acceder y modificar el atributo fuera de la clase. El método sólo toma un parámetro formal, self, que es una referencia a la instancia.

return self._price
Esta línea es exactamente lo que se esperaría de un getter normal. Se devuelve el valor del atributo protegido.

Observe cómo accedemos al atributo precio como si fuera un atributo público. No estamos cambiando la sintaxis en absoluto, pero en realidad estamos utilizando el getter como intermediario para evitar acceder a los datos directamente. Respetando el principio de encapsulamiento.

```
@price.setter
def price(self, new_price):
    if new_price > 0 and isinstance(new_price, float):
        self._price = new_price
    else:
        print("Please enter a valid price")
```

@price.setter
Se utiliza para indicar que este es el método setter del precio de la propiedad. Observe que no estamos usando @property.setter, estamos usando @price.setter. El nombre de la propiedad se incluye antes de .setter.
def price(self, new_price):
Observe cómo el nombre de la propiedad se utiliza como nombre del setter. También tenemos un segundo parámetro new_price, que es el nuevo valor que se asignará al atributo de precio si es válido.
Finalmente, tenemos el cuerpo del setter donde validamos el argumento para comprobar si es un float positivo y luego, si el argumento es válido, actualizamos el valor del atributo. Si el valor no es válido, se imprime un mensaje descriptivo. Puede elegir cómo manejar los valores no válidos según las necesidades de su programa.

Observe cómo no estamos cambiando la sintaxis, pero ahora estamos usando un intermediario (el setter) para validar el argumento antes de asignarlo.

```
@price.deleter
def price(self):
    del self._price
```

@price.deleter
Se utiliza para indicar que este es el método de eliminación de la propiedad de precio . Observe que esta línea es muy similar a @price.setter, pero ahora estamos definiendo el método de eliminación, por lo que escribimos @price.deleter.
def price(self)
Este método sólo tiene definido un parámetro formal, self.
del self._price
El cuerpo, donde eliminamos el atributo de instancia.

¿El atributo de instancia se eliminó correctamente? Cuando intentamos acceder a él nuevamente, se genera un error porque el atributo ya no existe.

No es necesario que defina los tres métodos para cada propiedad. Puede definir propiedades de solo lectura incluyendo sólo un método getter. También puede optar por definir un setter y un getter sin deleter.

Métodos en Python: instancia, clase y estáticos

Hemos visto cómo se pueden crear métodos con def dentro de una clase, pudiendo recibir parámetros como entrada y modificar el estado (como los atributos) de la instancia. Pues bien, haciendo uso de los decoradores, es posible crear diferentes tipos de métodos:

- Lo métodos de instancia "normales" que ya hemos visto
- Métodos de clase usando el decorador @classmethod
- Y métodos estáticos usando el decorador @staticmethod

```
mi_clase.py > ...
1   class Clase:
2       def metodo(self):
3           return 'Método normal'
4
5       @classmethod
6       def metododeclase(cls):
7           return 'Método de clase'
8
9       @staticmethod
10      def metodoestatico():
11          return "Método estático"
```

Los métodos de instancia son los métodos normales, que hemos visto anteriormente. Reciben como parámetro de entrada self que hace referencia a la instancia que llama al método. También pueden recibir otros argumentos como entrada. Y como ya sabemos, requieren instanciar un objeto para que puedan ser llamados.

```
14  mi_clase = Clase()
15  mi_clase.metodo()
```

A diferencia de los métodos de instancia, los métodos de clase **@classmethod** reciben como argumento cls, que hace referencia a la clase. Por lo tanto, pueden acceder a la clase pero no a la instancia. No hace falta instanciar un objeto para llamarlos. Pero también se pueden llamar sobre el objeto si se requiere.

```
13  Clase.metododeclase()
```

Por último, los métodos estáticos se pueden definir con el decorador **@staticmethod** y no aceptan como parámetro ni la instancia ni la clase. Es por ello por lo que no pueden modificar el estado ni de la clase ni de la instancia. Pero por supuesto pueden aceptar parámetros de entrada si se requiere.

```
13  Clase.metodoestatico()
```

Atributos en Python: instancia y clase

Variables de instancia

También llamadas atributos o variables de objeto, estas variables representan la información particular de cada una de las instancias de una clase, como por ejemplo, el nombre de cada ser humano, el color de una silla o el importe total de una factura. Una variable de instancia es exclusiva y particular de cada instancia.

Cuando creamos una clase en Python, lo más común es inicializar los atributos o variables de instancia en el método de inicialización __init__, aunque también podrían crearse variables de instancia en otro métodos de instancia.

Variables de clase

También se llaman atributos de clase o, a veces, variables estáticas (que no tiene nada que ver con constantes, pues su valor se puede modificar). Las variables de clase representan información que es común para todas las instancias de una clase.

De esta manera podemos tener diversas instancias de una clase y todas ellas compartirán los valores de las variables de clase. Si una instancia modifica el valor de una variable de clase, dicho valor queda modificado para todas las instancias.

Así, cuando necesitamos compartir un valor común para todas las instancias de una clase definiremos una variable de clase. La manera más sencilla de definir una variable de clase es hacerlo dentro de la clase pero fuera de las funciones.

dog.py > ...

```python
class Dog:

    animal_type = "Mammal" #class variable

    def __init__(self, name, age, color):
        """Initialize attributes."""
        #instance variable
        self.name = name
        self.age = age
        self._color = color
```

Herencia

No siempre es necesario empezar desde cero al escribir una clase. Si la clase que estás escribiendo es una versión especializada de otra clase que escribiste, puedes usar herencia . Cuando una clase hereda de otra, adquiere los atributos y métodos de la primera clase. La clase original se llama clase padre y la nueva clase es clase hija. La clase hija hereda todos los atributos y métodos de su clase padre, pero también es libre de definir nuevos atributos y métodos propios.

El método __init__() para una sub clase

Cuando escribes una nueva clase y extiendes otra clase existente, a menudo querrás llamar al método __init__() de la clase padre. Esto inicializará todos los atributos que se definieron en el método __init__() y los hará disponibles en la clase hija.

Para ello en el método __init__() de la clase hija escribimos la siguiente línea: super().__init__()

```
13    class MiSubClase(MiClase):
14        def __init__(self):
15            super().__init__()
```

En el ejemplo anterior tenemos una clase MiSubClase que extiende la MiClase (hereda todos los miembros). Luego al crear un objeto de MiSubClase, se ejecuta ambos métodos __init__( ).

La función super() es una función especial que le permite llamar a un método de otra clase superior en la jerarquía. El nombre super proviene de una convención de llamar a la clase principal superclase y a la clase secundaria subclase.

car.py > ...

```
1   class Car:
2
3       def __init__(self, model: str, year: int):
4           self._model = model
5           self._year = year
6
7       @property
8       def model(self):
9           return self._model
10      @model.setter
11      def model(self, new_model):
12          self._model = new_model
13
14      @property
15      def year(self):
16          return self._year
17      @year.setter
18      def year(self, new_year):
19          self._year = new_year

21  class ElectricCar(Car):
22
23      def __init__(self, autonomy_km: float, year: int, model: str):
24          self.autonomy_km = autonomy_km
25          super().__init__(model, year)
```

```
28  my_car = ElectricCar(250.6,2022,'Tito')
29  print(my_car.model)
```

```
PROBLEMS 6  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\valon\Desktop\PythonEjercicios-master\oop> python car.py
Tito
```

Polimorfismo

El polimorfismo es uno de los pilares básicos en la programación orientada a objetos, por lo que para entenderlo es importante tener las bases de la POO y la herencia bien asentadas.

En programación orientada a objetos, el polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

Definimos las clases Gato y Perro que extienden la clase Animal. Todas implementan el método hablar( )

```
1   class Animal:
2
3       def __init__(self) -> None:
4           pass
5
6       def hablar(self) -> str:
7           pass
8
9   class Perro(Animal):
10
11      def __init__(self) -> None:
12          pass
13      def hablar(self) -> str:
14          return "Guau!"
15
16  class Gato(Animal):
17
18      def __init__(self) -> None:
19          pass
20
21      def hablar(self) -> str:
22          return "Miau!"
```

A continuación creamos un objeto de cada clase y llamamos al método hablar(). Podemos observar que cada animal se comporta de manera distinta al usar hablar().

```
25    pet = Perro()
26    print(pet.hablar())
27    pet = Gato()
28    print(pet.hablar())
```

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\valon\Desktop\PythonEjercicic
Guau!
Miau!
```

En el caso anterior, la variable pet ha ido "tomando las formas" de Perro y Gato, pero en cualquier caso al llamar al método hablar se ejecuta el comportamiento correspondiente.

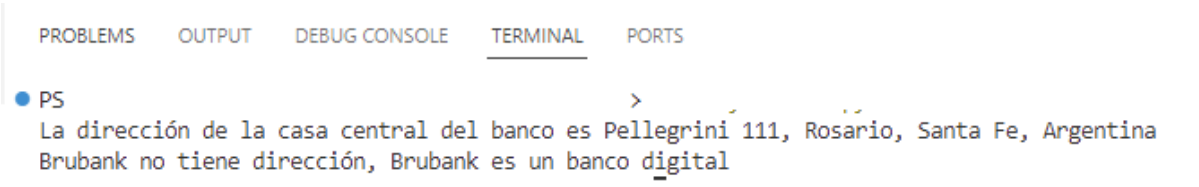
El concepto de polimorfismo, desde una perspectiva más general, se puede aplicar tanto a funciones como a tipos de datos. Así nacen los conceptos de funciones polimórficas y tipos polimórficos. Las primeras son aquellas funciones que pueden evaluarse o ser aplicadas a diferentes tipos de datos de forma indistinta; los tipos polimórficos, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

El concepto de polimorfismo podemos aplicarlo para sobrecargar o sobrescribir métodos.

Sobreescritura de métodos

Puede anular o sobreescribir la implementación de cualquier método de la clase padre que no se ajuste a lo que intenta modelar con la clase hija. Para hacer esto, define un método en la clase hija con el mismo nombre que el método que desea anular en la clase padre. Python ignorará la implementación del método de la clase padre y solo prestará atención a la implementación redefinida en la clase hija.

```
1  class Banco():
2      ...
3
4      def get_address(self) -> str:
5          return "La dirección de la casa central del banco es " + self.__address
6
7      ...
8
9  class Brubank():
10     ...
11
12     def get_address(self) -> str:
13         return "Brubank no tiene dirección, Brubank es un banco digital"
14
15     ...
```



Sobrecarga de métodos

En Python la sobrecarga de métodos (overloading) como tal, no existe. Quienes vienen de otros lenguajes como Java o C# se encuentran con algunas confusiones puesto que es algo muy común en estos lenguajes. La sobrecarga de métodos o comúnmente llamada Overloading es un práctica que consiste en tener diferentes métodos con el mismo nombre en una misma clase, y que el intérprete o compilador logre diferenciarlos por los tipos de datos que se envían como argumentos para los parámetros en la llamada al método.

Herencia Múltiple

Herencia múltiple hace referencia a la característica de los lenguajes de programación orientada a objetos en la que una clase puede heredar comportamientos y características de más de una superclase. Esto contrasta con la herencia simple, donde una clase sólo puede heredar de una superclase. Python soporta la herencia múltiple pero no es una buena práctica en programación y hay pocos casos dónde deba ser implementada.

Clase Abstracta

Un concepto importante en programación orientada a objetos es el de las clases abstractas. Unas clases en las que se pueden definir tanto métodos como propiedades, pero que no pueden ser instancias directamente. Solamente se pueden usar para construir subclases. Permitiendo así tener una única implementación de los métodos compartidos, evitando la duplicación de código.

Otra característica de estas clases es que no es necesario que tengan una implementación de todos los métodos. Pudiendo ser estos abstractos. Los métodos abstractos son aquellos que solamente tienen una declaración, pero no una implementación.

Las clases derivadas de las clases abstractas deben implementar necesariamente todos los métodos abstractos para poder crear una clase que se ajuste a la interfaz definida. En el caso de que no se defina alguno de los métodos no se podrá crear la clase.

Resumiendo, las clases abstractas definen una interfaz común para las subclases. Proporciona atributos y métodos comunes para todas las subclases evitando así la necesidad de duplicar código. Imponiendo además los métodos que deben ser implementados para evitar inconsistencias entre las subclases.

Para poder crear clases abstractas en Python es necesario importar la clase ABC y el decorador abstractmethod del módulo abc (Abstract Base Classes). Un módulo que se encuentra en la librería estándar del lenguaje, por lo que no es necesario instalar. Así para definir una clase privada solamente se tiene que crear una clase heredada de ABC con un método abstracto.

```
1   from abc import ABC, abstractmethod
2
3   class Animal(ABC):
4       @abstractmethod
5       def mover(self) -> None:
6           pass
7
8   class Gato(Animal):
9       def mover(self) -> str:
10          return 'Mover gato'
11
12  my_cat = Gato()
```

Ahora si se intenta crear una instancia de la clase animal, Python no lo permitirá indicando que no es posible. Es importante notar que que si la clase no hereda de ABC y contiene por lo menos un método abstracto, Python permitirá instancias las clases.

```
    a = Animal()
        ^^^^^^^^
TypeError: Can't instantiate abstract class Animal with abstract method mover
```

Las subclases tienen que implementar todos los métodos abstractos, en el caso de que falta alguno de ellos Python no permitirá instancias tampoco la clase hija.

Características

- **Una clase abstracta puede tener un constructor**

Las clases abstractas pueden tener constructores, pero no se pueden crear instancias de ellas directamente. Los constructores se utilizan cuando se crea una subclase concreta.
Definirías un constructor en una clase abstracta si deseas realizar alguna inicialización (en los campos de la clase abstracta) antes de que realmente tenga lugar la creación de instancias de una de las subclases.

- **Una clase abstracta puede tener campos de instancia y de clase**
- **Una clase abstracta puede tener métodos concretos y métodos abstractos**

Los métodos abstractos tiene el decorador @abstractmethod, el mismo debe situarse inmediatamente arriba del la definición del método.
Los métodos concretos, con implementación, no llevan el decorador @abstactmethod.

- **Una clase abstracta puede tener propiedades abstractas o concretas**
- **Una clase abstracta no se puede instanciar**
- **Al heredar de una clase abstracta se debe escribir la implementación de los métodos abstractos de la superclase.**
- **Se dice que una subclase sobreescribe un método de su superclase cuando define un método con las mismas características (nombre y parámetros) pero con distinta implementación.**

Nota: Sobreescritura, override o anular es el mismo concepto con diferentes nombres.

Cuidado: Si se define una clase abstracta en Python heredando de ABC pero no se define dentro ningún método abstracto la misma se puede instanciar. Si bien Python permite realizar esta acción no es correcto hacerlo.

raise NotImplementedError()

Esta excepción se deriva de RuntimeError.
En las clases base definidas por el usuario, los métodos abstractos deben generar esta excepción cuando requieren que las clases derivadas anulen el método.
También se pueden utilizar en el proceso de desarrollo mientras la clase que se está desarrollando tiene un método que aún es necesario agregar la implementación real.

```
@abstractmethod
def un_metodo(self):
    raise NotImplementedError
```

```
1  from abc import ABC, abstractmethod
2
3  class Persona(ABC):
4
5      # Constructor de la clase abstracta necesario para inciializar el campo nombre
6      def __init__(self, nombre: str) -> None:
7          self._nombre = nombre #campo protegido
8
9      @property #Propiedad concreta, puede sobreescribirse en la clase hija
10     def nombre(self) -> str:
11         return self._nombre.title()
12
13     @nombre.setter #Propiedad abstracta, debe sobreescribirse en la clase hija
14     @abstractmethod
15     def nombre(self, nuevo_nombre: str):
16         self._nombre = nuevo_nombre
17
18     @abstractmethod #Metodo abstracto, debe sobreescribirse en la clase hija
19     def __str__(self) -> str:
20         raise NotImplementedError
21
22     def mensaje(self) -> str: #Metodo concreto, puede sobreescribirse en la clase hija
23         return "Esto es una persona y su nombre es " + self.nombre
```

Mapeo de relaciones desde el UML a Python

Durante las ejercitaciones veremos cómo pasar de un diagrama de clases UML a código Python.

Modelado de Aplicaciones: UML - Diagrama de clase

El Lenguaje Unificado de Modelado (UML, Unified Model Language), es el lenguaje estándar de modelado para desarrollo de sistemas y de software. UML se ha convertido de facto en el estándar para modelado de aplicaciones software y ha crecido su popularidad en el modelado de otros dominios. Tiene una gran aplicación en la representación y modelado de la información que se utiliza en las fases de análisis y diseño. En diseño de sistemas, se modela por una importante razón: gestionar la complejidad. Un modelo es una abstracción de cosas reales.

Cuando se modela un sistema, se realiza una abstracción ignorando los detalles que sean irrelevantes. El modelo es una simplificación del sistema real. Con un lenguaje firmal de modelado, el lenguaje es abstracto aunque tan preciso como un lenguaje de programación. Esta precisión permite que un lenguaje sea legible por la máquina, de modo que pueda ser interpretado, ejecutado y transformado entre sistemas.

Para modelar un sistema de modo eficiente, se necesita una cosa muy importante: un lenguaje que pueda describir el modelo. ¿Qué es UML? UML es un lenguaje. Esto significa que tiene tanto sintaxis como semántica y se compone de: pseudocódigo, código real, dibujos, programas, descripciones... Los elementos que constituyen un lenguaje de modelado se denominan notación.

El bloque básico de construcción de UML es un diagrama. Existen tipos diferentes,

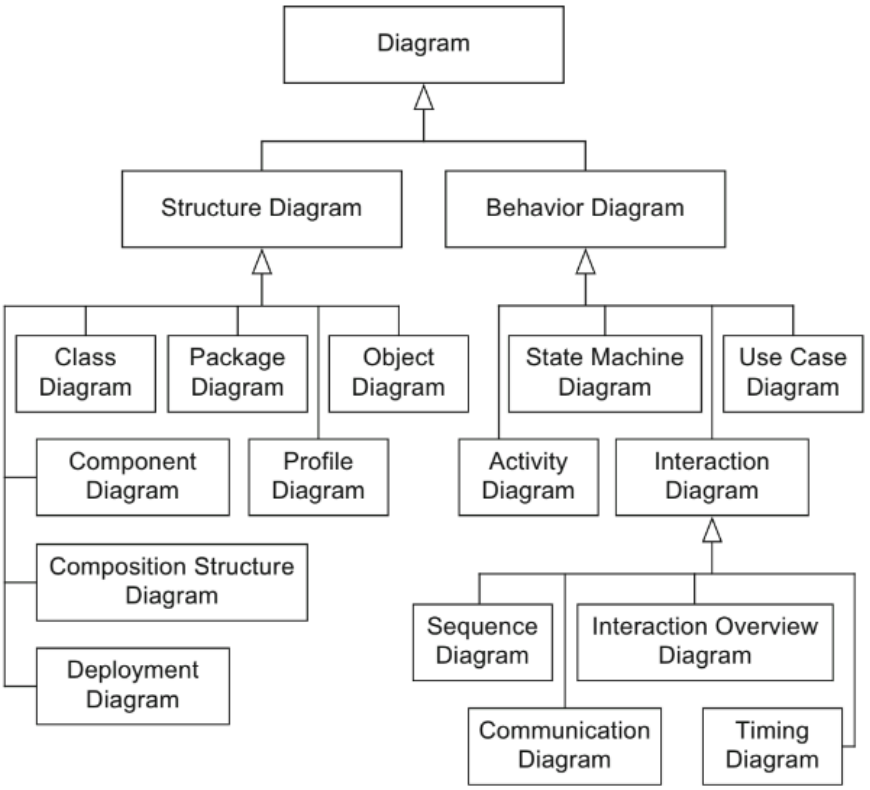


Figure 2.1
UML diagrams

Diagramas de estructura

UML ofrece siete tipos de diagramas para modelar la estructura de un sistema desde diferentes perspectivas. En estos diagramas no se considera el comportamiento dinámico de los elementos en cuestión (es decir, sus cambios en el tiempo).

Diagrama de clases

Usamos el diagrama de clases para modelar la estructura estática de un sistema, por lo que el diagrama de clases describe los elementos del sistema y las relaciones entre ellos. Estos elementos y las relaciones entre ellos no cambian con el tiempo.

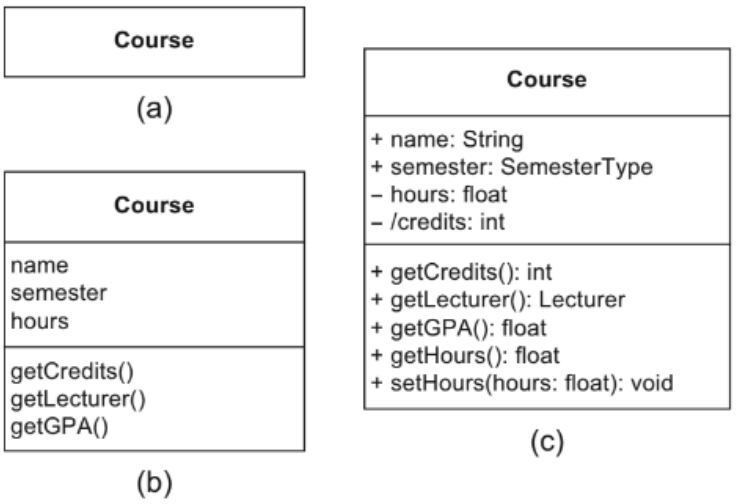
Por ejemplo, los estudiantes tienen un nombre y un número de legajo y cursan varias materias. Esta sentencia cubre una pequeña parte de la estructura universitaria y no pierde validez ni siquiera con el paso de los años. Son sólo los estudiantes los que cambian.

El diagrama de clases es sin duda el diagrama UML más utilizado. Se aplica en varias fases del proceso de desarrollo de software. El nivel de detalle o abstracción del diagrama de clases es diferente en cada fase. En las primeras fases del proyecto, un diagrama de clases le permite crear una vista conceptual del sistema y definir el vocabulario que se utilizará. Luego puede refinar este vocabulario en un lenguaje de programación hasta el punto de implementación. En el contexto de la programación orientada a objetos, el diagrama de clases visualiza las clases que componen un sistema de software y las relaciones entre estas clases. Debido a su simplicidad y popularidad, el diagrama de clases es ideal para bocetos rápidos. Sin embargo, también puedes usarlo para generar código de programa automáticamente. En la práctica, el diagrama de clases también se utiliza a menudo con fines de documentación.

Nomenclatura del diagrama de clases

En un diagrama de clases, una clase está representada por un rectángulo que se puede subdividir en varios compartimentos. El primer compartimento debe contener el nombre de la clase, que generalmente comienza con una letra mayúscula y se coloca centrado en negrita. Según las convenciones de nomenclatura comunes, los nombres de clases son sustantivos singulares.

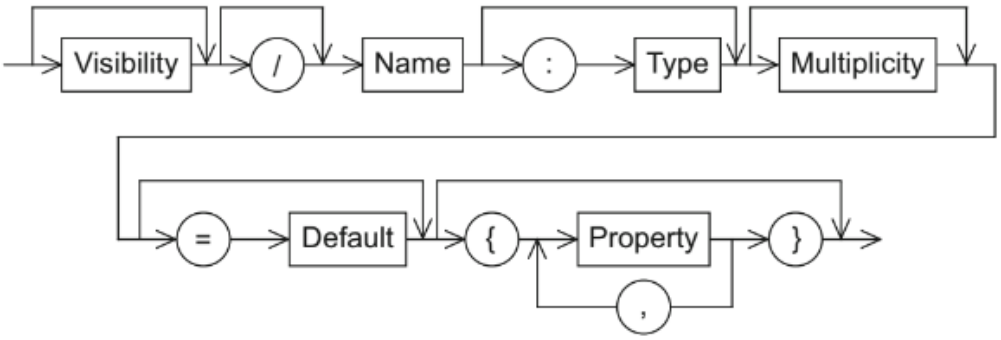
El segundo compartimento del rectángulo contiene los atributos de la clase, y el tercer compartimento contiene los métodos.



Si no se incluye información específica en el diagrama, esto no significa que no exista; simplemente significa que esta información no es relevante en este momento o no se incluye por razones prácticas, por ejemplo, para evitar que el diagrama se vuelva demasiado complicado.

Atributos

Un atributo tiene al menos un nombre. Y su sintaxis de definición es:



La especificación de una barra diagonal **/** antes del nombre de un atributo indica que el valor de este atributo se deriva de otros atributos. Un ejemplo de atributo derivado es la edad de una persona, que se puede calcular a partir de la fecha de nacimiento.

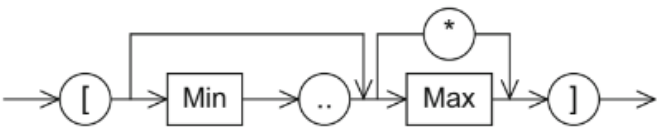
El tipo de atributo se puede especificar después del nombre usando **: Tipo**. Los posibles tipos de atributos incluyen tipos de datos primitivos del lenguaje.

Para definir un valor predeterminado para un atributo, especifique **= default** , donde el valor predeterminado es un valor o expresión definido por el usuario. El sistema utiliza el valor predeterminado si el valor del atributo no lo establece explícitamente el usuario.

Puede especificar propiedades adicionales del atributo entre corchetes. Por ejemplo, la propiedad **{readOnly}** significa que el valor del atributo no se puede cambiar una vez que se ha inicializado.

Multiplicidad

La multiplicidad de un atributo indica cuántos valores puede contener un atributo. Esto Generalmente se codifica como una tipo iterable dependiendo los tipos del lenguaje de programación.



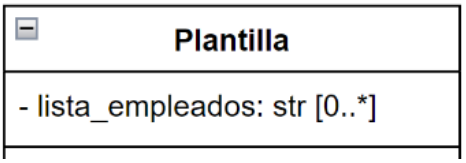
La multiplicidad se muestra como un intervalo encerrado entre corchetes en la forma [mínimo... máximo], donde mínimo y máximo son números naturales que indican los límites superior e inferior del intervalo. El valor del mínimo debe ser menor o igual que el valor del máximo. Si no hay un límite superior para el intervalo, éste se expresa con un asterisco ∗.

Si el mínimo y el máximo son idénticos, no es necesario especificar el mínimo ni los dos puntos; Por ejemplo [5].

La expresión [*] es equivalente a [0..*]. Si no especifica una multiplicidad para un atributo, se supone que el valor 1 es el predeterminado, lo que especifica un atributo de un solo valor.

Si un atributo puede adoptar múltiples valores, se podrá codificar por ejemplo en Python cómo:
Un conjunto (sin orden fijo de elementos, sin duplicados)
Una lista (orden fijo, posibles duplicados)

Puede realizar esta especificación combinando propiedades {non-unique} y {unique}, que definen si se permiten o no duplicados, y {ordered} y {unordered}, que fuerzan un orden fijo de los valores de los atributos.



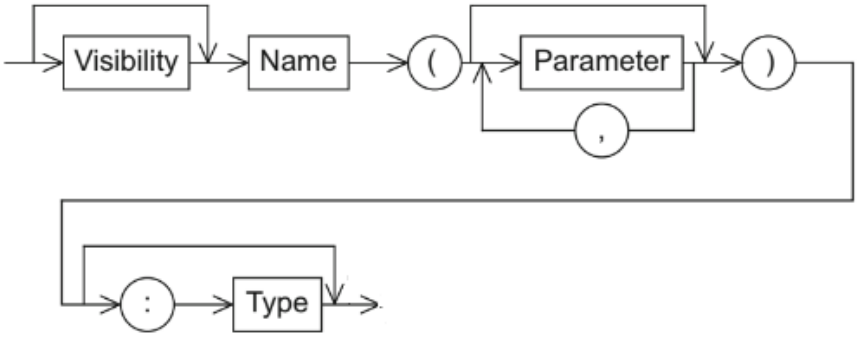
Por ejemplo, tenemos a la clase Plantilla que contiene a un atributo privado lista_empelado que es una colección de strings de 0 a un número indefinido.

Comportamiento

Las operaciones se caracterizan por su nombre, sus parámetros y el tipo de su valor de retorno. Cuando se llama a una operación en un programa, se ejecuta el comportamiento asignado a esta operación. En los lenguajes de programación, una operación corresponde a una declaración de método.

El diagrama de clases no es adecuado para describir el comportamiento de objetos en detalle ya que sólo modela firmas de las operaciones que proporcionan los objetos; no modela cómo se implementan realmente estas operaciones. UML ofrece diagramas de comportamiento especiales para representar la implementación de operaciones, por ejemplo el diagrama de actividades.

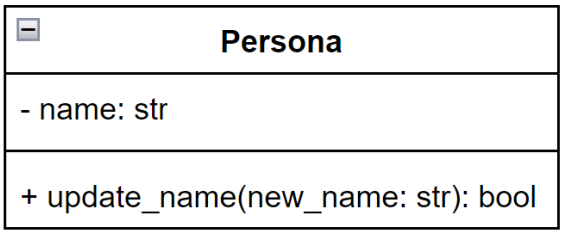
La sintaxis para definir un método en el diagrama de clase es:



En un diagrama de clases, el nombre del método va seguido de una lista de parámetros entre paréntesis. La lista en sí puede estar vacía. Un parámetro se representa de manera similar a un

atributo. La única información obligatoria es el nombre del parámetro. La adición de un tipo y un valor predeterminado son opcionales.

El valor de retorno es opcional y se especifica con el tipo de valor de retorno.



Por ejemplo, el comportamiento definido para la clase Persona, tenemos el método update_name, que recibe cómo único parámetro, new_name de tipo string y especifica el nuevo nombre de una persona. El valor de retorno tiene el tipo booleano. Si se devuelve verdadero, el cambio de nombre se realizó correctamente; de lo contrario, se devuelve falso.

Visibilidad

Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), deben colocarse la anotación antes del nombre de los miembros.

public	**+**	Todos los miembros declarados cómo públicos son de libre acceso desde cualquier otra parte de un programa.
private	**-**	Todos los miembros declarados cómo privados no son accesibles por fuera de la clase.
protected	**#**	Todos los miembros declarados cómo protegidos son accesibles por la clase en que fueran declarados y todas las subclases.

Variables de clase y métodos de clase

Los atributos normalmente se definen a nivel de instancia. Si, por ejemplo, una clase se realiza en un lenguaje de programación, se reserva memoria para cada atributo de un objeto cuando se crea. Estos atributos también se denominan variables de instancia o atributos de instancia.

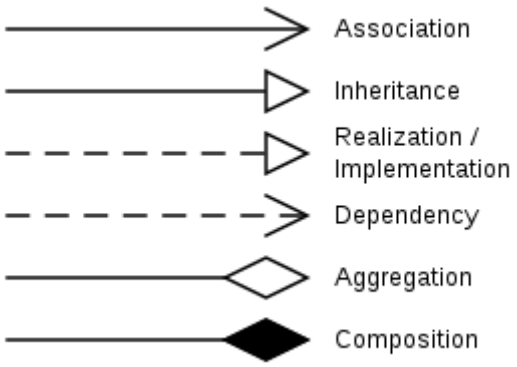
A diferencia de las variables de instancia, las variables de clase se crean sólo una vez para una clase y no por separado para cada instancia de esta clase. Estas variables también se denominan atributos estáticos o atributos de clase.

En el diagrama de clases, los atributos estáticos están subrayados, al igual que los métodos estáticos.

Los métodos estáticos, también llamados métodos de clase, se pueden utilizar si no se creó ninguna instancia de la clase correspondiente. Ejemplos de operaciones estáticas son funciones matemáticas como sin(x) o constructores. Los constructores son funciones especiales llamadas para crear una nueva instancia de una clase.

Relaciones

Las asociaciones entre clases modelan posibles relaciones, conocidas como vínculos, entre instancias de las clases. Describen qué clases son posibles socios de comunicación. Si sus atributos y operaciones tienen las visibilidades correspondientes, los socios de comunicación pueden acceder a los atributos y operaciones de cada uno. En un diagrama de clases puede verse las asociaciones entre las clases.



0	Sin casos (raro)
0..1	Ninguna instancia o una instancia
1	exactamente una instancia
1..1	exactamente una instancia
0..*	Cero o más instancias
*	Cero o más instancias
1..*	Una o más instancias

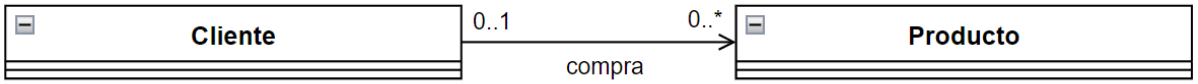
Multiplicidad en las asociaciones entre clases

Asociaciones

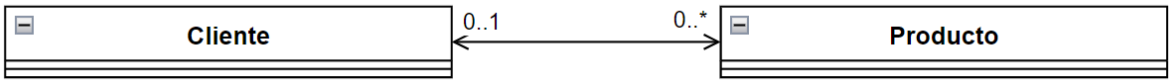
Una asociación binaria nos permite asociar las instancias de dos clases entre sí. Las relaciones se muestran como bordes (línea continua) entre las clases de socios involucradas. El borde se puede etiquetar con el nombre de la asociación seguido opcionalmente de la dirección de lectura, un pequeño triángulo negro. La dirección de lectura está dirigida hacia un extremo de la asociación y simplemente indica en qué dirección el lector del diagrama debe "leer" el nombre de la asociación. Además generalmente se coloca la multiplicidad.

En orientación a objetos, el comportamiento de un sistema se define en términos de interacciones entre objetos, es decir, intercambios de mensajes. "Enviar un mensaje" habitualmente resulta en la invocación de una operación en el receptor. Las asociaciones son

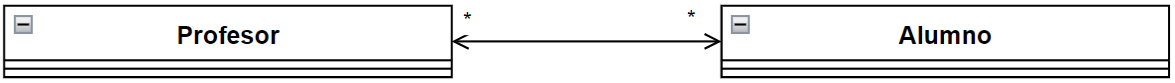
necesarias para la comunicación, ya que los mensajes son enviados a través de las asociaciones; sin asociaciones, los objetos quedarían aislados, incapaces de interactuar.
En los lenguajes de programación orientados a objetos podemos codificar estas relaciones cómo un atributo de tipo objeto de la relación.



En este caso el cliente conoce los productos que compró, por ejemplo en un resumen del pedido. Pero el Producto no conoce el cliente que lo compró.



En este caso el producto conoce al cliente que lo compró y por lo tanto, de una lista de productos vendidos es posible obtener una lista de los clientes que compran más de x cantidad de productos mensuales.



En este caso un profesor accede a todos sus alumnos, y un alumno puede acceder a todos sus profesores.

Dependencia

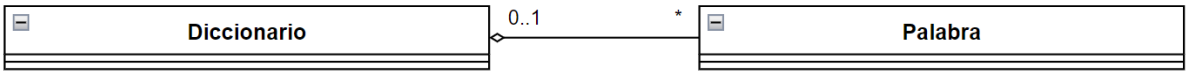
Es una forma de asociación que especifica algún tipo de dependencia entre dos clases, donde un cambio en la clase de la cual se depende puede afectar a la clase dependiente, pero no necesariamente a la inversa. En UML se representa como una asociación pero, en lugar de usar una línea sólida se utiliza una línea punteada. Puede agregarse una flecha para indicar dependencia asimétrica.
En la mayoría de los casos, las dependencias se reflejan en los métodos de una clase que utilizan el objeto de otra clase como parámetro .
Una relación de dependencia es una relación de “uso”. Un cambio en una cosa en particular puede afectar a otras cosas que la usan, y usar una dependencia cuando es necesario indicar que una cosa usa otra.

La clase CronogramaMateria tiene una dependencia con el CalendarioAcademico ya que es pasado cómo parámetro a la llamada del método realizar_cronograma. Un cambio en la clase CalendarioAcademico podría impactar en la implementación del método.

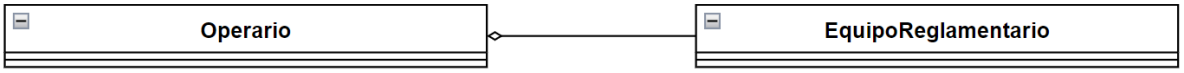
La clase Sueldo depende de la clase Calculadora, ya que para realizar comportamiento cómo por ejemplo calcular el sueldo a abonar, llama al método calcular_ret_ganancias.

Agregaciones

La agregación es un tipo de asociación que indica que una clase es parte de otra clase (composición débil).
Las relaciones agregadas también representan la relación entre el todo y una parte de la clase, los objetos miembros son parte del objeto general, pero el objeto miembro puede existir independientemente del objeto general.
La destrucción del compuesto no conlleva la destrucción de los componentes. Habitualmente se da con mayor frecuencia que la composición.
La agregación se representa en UML mediante un diamante de color blanco colocado en el extremo en el que está la clase que representa el "todo".



Por ejemplo, las palabras son parte de un diccionario, pero existen por fuera de él.



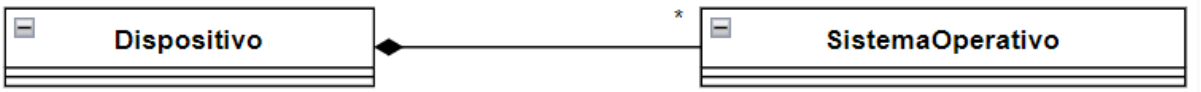
Por ejemplo, un operario y su equipo reglamentario son parte de la relación todo/parte, pero se pueden separar.

Estas relaciones comúnmente se codifican cómo un parámetro pasado en el constructor de una clase. A veces se codifica al igual que las asociaciones.

Composiciones

Composición es una forma fuerte de composición donde la vida de la clase contenida debe coincidir con la vida de la clase contenedora. Los componentes constituyen una parte del objeto compuesto. La supresión del objeto compuesto conlleva la supresión de los componentes.
El símbolo de composición es un diamante de color negro colocado en el extremo en el que está la clase que representa el "todo".

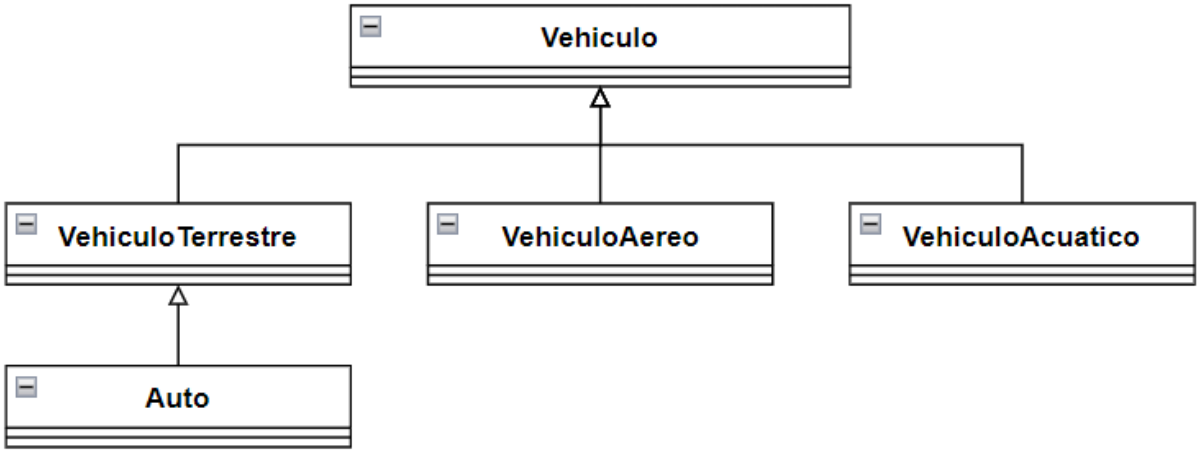
Estas relaciones comúnmente se codifican instanciando al objeto parte dentro de la clase todo, ya sea en el constructor o en algún método.



Imaginemos que una clase Dispositivo está compuesto de un sistema operativo dedicado, que fue diseñado específicamente para el dispositivo. Podemos decir que un celular está compuesto de su SistemaOperativo y que si el Dispositivo desaparece, también desaparecerá el SistemaOperativoDedicado. Entonces, aquí hay una relación de composición.

Herencia

La relación de generalización expresa que las características (atributos y operaciones) y asociaciones que se especifican para una clase general (superclase) se pasan a sus subclases. Por tanto, la relación de generalización también se denomina herencia. Esto significa que cada instancia de una subclase es simultáneamente una instancia indirecta de la superclase. La subclase "posee" todos los atributos de instancia y atributos de clase y todas las operaciones de instancia y operaciones de clase de la superclase siempre que no hayan sido marcadas con visibilidad privada.
La subclase también puede tener otros atributos y operaciones o entrar en otras relaciones independientemente de su superclase. En consecuencia, las operaciones que se originan en la subclase o superclase se pueden ejecutar directamente en la instancia de una subclase.
Una relación de generalización está representada por una flecha con punta triangular desde la subclase a la superclase.

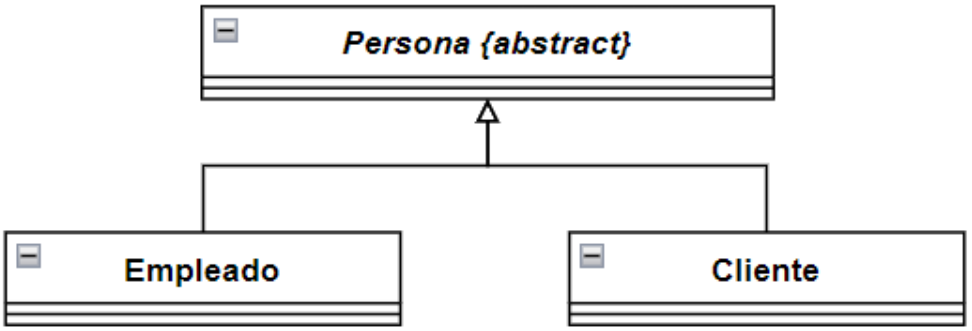


Esto se codifica extendiendo desde la subclase a la clase de la cual se quiere heredar. De esta forma generamos una jerarquía de clases.

Estas son todas las relaciones que vamos a utilizar.

Clases abstractas

Las clases de las que no se pueden crear instancias por sí mismas se modelan como clases abstractas. Estas son clases para las cuales no hay objetos; sólo se pueden crear instancias de sus subclases. Las clases abstractas se utilizan exclusivamente para resaltar características comunes de sus subclases y, por lo tanto, son sólo útiles en el contexto de relaciones de generalización. Las operaciones de clases abstractas también pueden etiquetarse como abstractas. Una operación abstracta La operación abstracta no ofrece ninguna implementación en sí misma. Sin embargo, requiere una implementación en las subclases concretas. Las operaciones que no son abstractas transmiten su comportamiento a todas las subclases.
Las clases abstractas y las operaciones abstractas están escritas en cursiva o indicadas mediante la especificación de la palabra clave {abstract} antes de su nombre. En particular, en los diagramas de clases producidos manualmente, se recomienda el uso de la segunda notación alternativa, ya que la escritura en cursiva es difícil de reconocer.



Supongamos que estamos diseñando un sistema para una empresa, entonces las clases Empleado y Cliente heredan de la clase abstracta Persona. No tiene sentido para el contexto del problema crear un objeto de la clase Persona, ya que se acotó el contexto a las personas que o son clientes o son empleados.

Bibliografía

https://docs.python.org/3/tutorial/classes.html
Seidl, M., Huemer, M. S. C., & Kappel, G. (2012). UML@ Classroom An Introduction to Object-Oriented Modeling.
Dive Into Design Patterns
Matthes, E. (2023). Python crash course: A hands-on, project-based introduction to programming. no starch press.
https://docs.python.org/3/tutorial/classes.html
https://realpython.com/python-pep8/
Matthes, E. (2023). Python crash course: A hands-on, project-based introduction to programming. no starch press.
https://www.freecodecamp.org/news/python-property-decorator

Versiones

Versión	
1.0	Versión Inicial
2.0	Unificación de los apunte de la unidad

Autores

María Mercedes Valoni