

The background of the entire page is an abstract, grayscale wireframe mesh. This mesh forms a complex, undulating shape that resembles a stylized letter 'P' or a series of connected loops. The lines of the mesh are thin and dark, creating a grid-like pattern that follows the contours of the shape. The overall effect is a modern, technical, and artistic aesthetic.

Programación con Python

Programación Orientada a Objetos en Python

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del Copyright, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo públicos. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

INICIATIVA Y COORDINACIÓN

DEUSTO FORMACIÓN

COLABORADORES

Realización:

E-Mafe E-Learning Solutions S.L.

Elaboración de contenidos:

Claudio García Martorell

Licenciado IT Telecomunicaciones especialidad Telemática.

Postgrado en Sistemas de Comunicación y Certificación en Business Intelligence TargIT University.

Concejal de Innovación y Tecnología.

Ponente y docente en distintas universidades y eventos.

Josep Estarlich Pau

Técnico de Ingeniería Informática.

Director Área de Software de la empresa Dismuntel.

Participante en proyectos con Python, C#, R y PHP orientados a *Machine Learning* y a la Inteligencia Artificial.

Supervisión técnica y pedagógica:

Gruñum educación y excelencia

Coordinación editorial:

Gruñum educación y excelencia

© Gruñum educación y excelencia, S.L.

Barcelona (España), 2021

Primera edición: septiembre 2021

ISBN: 978-84-1300-688-8 (Obra completa)

ISBN: 978-84-1300-693-2 (Programación Orientada a Objetos en Python)

Depósito Legal: B 11038-2021

Impreso por:

SERVIFORM

Avenida de los Premios Nobel, 37

Polígono Casablanca

28850 Torrejón de Ardoz (Madrid)

Printed in Spain

Impreso en España

Esquema de contenido

1. Conceptos generales

- 1.1. Programación estructurada vs. Programación Orientada a Objetos
- 1.2. Clases
- 1.3. Objetos

2. Objetos, atributos y herencia

- 2.1. Abstracción
- 2.2. Encapsulación
- 2.3. Herencia
- 2.4. Polimorfismo
- 2.5. Herencia múltiple

3. Métodos de colecciones y métodos especiales

- 3.1. Método constructor “__init__”
- 3.2. Método destructor “__del__”
- 3.3. Métodos matemáticos y lógicos
- 3.4. Métodos de colecciones: cadenas
- 3.5. Listas
- 3.6. Diccionarios

4. Uso de módulos y paquetes: definición y uso

- 4.1. Módulos
- 4.2. Paquetes
- 4.3. Paquetes distribuidos
- 4.4. Módulos estándar

Introducción

En este módulo veremos lo que son las clases, los objetos, la herencia... En definitiva, empezaremos a ver lo que es la Programación Orientada a Objetos (POO) y sus características.

Los ejemplos de código que hemos visto hasta ahora respondían a una programación estructurada y secuencial, pero ahora trabajaremos con otra metodología.

También veremos unas directrices de programación que pueden afectar mucho a nuestra forma de escribir programas, ya que ahorran mucho código y, lo que es más importante, despliegan una serie de funcionalidades más potentes que cualquier elemento visto hasta el momento.

1. Conceptos generales

En esta unidad, vamos a comparar dos paradigmas de programación usados hoy en día en los lenguajes de alto nivel: la **programación estructurada** y la **programación orientada a objetos**. También, las ventajas y desventajas de cada una, así como sus principales diferencias y, lo que es más importante, cuándo es recomendable usar una o la otra, en función del objetivo a conseguir.

1.1. Programación estructurada vs. Programación Orientada a Objetos

Hasta ahora, hemos definido variables, hemos planteado algoritmos y hemos desarrollado código en base a unas reglas y a un orden muy concreto. Pero a partir de ahora se nos abre una nueva metodología de programación con Python, más potente y más profunda, ya que abre una gran cantidad de posibilidades para afrontar el desarrollo de objetos. Hasta ahora, hemos trabajado con programación estructurada, pero a partir de este punto, vamos a profundizar en la programación orientada a objetos.

1.1.1. Programación estructurada

Es un paradigma de programación basado en la utilización de funciones y en tres estructuras de control para la generación de los algoritmos:

- **Estructura secuencial (o lineal):** se ejecuta una orden después de la anterior, siempre siguiendo un orden establecido. Esa secuencia de ejecución (orden#1, orden#2, orden#3, etc.) es la secuencia común de ejecución, ya que, por defecto, el intérprete de Python ejecuta las órdenes en el orden en el que aparecen en el código.
- **Estructura condicional:** se ejecuta una orden (o conjunto de órdenes) según si un valor booleano es verdadero o falso. Este valor booleano puede venir de una comparación aritmética (ej.: "If (7<5):") o booleana (ej.: "If (variable==True):"). Para estas estructuras condicionales, Python contempla los elementos *if*, *elif* y *else*.
- **Estructura iterativa:** ejecución de una orden (o conjunto de órdenes) en base a que una variable booleana sea verdadera o no. Para

esta estructura, Python contempla las estructuras o bucles *while* y *for*.

Al principio, cuando aún no se tenía una orientación a objetos y los algoritmos de decisión tenían muchas limitaciones, los programadores solían utilizar un comando llamado “goto”, que hacía que el flujo del programa saltara o fuese (*go to*, “ir a” en inglés) a una línea o parte del programa en concreto.

Así se forzaban las bifurcaciones o decisiones en los algoritmos, lo que creaba muchos problemas para seguir la trazabilidad y el flujo de datos del programa.

Por ello, se empezó a desarrollar una filosofía de programación más elegante, más concisa y menos confusa a la hora de interpretar el código, así se inició la metodología de programación estructurada.

En el año 1968, el científico de computación Edsger Dijkstra publicó un artículo en el que sostenía que el comando “goto” era perjudicial para la programación, pues su uso era una amenaza para la claridad y la trazabilidad que debía tener la programación estructurada, ya que cuanto más se usaba dentro de un código, menos legible y comprensible era este.

Todo se empezó a normalizar cuando Niklaus Wirth diseñó un nuevo lenguaje de programación (Pascal, lanzado en 1970), que se ha usado desde entonces como paradigma de la programación estructurada, sirviendo como ejemplo a muchos otros.

Esta modalidad de programación o forma de estructurar el código se caracteriza principalmente por que el programador puede partir el código en bloques o módulos con una estructura lógica en cada uno de ellos, así como relacionarlos entre sí con una serie de reglas o condiciones.

El objetivo que se persigue cuando se utiliza la programación estructurada es dotar de más claridad al código a la hora de seguir el flujo de datos, de dotar de más ayudas a la hora de depurar errores y sobre todo a la hora de interpretar la funcionalidad del mismo.

Dijkstra postuló una serie de ideas que hacían referencia a la programación estructurada, a cómo debía ser y a qué debía contener y las plasmó en los siguientes fundamentos:

- Con la programación estructurada se puede compilar cualquier *software* únicamente haciendo uso de tres estructuras de control: **secuencial**, **repetitiva** y de **alternativas**.
- Para la escritura del código, se establece una jerarquía en el mismo: de arriba a abajo o técnica descendente.
- Limitación de los rangos de validez y la visibilidad (y acceso) de algunas variables y de las estructuras de datos.

Por supuesto, la programación estructurada debe ser independiente del lenguaje en el que se implemente ya que no es algo propio del lenguaje; es más como una filosofía, una serie de pautas para la programación... da igual en qué entorno y en qué lenguaje.

Hemos de asegurarnos de que existe una correcta definición de los módulos en los que partimos nuestro *software*, que existe una jerarquía y que el código es claro y fácil de interpretar.

Una de las grandes ventajas que aporta la programación estructurada es que podemos llegar a escribir nuestro *software* en pseudocódigo, independientemente del lenguaje que utilicemos, facilitando así la base por la cual empezar a guiarnos a la hora de programar y que más adelante podremos traducir al lenguaje concreto que se vaya a utilizar, pero de esa manera ya habremos definido módulos, bloques, jerarquías, etc.

A continuación, enumeramos las ventajas y desventajas de la programación estructurada:

- **Ventajas**

- Los programas son más fáciles de interpretar, se pueden leer de forma secuencial (incluso lineal) y no existen los saltos entre líneas (comando "goto").
- La estructura de los programas es clara, ya que las órdenes están más relacionadas entre sí.
- Se pone el foco en la depuración de errores (o *debugging*) haciendo que se vea reforzada la fase de pruebas, por lo que se aporta mucha más calidad en el producto final, así como en la identificación y clasificación de errores.

- Derivado del punto anterior, se reducen los costes de mantenimiento del programa ya que resulta mucho más fácil la modificación y corrección del código.
- Es más fácil acoplar y encajar el *software* en la necesidad del cliente ya que este paradigma aumenta considerablemente la flexibilidad del *software*, no solo a la hora de desarrollarlo sino a las posibilidades de uso que puede aportar al usuario final.
- Es similar al vocabulario de palabras y símbolos en inglés (sentando las bases de lo que son los lenguajes de alto nivel, cercanos al lenguaje humano).
- Se facilita la gestión de cambios en el programa ante requisitos.
- La orientación de este paradigma es hacia la resolución del problema, hacia la virtualización de la solución y no hacia la optimización de recursos de la máquina (*hardware*).
- La filosofía es independiente de lenguaje, independiente del lenguaje, la plataforma y la máquina, por lo que se puede traducir y ejecutar en cualquier máquina (Figura 1.1).



Figura 1.1

Para el programador resulta más sencillo desarrollar bajo este paradigma, lo que resulta en un aumento de la velocidad de desarrollo del *software*.

• Desventajas

- La programación estructurada está orientada a lenguajes de alto nivel, por lo que el compilador o traductor de estos tiene que traducirla a lenguaje máquina, con el descenso de rendimiento que ello conlleva.

- De la misma forma, un código de lenguaje de alto nivel puede resultar menos eficiente para con la máquina en la que se está ejecutando si se compara con lenguaje máquina o ensamblador.
- El código, aunque modularizado, puede repetirse en varias partes del programa. Es decir, una variable puede tomar diferentes valores en diferentes partes del código, dando pie a posibles errores a la hora de asignar valores o ser llamada.
- Si el programa es muy grande, se puede consumir mucho tiempo haciendo cambios a la hora de actualizar los tipos de datos, ya que cuando hay una modificación en estos, se debe de actualizar el código en todas las ubicaciones en las que se llame o se use ese tipo de datos, implicando que tenga que hacerse por varios programadores si es que cada uno está trabajando en una parte del código.

1.1.2. Programación Orientada a Objetos

El otro paradigma de programación es la **Programación Orientada a Objetos (POO)**, que se centra más en el contenido, en la forma de obtener la información y en los datos que en la estructura en sí (aunque tiene una bien definida). Los objetos, que son abstracciones del lenguaje de programación, contienen los datos clasificados de una manera determinada (cada objeto tendrá unas características diferentes para almacenar y gestionar los datos) y eso provocará que la salida y los resultados obtenidos sean diferentes a su vez también. Como ya hemos visto anteriormente, los objetos son “contenedores de datos” pero con una estructura muy concreta.

Algunos de los lenguajes de programación de alto nivel de hoy en día permiten que algunos de los objetos que ya llevan incorporados se puedan agrupar en librerías o bibliotecas; asimismo, también permiten al usuario crear objetos y agruparlos creando sus propias bibliotecas.

Este paradigma se basa en varias técnicas de programación para la generación de los objetos y aunque veremos más en profundidad todas estas características más adelante, es conveniente enumerarlas y definir las resumidamente:

- **Abstracción:** característica por la que debemos centrarnos en qué ha de hacer un objeto más que en la parte técnica de cómo lo hace, realizando una “abstracción” de la funcionalidad de ese mismo objeto en el mundo real.



Certificación

Es importante interiorizar la POO, pues en numerosas preguntas de la certificación se hace referencia a esta.

- **Modularidad:** característica por la que podemos dividir un programa en varias partes más pequeñas, independientes y autocontenidas. Además, han de ser reutilizables si fuera necesario. A estos trozos de código “modulares” se les llama **módulos**.
- **Encapsulación:** característica por la cual se realiza una ocultación de la información del objeto (de sus datos), de manera que solo se pueda acceder a variables o partes funcionales de ese módulo mediante funciones y operaciones especialmente definidas para ese objeto.
- **Herencia:** es la propiedad por la cual una clase puede “heredar” (de ahí el nombre) características de una clase superior para poder definir objetos similares a partir de esta última. Se heredan atributos y métodos, aunque estos últimos se puedan sobrescribir (los que no estén protegidos) para adaptarlos a los requisitos de la nueva clase.
- **Polimorfismo:** posibilidad de identificar de forma similar comportamientos similares asociados a objetos diferentes. La idea principal es que se sigan siempre las mismas directrices, aunque los objetos y sus resultados sean diferentes.

La POO se empezó a usar en los años 60, concretamente a partir del lenguaje de programación **Simula**, que fue el primer lenguaje que hizo una aproximación a lo que después se terminaría llamando “objetos”; si bien es cierto que fue en la década de los 70 con Smalltalk cuando se desarrolló mejor la teoría y las bases de la Programación Orientada a Objetos. Pese a todo lo anterior, la POO no se popularizó hasta entrada la década de los 90.

En Python, los objetos pueden tener una cantidad no fija de datos; es decir, no hay una limitación en cuanto a la cantidad ni, por supuesto, en cuanto a la forma en la que los objetos se definen.

Algunas de las características propias de los objetos en Python son:

- Todo es un objeto, incluyendo los tipos y clases.
- Se permite herencia múltiple (herencia de diferentes clases).
- No existe el concepto de elemento “privado” (como en Java), aplicable a métodos y a atributos.
- Los atributos pueden ser modificados directamente.

- Permite *monkey patching* y el *duck typing*.
- Permite la sobrecarga de operadores.
- Permite la creación de nuevos tipos de datos.

A continuación, enumeramos las principales ventajas y desventajas de este tipo de programación:

- **Ventajas**

- **Reusabilidad:** si las clases están bien definidas, se podrá reutilizar y llamar a ese código en cualquier parte del programa, ahorrándonos el trabajo de volver a definir clases.
- **Mantenimiento:** gracias a la abstracción del problema, el código desarrollado con POO es más sencillo de leer e interpretar, ya que podemos ocultar características e información del objeto dejando a la vista lo más relevante o lo que realmente se usa, haciendo así el código mucho más claro y más fácil de mantener.
- **Modificación:** la posibilidad de crear, modificar o borrar objetos nos permite una operativa muy flexible con los objetos a la hora de cambiar alguno de sus parámetros.
- **Fiabilidad:** al tener el código modularizado, se aplica involuntariamente el principio de “divide y vencerás” para atajar un problema, por lo que al poder probar pequeñas partes de código y aislar al máximo los posibles errores, aumenta la calidad del *software*, así como la fiabilidad de funcionamiento del algoritmo.

- **Desventajas**

- El concepto de objeto y, más concretamente, de abstracción que tiene un programador puede diferir del concepto de abstracción que pueda tener otro programador, por lo que el objeto puede definirse de forma diferente por uno y otro, y es necesaria una gran cantidad de documentación al respecto para unificar criterios.
- Se requiere un esfuerzo extra por parte del programador, pues modelar una solución de forma abstracta no siempre es rápido o fácil.
- La ejecución de programas orientados a objetos es más lenta.

1.2. Clases

Para la creación de los objetos, se necesitan las **clases**. Es el paso previo. A partir de una clase determinada se pueden crear varios objetos que contengan las características de esta. Las clases serán las que nos permitan realizar una abstracción de los datos y su operativa. Podríamos representarlo de la siguiente manera (Figura 1.2):

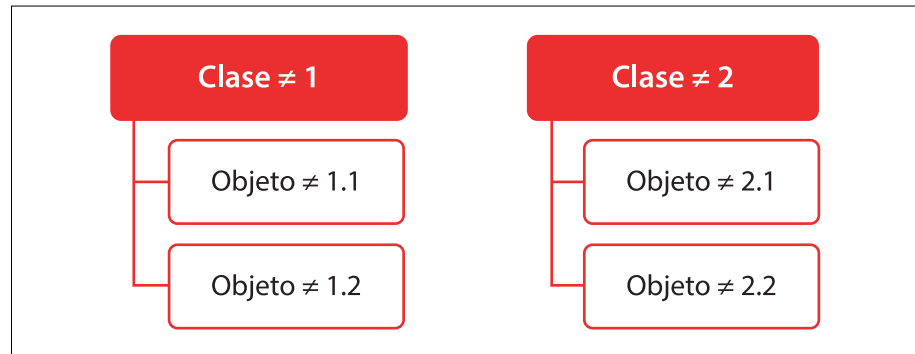


Figura 1.2
Representación gráfica para definir la relación entre clase y objeto.

Una clase se compone de una serie de **métodos** y **datos** (o atributos). Como ya hemos dicho antes, se pueden generar objetos a partir de una misma clase; por hacer un símil (y para practicar la abstracción), podríamos decir que la clase es un molde de pastel y los objetos son los pasteles. El molde (la forma) del pastel en esencia será el mismo, pero el tipo de pasteles podría variar en cuanto a color, sabor o ingredientes.

Siguiendo con la analogía gráfica anterior, podríamos aumentar el detalle del diagrama (Figura 1.3):

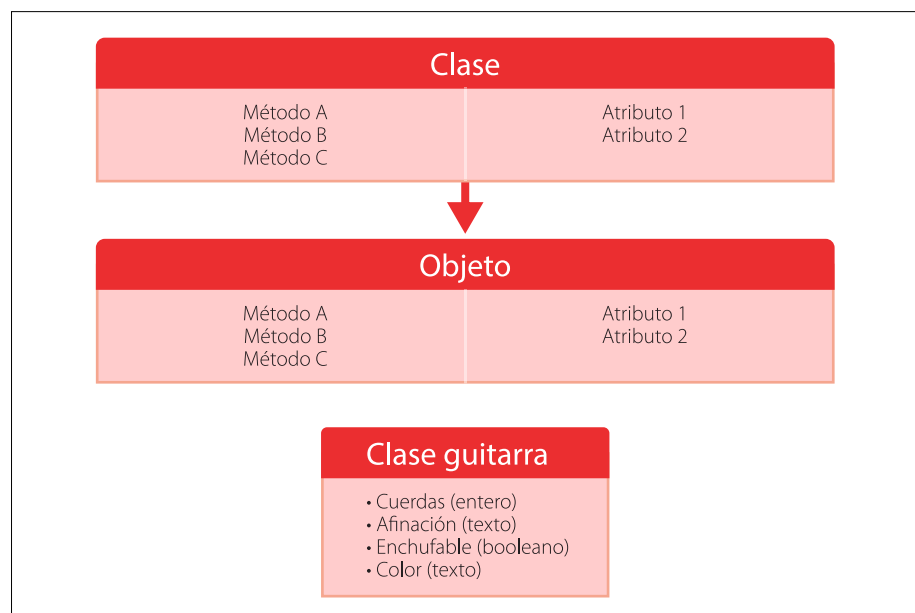
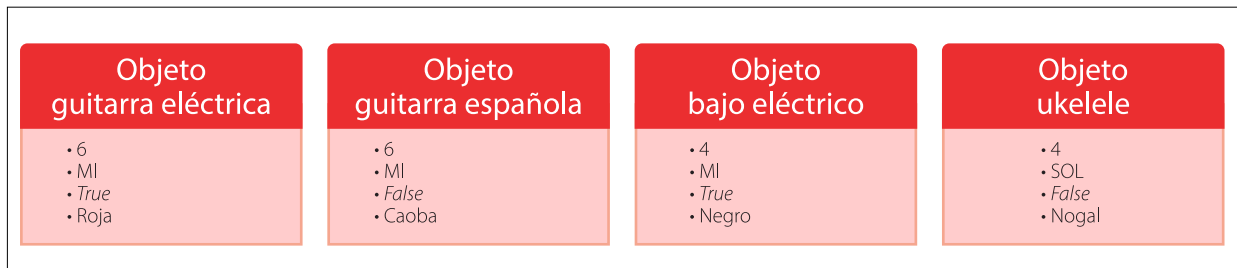


Figura 1.3
Representación gráfica para definir la relación entre clase y objeto incluyendo atributos.

Por darle un sentido más práctico al ejemplo anterior (y por no caer en el ejemplo de las galletas o los pasteles), podríamos generar una representación como la que se expone a continuación (Figura 1.4):

Figura 1.4

Representación de los atributos de un objeto.



Y los objetos que se pueden crear son (Figura 1.5):

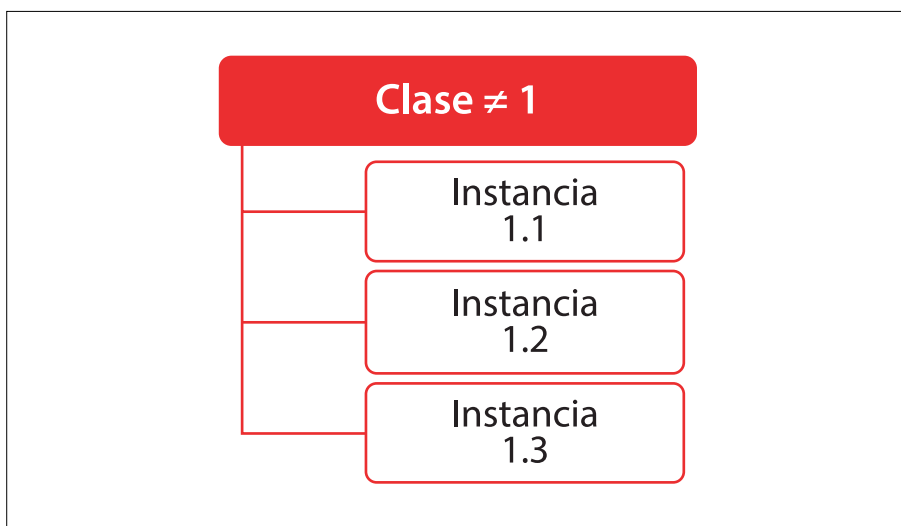


Figura 1.5

Representación de las instancias de una clase.

Cada vez que generamos un objeto a partir de una clase, podemos decir que hemos creado una **instancia** de esa clase. Y obviamente, pueden coexistir varios objetos activos de la misma clase durante el ciclo de vida del programa.

Respecto a las **variables**, existen dos tipos que podemos usar:

- **Variable de clase:** es compartida por todas las instancias (u objetos) de una clase. Se definen dentro de la clase, pero nunca dentro de un método (más adelante veremos las limitaciones de los ámbitos). Este tipo de variables no se utilizan con tanta frecuencia como las variables de instancia.
- **Variable de instancia:** se define dentro de un método (o función) y es propio del objeto (o instancia, de ahí su nombre).

Una clase consta de dos partes:

- **Encabezado:** comienza con el término *class* seguido del nombre de la clase (en singular) y dos puntos (":").
- **Cuerpo:** en él definimos los atributos (variables de clase) y los métodos y la documentación de la propia clase.

```
class Guitarra:
    #CABECERA
    """Clase para definir objetos del tipo guitarra. Los
    valores de las variables están por defecto"""

    #CUERPO

    cuerdas = 6
    eléctrica = False
    color = "Blanco"
    afinación = "MI"

    def conocerAfinacion(self, afinacion):
        return ("La guitarra está afinada en ", afinación)

    def conocerCuerdas(self, cuerdas):
        return ("La guitarra tiene ", cuerdas, "cuerdas")
```

La documentación que queramos añadirle a la clase deber ir al principio de la cabecera, antes de la definición de cualquier método o variable, ya que en esa documentación explicaremos el propósito de la clase, qué variables contiene y qué valores pueden tomar, así como una explicación de los métodos.

Así, pues, desde cualquier punto del programa podremos consultar la documentación de esa clase poniendo la instrucción:

NombreClase.__doc__ que en nuestro caso sería **Guitarra.__doc__**

A la hora de definir una clase, no es obligatorio completarla con un número de variables o métodos determinado, es más, podemos definir la clase tan solo nombrándola, sin incluir ningún tipo de información adicional. Esto se haría con el comando "pass"; y aunque este comando no ejecuta nada, simplemente reserva el sitio de código por si en un futuro queremos desarrollarlo y a la vez, permite que se instancien objetos de esa clase (vacía).


```
class Amplificador:
    pass
```

Más allá de eso, incluso con una clase vacía se pueden realizar algunas operaciones:

```
class Amplificador:
    pass

ampli1 = Amplificador()
# Crea el objeto ampli1 de clase Amplificador

ampli2 = Amplificador()
# Crea el objeto ampli2 de clase Amplificador

print(ampli1 == ampli2)
```

Este último comando retornará un valor “False”, ya que, aunque los objetos se han creado a partir de la misma clase y no se han hecho modificaciones en ellos, son objetos distintos.

Por otra parte, no es estrictamente necesario que creamos las variables de clase nada más definir la clase; podemos crear la clase vacía (solo con la instrucción “pass”) y llamarla junto con un atributo que, si no existe, se creará en ese momento.

```
Class Amplificador:
    Pass
#Creamos la clase sin ningún tipo de atributo ni variable de clase

ampli1 = Amplificador ()
# Crea el objeto ampli1 de clase Amplificador
ampli2 = Amplificador ()
# Crea el objeto ampli2 de clase Amplificador

#Asignamos un valor a los atributos .marca y .
color que NO están definidos
Amplificador.marca = “Marshall”
Amplificador.color = “Negro”

#Llamamos al atributo .color
print(Amplificador.color)
```

El resultado por pantalla será “Negro”, que es el valor que le hemos asignado al atributo “Amplificador.color”.

En el siguiente ejemplo se define una clase mucho más completa:

```
class Clubdelectura:
    Clase para un club de lectura, donde cada miembro aportará una cantidad de libros'
    cantidadMiembros = 0
    cantidadLibros = 0

    def __init__(self, nombre, libros):
        self.nombre = nombre
        self.Libros = libros
        Clubdelectura.cantidadMiembros +=1
        Clubdelectura.cantidadLibros += libros

    def mostrarMiembro(self):
        Return (self.nombre, self.libros)

    def mostrarCantidadLibros(self):
        return("Ahora mismo hay ",
        Clubdelectura.cantidadMiembros, " películas en el videoclub")

    def mostrarLibrosTotales(self):
        return(Clubdelectura.cantidadLibros)

    def mostrarMediaLibros(self):
        if Clubdelectura.cantidadLibros > 0:
            return(Clubdelectura.cantidadLibros/Clubdelectura.cantidadMiembros)
        else:
            return("Sin libros")
```

La clase "Clubdelectura" tiene dos variables de clase, "cantidadLibros" y "cantidadMiembros", a las que podremos acceder mediante "Clubdelectura.cantidadLibros" y "Clubdelectura.cantidadMiembros" respectivamente.

Después de la declaración de las variables, vemos que se declaran una serie de métodos, comenzando por el constructor de clase "__init__()" que ya explicaremos más adelante en profundidad, pero aquí sirve para inicializar las variables "Clubdelectura.cantidadLibros" y "Clubdelectura.cantidadMiembros".

Un elemento común de los métodos creados, es que incluyen como primer argumento "self". Esto es una referencia al propio objeto, es decir, al objeto en concreto que llama al método. Como el valor de "self" ya está implícito, no es necesario escribir este argumento cuando se llame al método en cuestión.

Los otros métodos (o funciones) sirven para mostrar el valor de las variables de clase y hacer cálculos con ellas.

1.3. Objetos

Como ya hemos podido deducir del punto anterior, si bien la clase es la parte que define las reglas del juego con los objetos, con lo que realmente vamos a trabajar, modificar, asignar valores y extraer información es con los objetos.

Su nombre viene de la abstracción que le damos a un conjunto de funcionalidades del propio objeto, asemejándolo con el comportamiento que pudiera tener ese mismo objeto, pero en la vida real. El objeto virtual a semejanza del objeto físico tiene estados, valores y puede realizar una serie de acciones.

Queda claro que cuando hablamos de un objeto, estamos hablando de un elemento dinámico, que cambia, que toma valores que pueden variar, que interactúa con el usuario y sus acciones, mientras que una clase es la estructura rígida a partir de la que se crean los objetos. Volviendo al ejemplo más usado en el mundo de la POO, la clase es el molde y el objeto es la galleta.

Como ya hemos dicho en otras ocasiones, en Python todo son objetos, pero la gran diferencia entre un tipo de datos (que viene a ser un objeto con unas características muy concretas) y un objeto instanciado a partir de una clase desarrollada por el usuario es justamente eso: que el segundo objeto lo define el programador, no es un tipo básico.

Así pues, volviendo al ejemplo del punto anterior, la clase a partir de la cual se van a construir (o instanciar) objetos es “Guitarra” (la hemos llamado así para abreviar, por similitudes). Hay varios instrumentos del mundo real que podrían incluirse dentro de la familia de las guitarras: los bajos, los ukeleles, las guitarras clásicas, las acústicas, las eléctricas, pero cada una tiene una serie de peculiaridades, que se dan en torno a una serie de elementos comunes. Los atributos de la clase son esos elementos comunes, que en nuestro caso eran el número de cuerdas, la afinación, el color y si era enchufable.

Para saber el tipo de un objeto, utilizamos la función “Type ()”, porque nos puede dar información concreta del tipo de objeto con el que estamos trabajando.



Certificación

Es importante interiorizar los objetos, pues en numerosas preguntas de la certificación se hace referencia a estos.

Ya hemos visto anteriormente que podemos probar la función *type* en algunos objetos o tipos de datos en Python.

```
type(42)
#int

type("Hola mundo")
#str

type(True)
#bool

type([])
#list

type(3.14)
#float

type({})
#dict
```

¿Y qué pasa si lo aplicamos a un objeto? Volviendo a nuestro ejemplo del amplificador:

```
class Amplificador:
    pass
#Creamos la clase sin ningún tipo de atributo ni variable de clase

ampli1 = Amplificador()
# Crea el objeto ampli1 de clase Amplificador
ampli2 = Amplificador()
# Crea el objeto ampli2 de clase Amplificador

#Asignamos un valor a los atributos .marca y .
color que NO
están definidos
Amplificador.marca = "Marshall"
Amplificador.color = "Negro"

type(ampli1)
# Nos da como resultado __main__.Amplificador
```

Y también existe la posibilidad de crear un objeto de objetos; es decir, que la información contenida dentro de un objeto sea una colección de objetos (del tipo lista, por ejemplo). Aunque parezca algo complejo, es bastante sencillo de definir. Volviendo a nuestro ejemplo de la clase "Guitarra" y atendiendo a los números de línea de las instrucciones:

```

01: class Guitarra:
02:
03: # Constructor de clase
04: def __init__(self, nombre, cuerdas, afinacion, color, enchufable):
05:     self.nombre = nombre
06:     self.cuerdas= cuerdas
07:     self.afinacion = afinacion
08:     self.color = color
09:     self.enchufable = enchufable
10:     print('Se ha creado el instrumento:', self.nombre)
11:
12: #Redefinimos el método str de la clase
13: def __str__(self):
14:     return '{} {}'.format(self.nombre, self.color)
15:
16:
17: class Coleccion:
18:
19: guitarras = []# Esta lista contendrá objetos de la clase Guitarra
20:
21: #El constructor toma el objeto self y una lista vacía
22: def __init__(self, guitarras=[]):
23:     self.guitarras = guitarras
24:
25: def agregar(self, g): # g será un objeto Guitarra
26:     self.guitarras.append(g) # g se añadirá a la lista
27:
28: def mostrar(self):
29:     for g in self.guitarras:
30:         print(g) # Print toma por defecto el método str(g)
31:
32:
33:     g = Guitarra ("Washburn",6, "mi", "Negro", True)
34: c = Colección([g]) #Añado una lista con una guitarra a través del constructor de clase
35:     c.mostrar()
36:     c.agregar(Guitarra("Epiphone", 6, "mi", "Ocre", True))
37:     c.mostrar()

```

- **Línea 01:** se crea la clase de un objeto "Guitarra".
- **Línea 12:** se redefine el método "por defecto" de todos los objetos de Python para sacar por pantalla la información. Se verá más adelante en profundidad.
- **Línea 16:** se crea el objeto "Colección", que será una lista de las guitarras que tenemos, por lo que almacenaremos objetos "Guitarra" en su interior.

- **Línea 26:** el comando “append” añade un elemento a la lista. En este caso, está añadiendo un objeto a la lista de objetos.
- **Línea 33:** creamos un objeto del tipo “Guitarra” a través de su constructor de clase.
- **Línea 34:** se crea un objeto “Colección” con un objeto “Guitarra” (el creado en la línea 33).
- **Línea 36:** se crea un objeto “Guitarra” con su constructor, pero desde el objeto contenedor “Colección”.



- La programación estructurada se basa en una secuencia de código con bucles de iteración, condicionales y comparaciones entre valores.
- La Programación Orientada a Objetos (POO) se centra en la creación de elementos en base a una serie de atributos y métodos definidos en una clase para poder trabajar con estructuras independientes.
- Las clases son el molde, las especificaciones de lo que debe contener el objeto. Se le pueden definir métodos (funciones) y atributos (variables).
- Los objetos son elementos con los que el usuario puede interactuar y mediante los que podemos diseñar cómo se organiza la información, tanto la que mostramos como la que nos puede entrar.

2. Objetos, atributos y herencia

En esta unidad, veremos las principales características de la Programación Orientada a Objetos y qué cualidades o destrezas debemos entrenar y adquirir para poder desarrollar con este paradigma de programación no solo a nivel sintáctico o gramatical, sino también desde un punto de vista filosófico o de concepto. Para ello, tenemos que aprender a abstraer conceptos y darles forma o saber qué partes de los objetos queremos mostrar y cuáles debemos proteger.

2.1. Abstracción

En la POO, ya hemos visto que uno de los conceptos que hay que tener muy claro es el de objeto; es decir, saber qué son, de dónde vienen, qué se puede hacer con ellos, etc. También la **abstracción** es otro de los conceptos de obligado conocimiento para poder programar en orientación a objetos.

Como ya hemos visto, la abstracción hace referencia a cómo se debe usar algo (un objeto, en este caso), sin atender a términos técnicos ni profundizar en las bases. Simplemente, hay que pensar en las funcionalidades últimas de ese objeto e intentar ofrecer funciones de alto nivel, pero sencillas de usar, para que el usuario pueda interactuar de forma más fluida con la aplicación. Es lo que se conoce como el principio de enfoque de caja negra.

Este enfoque es muy usado en la capa superior de toda aplicación, incluso en desarrollo web para la orientación de la interfaz de usuario. Sin embargo, para ofrecer una funcionalidad compleja de forma clara, hay que ser consciente de todo lo que se debe mover, consultar y calcular.

Una analogía del mundo real sería la de un cajero automático. Cuando vamos a sacar dinero o a consultar nuestra cuenta, lo hacemos de una forma rápida y precisa, ya que tenemos un panel con botones que nos lleva a esas funciones en concreto. Todo lo que hay por detrás nos da igual, ya que las consultas que se hacen a las bases de datos, los cables y estructuras del cajero, etc. tienen una información cuyo *background* no es importante para el usuario. Este es un ejemplo muy concreto de abstracción, un panel que nos permite:



Certificación

Es importante interiorizar la abstracción, pues en numerosas preguntas de la certificación se hace referencia a esta.

- Sacar dinero
- Ingresar dinero
- Consultar el estado de la cuenta

Algo similar sucede en la Programación Orientada a Objetos; siguiendo con el caso anterior, si tuviésemos una clase que se llamase “Cajero”, un buen ejercicio de abstracción sería que ofreciese (al menos) tres métodos para que el usuario interactuase con ellos:

- sacarDinero()
- ingresarDinero()
- consultarCuenta()

Si llevamos a la práctica y concretamente al código este concepto de abstracción, en Python existen las clases y los métodos abstractos. Las primeras son las que contienen métodos abstractos, que son aquellos que se han declarado, pero no se han implementado.

Entre las propiedades de las clases abstractas se encuentra la imposibilidad de ser instanciadas. Por otro lado, no todos los métodos deben estar implementados, porque los abstractos son los que solo están declarados, pero no tienen ninguna funcionalidad desarrollada.

Las clases que se deriven de las clases abstractas sí que, obligatoriamente, han de implementar todos los métodos para crear una clase que se ajuste a la interfaz definida. Si queda algún método sin implementar, será imposible crear la clase.

Para la creación de clases abstractas en Python necesitamos importar la clase “ABC” y el decorador “abstractmethod” del módulo abc (*Abstract Base Classes*). Este módulo se encuentra en la librería estándar del lenguaje, por lo que no es necesario que se instale.

```
from abc import ABC, abstractmethod
class Felino(ABC):
    @abstractmethod
    def maullar(self):
    pass
```

Si intentamos crear una instancia de la clase “Felino”, no va a ser posible puesto que el método abstracto “maullar” no está definido.

```
class Felino(ABC):
    def maullar(self):
        pass

    # Felino() Esta instrucción dará error

class Felino():
    @abstractmethod
    def maullar(self):
        pass

Felino()
```

En este último caso no daría error, ya que no hereda de ABC.

Las subclases (las clases que heredan de una superior) también deben implementar todos los métodos abstractos, ya que como se ha dicho antes, en caso de que falte alguno por implementar, Python no deja instanciar.

```
class Felino(ABC):
    def maullar(self):
        pass

    @abstractmethod
    def dormir(self):
        print('Zzzz... zzz')

class Gato(Felino):
    def maullar(self):
        print('Miaaaaau!')

g = Gato() # Error
```

Arreglando el código anterior para evitar el error, tenemos:

```
class Felino(ABC):
    def maullar(self):
        pass

    @abstractmethod
    def dormir(self):
```

```

print('Zzzz... zzz')

class Gato(Felino):
    def maullar(self):
        print('Miaaaau!')

    def dormir(self):
        print('El gato está durmiendo')

Manolo = Gato()
Manolo.maullar()
Manolo.dormir()

```

2.2. Encapsulación

En la POO existe un concepto llamado **encapsulación** que consiste en ocultar o, mejor dicho, no permitir el acceso a algunos métodos o variables de la clase desde el exterior. Es decir, que, con el fin de protegerlos frente a modificaciones, desde fuera no se puede acceder a los atributos y funciones y únicamente se puede hacer mediante el propio objeto.

Este concepto es muy utilizado en lenguajes como Java, pero en Python no se permite, ya que trata a todos los elementos como públicos y no oculta ni métodos ni atributos al exterior. Por lo tanto, el concepto de encapsulación directamente no debería existir, aunque es cierto que en los casos en que es estrictamente necesario, se puede simular esta característica.

```

class GrupoMusical:
    miembrosDefault = 4
    def __init__(self, miembros):
        self.miembros = miembros

miGrupo = GrupoMusical(3)
miGrupo.miembrosDefault #Saca por pantalla 4
miGrupo.miembros #Saca por pantalla 3

```

Como vemos, ambos atributos son accesibles, tanto el atributo de clase “miembrosDefault” como el atributo de instancia “miembros”. Pero podría no ser el comportamiento deseado y, entonces, tendríamos la necesidad de proteger algún atributo al que solo la clase u objeto pudieran acceder para modificar.

Para eso, a la hora de nombrarlos, los precederemos por un doble guion bajo “__”. Esto le dirá a Python que los interprete como privados y no serán accesibles desde el exterior.

```
class GrupoMusical:
    miembrosDefault = 4      #atributo de clase público
    __presupuesto = 8000     #atributo de clase privado

    # Constructor de clase
    def __init__(self, miembros):
        self.miembros = miembros

    # Método privado, no accesible desde el exterior
    def __mostrarPresupuesto(self):
        if(self.__presupuesto<15000):
            print("Somos pobres")

    # Método público, accesible desde el exterior
    def accederMostrarPresupuesto(self):
        # El método sí es accesible desde el interior
        self.__mostrarPresupuesto()

miGrupo = GrupoMusical(3)

miGrupo.__presupuesto
#Generará un error

miGrupo.__mostrarPresupuesto
#Generará un error

miGrupo.miembrosDefault
#Saca por pantalla '4'

miGrupo.accederMostrarPresupuesto()
#Saca por pantalla 'Somos pobres'
```

Por otra parte, podemos hacer uso del comando “dir” para ver el listado de funciones y variables de nuestra clase.

```
print(dir(miGrupo))
```

Lo que obtenemos por pantalla entonces es:

```
[ '_GrupoMusical__mostrarPresupuesto', '_GrupoMusical__presupuesto',
  '__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
  '__eq__', '__format__', '__ge__', '__getattr__', '__gt__',
  '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
  '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
  '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
  '__weakref__', 'accederMostrarPresupuesto', 'miembros',
  'miembrosDefault']
```

Podemos ver que no se encuentran en la lista ni el atributo privado “__presupuesto” ni el método privado “__mostrarPresupuesto”.

A pesar de las dificultades que pone Python a la hora de acceder a una variable o método que interpreta como privado, sí podemos acceder a ellos, haciendo un poco de trampa. No es que Python los elimine del acceso, sino que cambia la nomenclatura para acceder, por lo que, con la sintaxis correcta, podemos acceder a los atributos que hemos definido como privados:

```
print (miGrupo._GrupoMusical__presupuesto)
```

2.3. Herencia

El concepto de **herencia** es uno de los más importantes en la Programación Orientada a Objetos. Nos permite crear una jerarquía a partir de una clase padre o superclase de la que derivarán las clases hijas o subclases.

Las clases hijas heredarán las características con las que se haya definido a la clase padre, es decir, con los atributos y métodos que se hayan definido en la clase padre. Para crear una clase hija, únicamente tendremos que pasar como atributo el nombre de la clase padre en la definición de la clase hija.

```
# Clase padre (superclase)
class Animal:
    pass

# Clase hija (subclase)
class Mapache(Animal):
    pass
```

En este caso, hemos creado la clase padre “Anima”l y hemos creado una clase hija que hereda de ella, llamada “Mapache”, ya que el mapache es un tipo de animal.

Como vemos, tanto la clase padre como la hija no tienen código, así que vamos a implementarlo para ver cómo se comportan las clases hijas al heredar las funcionalidades del padre.

```
#Clase padre
class Pantalon:
    def __init__(self, talla, color, coste, largo):
        self.talla = talla
        self.color = color
        self.coste = coste
        self.largo = largo

    # Este método se define pero no se implementa, porque tendrá
    # características propias, ya que se multiplicará
    # el coste por un coeficiente propio de cada objeto
    def definirPrecio(self):
        # Método vacío
        pass

    # Este método sí se implementa ya que no depende de ninguna
    # consideración particular o propia de cada objeto, sino que
    # es una consideración general
    def temporada(self):
        if(self.largo == True):
            print ("Otoño & Invierno")
        else:
            print ("Primavera&Verano")

class Jeans(Pantalon):
    def definirPrecio(self):
        print("El precio será: ", self.coste * 1.40)

class Shorts(Pantalon):
    def definirPrecio(self):
        print("El precio será: ", self.coste * 1.15)

miPantalon1 = Jeans("M", "Azul", 120, True)
miPantalon2 = Jeans("M", "Negro", 100, True)
miPantalon3 = Shorts("L", "Estampado", 50, False )

miPantalon1.definirPrecio()
#Sacaré por pantalla "El precio será: 168.0"
miPantalon3.temporada()
#Sacaré por pantalla "Primavera&Verano"
```

Como podemos ver en el ejemplo anterior, la clase padre tiene dos métodos, “definirPrecio” y “temporada”. El primer método no está implementado, porque preferimos que cada clase hija lo implemente con sus particularidades, ya que en este caso y dependiendo del tipo de pantalón, su coste se multiplicará por una constante u otra para obtener el PVP.

Por el contrario, el segundo método sí está definido, ya que atiende a un criterio general en base a la longitud del pantalón (que nos viene definida como parámetro en el constructor cuando se crea el objeto). Por lo tanto, no tiene sentido reescribirlo, ya que sería repetir el mismo código que en la clase padre y no estaríamos ahorrando ni código, ni tiempo a la hora de trabajar con objetos.

2.4. Polimorfismo

Una vez entendido el concepto de herencia en Programación Orientada a Objetos, podemos definir otro de los pilares básicos de este paradigma de programación como es el **polimorfismo**.

Como su nombre indica, el polimorfismo aplicado a un objeto hace que este tome diferentes formas. El concepto es bastante abstracto por sí mismo, pero significa que se puede acceder a objetos de diferentes clases, utilizando una misma interfaz o método, comportándose de forma distinta o cambiando de forma si son accedidos.

El polimorfismo está íntimamente ligado al *duck typing* en lenguajes que, como Python, tienen tipado dinámico. No es necesario que los objetos compartan una interfaz, basta con que tengan el mismo método al que queremos llamar.

Si utilizamos un ejemplo muy concreto, supongamos que tenemos una clase padre llamada “Instrumento”, cuyo único método es el de “Afinación” (para determinar en qué nota de la escala está afinado ese instrumento, independientemente del que sea).

```
class Instrumento:
    def afinacion(self):
        pass
```

Creamos dos clases que heredan de “Instrumento”: “Guitarra” y “Ukelele”. Tanto una como la otra, implementan el método afinación de forma distinta.

```
class Guitarra(Instrumento):
    def afinacion(self):
        print("Mi Mayor")

class Ukelele(Instrumento):
    def afinacion(self):
        print("Sol Mayor")
```

Así pues, creamos un objeto de cada clase hija y llamamos al método “afinación()”, pero desde la clase padre, observando que los resultados van a ser distintos.

```
for Instrumento in Guitarra(), Ukelele():
    Instrumento.afinacion()
```

```
# Mi Mayor
# Sol Mayor
```

En el caso anterior, la variable “Instrumento” ha ido cambiando su forma, amoldándose a “Guitarra” y luego a “Ukelele”.

2.5. Herencia múltiple

La **herencia múltiple** es una característica que muy pocos lenguajes tienen y que Python incorpora. Se da cuando una clase hija hereda de dos (o más) clases padre. Esto hace que la clase hija incorpore todos los atributos y métodos de las clases padre.

El problema surge cuando hay métodos o variables que tienen el mismo nombre en las distintas clases padre. En estos casos, Python prioriza a la clase que más a la izquierda se encuentra en la declaración de herencia, desestimando los valores del resto de las clases. Hay que recordar que esta criba o priorización se da solo en los atributos o métodos que coinciden en nombre, ya que a los que no coinciden los incorpora directamente.

En el siguiente ejemplo tenemos una clase padre “León”, otra clase padre “Tigre” y una clase hija que heredará de ambas y que se llama “Ligre” (sí, ese animal existe en la realidad y proviene de la unión de un león y una tigresa).

Hay métodos propios del “León”, como “Velocidad”, que la clase “Ligre” heredará sin problemas, de igual manera que podrá heredar el método

“Alimentación” de la clase “Tigre”. Además, se ha definido un método propio del “Ligre” que es “Longitud”. Atención especial al caso de “Identificación”, puesto que es un método común a “León” y “Tigre”, pero atendiendo a la definición de la clase “Ligre”: los valores que prevalecen serán los de la clase “León”, ya que está más a la izquierda.

```
#Clase padre nº1
class Leon:
    #El método identificacion() es común a león y tigre
    def identificacion(self):
        print("Soy un león")

    #El método velocidad() es propio del león
    def Velocidad(self):
        print("Puedo correr a 80km/h")

#Clase Padre nº2
class Tigre:

    #El método identificacion() es común a león y tigre
    def identificacion(self):
        print("Soy un Tigre")

    #El método alimentacion() es propio del tigre
    def Alimentacion(self):
        print("Soy omnívoro")

#Clase que hereda de ambas
class Ligre(Leon,Tigre):

    #El método longitud() es el único propio del ligre
    def Longitud(self):
        print("Mido 3.5m")

#Creamos una instancia
ligre = Ligre()

#Método heredado y común a León y Tigre
ligre.identificacion() # Saca por pantalla "Soy un león"

#Método heredado de León
ligre.Velocidad()      # Saca por pantalla "Puedo correr a 80km/h"

#Método heredado de Tigre
ligre.Alimentacion()   # Saca por pantalla "Soy omnívoro"

#Método propio de Ligre
ligre.Longitud()       # Saca por pantalla "Mido 3.5m"
```



Resumen

- La abstracción se basa en tomar un objeto del mundo real e imaginar sus funcionalidades y la forma en que el usuario interactúa con ellas, para llevar después toda esa información al desarrollo de objetos a partir de clases.
- La encapsulación consiste en saber dónde podemos proteger como privados los elementos del código de nuestro objeto para ocultarlos al usuario.
- La herencia es la creación de clases hijas a partir de clases padre de las que heredan los métodos y los atributos que, si bien pueden funcionar de igual manera, también se pueden reimplementar para una mayor eficacia del código.
- El polimorfismo consiste en hacer que una clase se comporte de diferente forma, tomando valores de objetos distintos, pero con variables iguales, otorgando así al usuario flexibilidad a la hora de tratar los datos en los objetos.
- La herencia múltiple se basa en que una clase hija pueda heredar métodos y atributos de varias clases padre.

3. Métodos de colecciones y métodos especiales

Dos de los métodos especiales más importantes en Python son el **constructor** y el **destructor**. Puede ser interesante ver que podemos prescindir de ambos y que nuestro programa funcionaría igual. Sin embargo, el saber cómo editarlos y modificarlos según nuestros intereses hará que el programa se optimice y podremos tener un mayor control sobre lo que ocurre con los objetos, tanto al inicio como al final de la vida del objeto.

3.1. Método constructor “__init__”

Como ya hemos visto en algunos ejemplos anteriores, a veces para empezar a desarrollar las instrucciones dentro de un objeto, el primer método o clase con el que nos encontramos es el método “__init__”. Constructor “__init__”.

Este es un método especial, su objetivo principal es inicializar los atributos del objeto que estamos creando. Es decir, no podemos empezar a operar con los atributos de una instancia si estos no tienen un valor. Una forma de asegurarnos de que lo tienen es haber pasado antes por el inicializador.



Certificación

Es importante interiorizar el método constructor, pues en numerosas preguntas de la certificación se hace referencia a este.

```
class vehículo:

    def mostrarCaracterísticas():
        print ("El vehículo es de color ", self.color)
        print ("El vehículo tiene ", self.ruedas, " ruedas.")
        print ("El vehículo posee ", self.plazas, " plazas disponibles")
```

Al llamar a la función “vehículo.mostrarCaracterísticas()” nos generará el siguiente error:

```
NameError                                Traceback (most recent call last)
<ipython-input-5-8c9a3f351136> in <module>()
      8     print ("El vehículo posee ", self.plazas, " plazas disponibles")
      9
--> 10 vehículo.mostrarCaracterísticas()
```

```

<ipython-input-5-8c9a3f351136> in mostrarCaracterísticas()
      4
      5 def mostrarCaracterísticas():
----> 6     print ("El vehículo es de color ", self.color)
      7     print ("El vehículo tiene ", self.ruedas, " ruedas.")
      8     print ("El vehículo posee ", self.plazas, " plazas disponibles")

NameError: name 'self' is not defined

```

Tratando los datos con un inicializador previo, quedaría así:

```

class Vehículo:

    def __init__(self):
        self.color="Blanco"
        self.ruedas=4
        self.plazas=7

    def mostrarCaracterísticas(self):
        print ("El vehículo es de color ", self.color)
        print ("El vehículo tiene ", self.ruedas, " ruedas.")
        print ("El vehículo posee ", self.plazas, " plazas disponibles")

coche = Vehículo()
coche.mostrarCaracterísticas()

```

Obteniendo por pantalla:

```

El vehículo es de color  Blanco
El vehículo tiene  4  ruedas.
El vehículo posee  7  plazas disponibles

```

Este es un método al que no será necesario llamar, ya que se hará automáticamente sin que nosotros lo hagamos en cuanto sea creado el objeto. Tiene algunas particularidades propias:

- No puede devolver ningún tipo de dato.
- Es un método no obligatorio, es decir, si no lo declaramos, no obtendremos un error, aunque lo común es hacerlo.
- Puede recibir parámetros que sirvan para la inicialización de las variables o atributos de la clase.

Atendiendo a la teoría, al método “__init__” no se le podría considerar solo como un constructor, sino más bien como un inicializador. Su misión, como ya hemos visto, es inicializar variables, pero no las construye, no las crea, por así decirlo.

El verdadero constructor en Python es el método especial “__new()__”. Este método es el encargado de crear el objeto reservándole espacio en memoria, mientras que lo que hace “__init()__” es dar un valor a los atributos (inicializar). Pero no nos hemos de preocupar por llamarlo, porque lo hace automáticamente en segundo plano Python, una vez que creamos un objeto a partir de una clase.

De la misma forma que a “__init()__” se le pasa el parámetro *self* cuando se declara, cuando se ejecuta “__new()__” se le pasa un atributo llamado *cls*, que es la clase que va a construir.

En muy pocas ocasiones, el programador tendrá que usar la función especial “__new(cls)”, pero en cualquier caso podemos decir que la parte constructora de Python consta de la ejecución de dos métodos especiales: “__new(cls)” + “__init(self)”, mientras que en otros lenguajes de programación la construcción recae en una sola función, que fusiona a las anteriores.



Para saber más

A diferencia de otros lenguajes, en los que está permitido implementar más de un constructor, en Python solo se puede definir un método “__init__()”.

3.2. Método destructor “__del__”

Hasta ahora hemos visto que para crear (o instanciar) un objeto creábamos una clase. A partir de allí, podíamos crear clases hijas que heredasen y en última instancia, creábamos objetos que podíamos inicializar con la orden “__init__”.

Sin embargo, hasta el momento, no nos hemos preocupado sobre el ciclo de vida del objeto más allá de crearlo y usarlo. Pero a veces, cuando nuestros objetos se vuelven muy complejos, es necesario implementar alguna acción o algoritmo para que el objeto deje de estar en memoria y se destruya. Para eso usamos el destructor.

Hacemos uso de la instrucción “__del__” para invocar la destrucción del objeto. No se suele reescribir este método, ya que su funcionalidad es la esperada en todos los casos y además se maneja automáticamente. Su uso se justifica para forzar una destrucción del objeto en base a liberar espacio en memoria cuando se trabaja con estructuras y objetos excesivamente complejos. La información que maneja el destructor “__del__”

internamente no es que fulmine inmediatamente al objeto, sino que lo que hace Python es decrementar en uno la cuenta de referencias a ese objeto. Cuando se deja de utilizar y en el momento en el que esa cuenta llega a 0, se llama al método “__del__” y entonces el objeto se da por destruido.

```
class Edificio:

    #Inicializamos los valores y avisamos de cuándo se ha construido el objeto
    def __init__(self):
        self.altura = 0
        print('Se han construido los cimientos')

    #Esta función únicamente suma 1 cuando es llamada
    def Construir(self) :
        self.altura = self.altura + 1
        print('Se ha construido el piso nº ',self.altura)

    #Definimos la función de destrucción para saber cuándo se destruye el objeto
    def __del__(self):
        print('Se ha destruido el edificio de ', self.altura, " alturas")

#Creamos una instancia
rascacielos = Edificio()

#Llamamos a la función de operaciones varias veces para darle un valor al objeto
rascacielos.Construir()
rascacielos.Construir()
rascacielos.Construir()
rascacielos.Construir()

#Al asignar un nuevo valor al objeto, se ha de destruir todo valor anterior
rascacielos = 57

print('Se han construido un total de ',rascacielos, " alturas")
```

Al ejecutar este programa, se produce el siguiente resultado:

```
a)    Se han construido los cimientos
b)    Se ha construido el piso nº  1
c)    Se ha construido el piso nº  2
d)    Se ha construido el piso nº  3
e)    Se ha construido el piso nº  4
f)    Se ha destruido el edificio de  4  alturas
g)    Se han construido un total de  57  alturas
```

En el punto a) se crea el objeto, pasando por el inicializador “__init__” y poniendo el valor de la variable “altura” a 0.

Hasta el punto e) se llama a un método que hace cálculos con la variable.

El punto f) corresponde con la línea en la que se llama a la operación de asignación “rascacielos = 57” porque ahí se le está diciendo “deshecha el objeto que tenías hasta ahora (el que sumaba altura) y toma el siguiente valor (57)”. Por eso, para descartar el objeto, se llama al método destructor, que pone en pantalla el mensaje: “Se ha destruido el edificio de 4 alturas”.

Un objeto se puede destruir de varias maneras, además de la que ya hemos mostrado. Veamos otras alternativas partiendo del mismo ejemplo anterior:

```
class Edificio:

    #Inicializamos los valores y avisamos de cuándo se ha construido el objeto
    def __init__(self):
        self.altura = 0
        print('Se han construido los cimientos')

    #Esta función únicamente suma 1 cuando es llamada
    def Construir(self) :
        self.altura = self.altura + 1
        print('Se ha construido el piso nº ',self.altura)

    #Definimos la función de destrucción para saber cuándo se destruye el objeto
    def __del__(self):
        print('Se ha destruido el edificio de ', self.altura, " alturas")

#Creamos una instancia
rascacielos = Edificio()

#Llamamos a la función de operaciones varias veces para darle un valor al objeto
rascacielos.Construir()
rascacielos.Construir()
rascacielos.Construir()
rascacielos.Construir()

#Al asignar un nuevo valor al objeto, se ha de destruir todo valor anterior
del(rascacielos)

print('Se han construido un total de ',rascacielos.altura, " alturas")
```

Aquí se utiliza la función reservada “del (rascacielos)” que llama directamente al destructor. Más adelante, se intenta volver a llamar al objeto “Rascacielos” para sacar su valor por pantalla, pero obtenemos el siguiente error:

```

Se han construido los cimientos
Se ha construido el piso nº 1
Se ha construido el piso nº 2
Se ha construido el piso nº 3
Se ha construido el piso nº 4
Se ha destruido el edificio de 4 alturas
-----
NameError                                Traceback (most recent call last)
<ipython-input-10-4eb195e3aaa4> in <module>()
    27 del(rascacielos)
    28
--> 29 print('Se han construido un total de ',rascacielos, " alturas")

NameError: name 'rascacielos' is not defined

```

Nos dice que no puede sacar nada por pantalla, porque, obviamente, el objeto no existe, ya que lo acabamos de destruir justo en la línea anterior.

Por otra parte, y pese a lo que acabamos de ver, este tipo de funciones especiales tienen métodos para acceder a ellas, también se pueden llamar de forma directa como si fueran métodos normales pertenecientes al objeto. Esto lo conseguiríamos con el comando “rascacielos.__del__()”.

3.3. Métodos matemáticos y lógicos

La posibilidad de determinar el comportamiento de los operadores básicos no se permite en la mayoría de los lenguajes. A esta técnica se la llama sobrecarga de operadores y, si bien no es algo obligatorio en la Programación Orientada a Objetos, hace que el desarrollo sea más cómodo y el código un poco más elegante.

Las siguientes funciones se encargan de implementar los operadores aritméticos básicos que ya hemos visto anteriormente. Para saber a qué equivalen, podemos decir que cuando ejecutamos una operación simple como “variable1 – variable2”, se estaría llamando al método “x.__sub__(y)”.

- + __add__(self, otro)
- - __sub__(self, otro)
- * __mul__(self, otro)

- / `__div__(self, otro)`
- % `__mod__(self, otro)`
- Divmod `__divmod__(self, otro)`
- ** `__pow__(self, otro, [módulo])`
- << `__lshift__(self, otro)`
- >> `__rshift__(self, otro)`
- & `__and__(self, otro)`
- “ `__xor__(self, otro)`
- | `__or__(self, otro)`

Los siguientes métodos implementan de igual manera las operaciones aritméticas anteriores, pero únicamente en el caso en que el operando de la izquierda no implemente las operaciones básicas.

Por ejemplo, si en la operación (“variable1 & variable2”) el primer elemento no implementase el método “`__and__`”, se llamaría entonces a “variable2.`__rand__`(variable1)”.

- `__radd__(self, otro)`
- `__rsub__(self, otro)`
- `__rmul__(self, otro)`
- `__rdiv__(self, otro)`
- `__rmod__(self, otro)`
- `__rdivmod__(self, otro)`
- `__rpow__(self, otro)`
- `__rlshift__(self, otro)`
- `__rrshift__(self, otro)`

- `__rand__(self, otro)`
- `__rxor__(self, otro)`
- `__ror__(self, otro)`

Para los operadores de asignación extendidos:

- `+=` `__iadd__(self, other)`
- `-=` `__isub__(self, other)`
- `*=` `__imul__(self, other)`
- `/=` `__idiv__(self, other)`
- `//=` `__ifloordiv__(self, other)`
- `%=` `__imod__(self, other)`
- `**=` `__ipow__(self, other[, módulo])`
- `<<=` `__ilshift__(self, other)`
- `>>=` `__irshift__(self, other)`
- `&=` `__iand__(self, other)`
- `^=` `__ixor__(self, other)`
- `|=` `__ior__(self, other)`

Para la implementación de las funciones de positivizar, negativizar, sacar el valor absoluto e invertir.

- Negativo `__neg__(self)`
- Positivo `__pos__(self)`
- Valor abs. `__abs__(self)`
- Inversión `__invert__(self)`

Para las funciones de conversión para los tipos complejo, entero, *long* y de coma flotante:

- `__complex__(self)`
- `__int__(self)`
- `__long__(self)`
- `__float__(self)`

Cuando llamamos a las siguientes funciones internas de visualización, nos mostrarán una cadena del valor en el formato indicado. Por ejemplo, si nuestro objeto es "K", llamando a "K.__oct__" nos mostrará a "K" en formato octal (o base 89).

- `__oct__(self)`
- `__hex__(self)`

3.4. Métodos de colecciones: cadenas

```
#capitalize() - Retorna la cadena con la primera letra en mayúscula:
"me gusta el Rock".capitalize()
#POR PANTALLA: 'Me gusta el Rock'

#upper() - Retorna la cadena con todas las letras en mayúscula:
"Me gusta el Rock".upper()
#POR PANTALLA: 'ME GUSTA EL ROCK'

#lower() - Retorna la cadena con todas las letras en minúscula:
"Me gusta el Rock".lower()
#POR PANTALLA: 'me gusta el rock'

#islower() - Retorna True si el string está todo en minúsculas:
"me gusta el Rock".islower()
#POR PANTALLA: False

#isupper() - Retorna True si la cadena es todo mayúsculas:
"Me gusta el Rock".isupper()
#POR PANTALLA: False

#title() - Retorna la cadena con la primera letra de cada palabra en mayúscula:
"me gusta el Rock".title()
#POR PANTALLA: 'Me Gusta El Rock'
```

```

#count() - Retorna las veces que aparece una secuencia en la cadena original:
"Me gusta el Rock".count('Rock')
#POR PANTALLA: 1

#find() - Retorna el índice donde aparece la secuencia a buscar (-1 si no aparece):
"Me gusta el Rock".find('gusta')
#POR PANTALLA: 4

#rfind() - Retorna el índice donde aparece la secuencia a buscar, en orden inverso:
"El Rock es mejor que el Hard Rock".rfind('Rock')
#POR PANTALLA: 30

#isdigit() - Retorna True si el string que le pasamos es un número (False si no):
c = "42"
c.isdigit()
#POR PANTALLA: True

#isalnum() - Retorna True si el string que le pasamos tiene números y letras:
c = "C3P0"
c.isalnum()
#POR PANTALLA: True

#isalpha() - Retorna True si el string que le pasamos son todo letras:
c = "R2D2"
c.isalpha()
#POR PANTALLA: False

#istitle() - Retorna True si la primera letra de cada palabra está en mayúscula:
"Heavy Rock".istitle()
#POR PANTALLA: True

#isspace() - Retorna True si la cadena está llena de espacios:
" 091 ".isspace()
#POR PANTALLA: False

#startswith() - Retorna True si la cadena empieza con el string que le pasamos:
"Me gusta el Rock".startswith("Rock")
#POR PANTALLA: False

#endswith() - Retorna True si la cadena acaba con una subcadena:
"Me gusta el Rock".endswith('Rock')
#POR PANTALLA: True

#split() - Parte la cadena en subcadenas a partir de los espacios y retorna una lista:
"Rock and Roll".split()[0]
#POR PANTALLA: 'Rock'

#Se puede indicar también el carácter por el cual se puede separar:
"A,mí,me,gusta,la,paella".split(',')
#POR PANTALLA: ['A', 'mí', 'me', 'gusta', 'la', 'paella']

```

```

#join() - Junta todos los caracteres de un string que le pasamos mediante un caracter que le
indiquemos:
",".join("Me gusta el Rock")
#POR PANTALLA: 'M,e, ,g,u,s,t,a, ,e,l, ,R,o,c,k'

" ".join("Hola")
#POR PANTALLA: 'H o l a'

#strip() - Elimina todos los espacios en blanco que haya al principio y final de la cadena:
"  Me gusta el Rock  ".strip()
#POR PANTALLA: 'Me gusta el Rock'

#Se puede indicar también cuál es el carácter que queremos borrar:
"----HOLA---".strip('-')
#POR PANTALLA: 'HOLA'

#replace() - Sustituye una subcadena o carácter dentro de una cadena por otra:
"Me gusta el Rock".replace('o','i')
#POR PANTALLA: 'Me gusta el Rick'

#Se puede indicar un número máximo de veces de sustituciones:
"Me gusta mucho mucho mucho muchísimo la paella".replace(' mucho',' ',3)
#POR PANTALLA: 'Me gusta muchísimo la paella'

```

3.5. Listas

```

#append() - Añade un elemento al final de la lista:
list = [1,2,3,4,5]
list.append(10)
list

#POR PANTALLA: [1, 2, 3, 4, 5, 10]

#clear() - Elimina todos los elementos de la lista:
list.clear()
list

#POR PANTALLA: []

#extend() - Junta dos listas:
l1 = [42,33,99]
l2 = [1,2,3]
l1.extend(l2)
l1

#POR PANTALLA: [42, 33, 99, 1, 2, 3]

#count() - Cuenta las veces que aparece un elemento:

```



Certificación

Es importante interiorizar cadenas y listas, pues en numerosas preguntas de la certificación se hace referencia a estas.

```

["Me", "gusta", "bailar"].count("Me")

#POR PANTALLA: 1

#index() - Retorna la posición en la que aparece un elemento (error si no aparece):
["Hola", "que", "tal"].index("que")

#POR PANTALLA: 1

#insert() - Añade un elemento en una posición en concreto:
#Hay que tener en cuenta que la primera posición es 0 y que la penúltima es -1:
l = [10,20,30]
l.insert(0,0)
l

#POR PANTALLA: [0, 10, 20, 30]

l = [1,2,3,5]
l.insert(-1,4)
l

#POR PANTALLA: [1, 2, 3, 4, 5]

#Podemos calcular la última posición con el método len():
l = [10, 20, 30 ]
n = len(l)
l.insert(n,40)
l

#POR PANTALLA: [10, 20, 30, 40]

#pop() - Saca un elemento de la lista (y lo elimina):
l = [1,2,3,4,5]
print(l.pop())
#POR PANTALLA: 5

#Si volvemos a consultar la lista
print(l)
#POR PANTALLA: [1, 2, 3, 4]

#Se puede indicar qué elemento en concreto queremos sacar mediante su posición (por defecto es el último):
print(l.pop(0))
#POR PANTALLA: 1
print(l)
#POR PANTALLA: [2, 3, 4]

#remove() - Elimina el primer valor de la lista que coincida con el valor que le pasamos:
l = [1, 2, 3, 4, 4, 4, 5]
l.remove(4)
print(l)

```

```

#POR PANTALLA: [1, 2, 3, 4, 4, 5]

#reverse() - Le da la vuelta a la lista actual:
l.reverse()
print(l)

#POR PANTALLA: [5, 4, 4, 3, 2, 1]

#sort() - Ordena los elementos (de menor a mayor según valor):
list = [1, 2, 3, 50, -10]
list.sort()
list

#POR PANTALLA: [-10, 1, 2, 3, 50]

#Admite un argumento para poder indicar el sentido de la ordenación:
list.sort(reverse=True)
list

#POR PANTALLA: [50, 3, 2, 1, -10]

```

3.6. Diccionarios

Muy similares a los de las listas, incluso comparten nombre y en esencia su función es la misma. Partimos de un diccionario muy básico con únicamente 3 elementos:

```

comunidades = { "C.Valenciana":"Valencia", "Andalucía":"Sevilla", "Aragón":"Zaragoza" }

#get() - Busca un elemento a través de la clave que le pasamos. Si no existe, le podemos
definir un mensaje:
comunidades.get('Navarra','no se encuentra')
#POR PANTALLA : 'No existe'

#keys() - Genera un listado de las claves de los elementos contenidos en el diccionario:
comunidades.keys()
#SE GENERA: dict_keys(['C.Valenciana', 'Andalucía', 'Aragón'])

#values() - Genera un listado de los valores de los elementos contenidos en el diccionario:
comunidades.values()
#SE GENERA: dict_values(['Valencia', 'Sevilla', 'Zaragoza'])

#items() - Genera un listado de pares clave-valor de los registros existentes en el diccionario:
comunidades.items()

```

```

#SE GENERA: dict_items([('C.Valenciana', 'Valencia'), ('Andalucía', 'Sevilla'), ('Aragón',
'Zaragoza')])

for clave, valor in comunidades.items():
    print(clave, valor)

#POR PANTALLA :
#C.Valenciana Valencia
#Andalucía Sevilla
#Aragón Zaragoza

#pop() - Muestra un registro a partir de su clave y después lo borra. Si no existe, le pode-
mos definir un mensaje:
comunidades.pop("Andalucía", "no se ha encontrado")

#POR PANTALLA : 'Sevilla'

comunidades.pop("País Vasco", "no se ha encontrado")

#POR PANTALLA : 'No existe'

#clear() - Borra todos los registros del diccionario:
comunidades.clear()
comunidades

#POR PANTALLA : {}

```




Resumen

- El constructor es un método para inicializar las variables de nuestro objeto una vez que vayamos a trabajar con él.
- No es obligatorio desarrollar el código del destructor, porque Python lo llama automáticamente, pero en los casos de objetos muy complejos y/o varias instancias simultáneas en memoria, sí es recomendable para no malgastar recursos.
- Los métodos especiales están por defecto en Python y van desde la forma en la que se muestra un *string* por pantalla hasta métodos matemáticos, que ayudan a hacer cálculos.
- Existen métodos de clase, que son propios de los tipos de objetos que ya hemos visto como listas, diccionarios o cadenas. Son muy útiles para dar formato a la información y poder estructurarla como mejor nos convenga.

4. Uso de módulos y paquetes: definición y uso

Los conceptos de módulos, paquetes y paquetes distribuidos, que veremos aquí, completan todo lo aprendido sobre objetos y nos permitirán llevar el paradigma de la Programación Orientada a Objetos un paso más allá. Vamos a expandir nuestro código y a optimizar varias tareas como la trazabilidad o la depuración de errores, que simplificarán mucho nuestro trabajo. También veremos cómo exportar y llevar con nosotros algún código que sea común a varios de nuestros programas: funciones recurrentes, cálculos repetitivos, constantes, etc.

4.1. Módulos

Cuando hablamos de módulos en Python, nos referimos a un archivo aparte, donde existen atributos, clases, métodos, etc., a los que se puede acceder desde otro archivo.

Esta es una forma muy útil de organizar la información, ya que cuando se trabaja con proyectos complejos conviene tener el código lo más modularizado que sea posible. Ya no solo de cara al mantenimiento del archivo, sino por cuestiones de seguridad:

- El mantenimiento del *software* partido en varios módulos o archivos se simplifica, porque en lugar de abrir un archivo donde esté todo el código de nuestro programa, podemos atacar pequeñas unidades para hacer cambios concretos. Por ejemplo, si identificamos un error en una línea de código y hay que buscarla para modificarla, no es lo mismo modificar una instrucción en un archivo de 1.000 líneas que modificar una línea en un archivo que tenga 10 líneas de código. Esto simplifica mucho las tareas de visualización y comprensión del *software*.
- En cuanto a las cuestiones de seguridad, muchas veces podemos afrontar problemas como sobreescritura de archivos, cambios no guardados, versiones desactualizadas y eso es mejor que nos pase con un archivo de 10 líneas que con uno de 1.000. Además, en el caso de tener los archivos en un servidor, es más lógico separar la parte del negocio de la parte del cliente.

Un módulo no es más que un archivo con extensión .py, que tiene dentro un código Python. Un ejemplo de uso de módulos sería tener un archivo principal, donde se hacen cálculos con variables y, en base a sus resultados, llamar a una serie de funciones. En este caso, sería conveniente tener separados los ficheros de los cálculos con las variables (fichero principal) y otro fichero con toda la relación de funciones posibles. Así, si detectamos un error o queremos hacer una modificación en las funciones, es mucho más fácil de identificar y aislar.

Lo primero que necesitamos tener es, obviamente, al menos dos archivos: uno principal y otro al que llamaremos desde el primero. En este caso, asumimos que ambos ficheros están en la misma carpeta.

- menu.py
- operaciones.py

Desarrollamos el código del fichero operaciones.py, donde definimos una serie de operaciones matemáticas:

CÓDIGO PARA LA IMPLEMENTACIÓN DE OPERACIONES

```
def sumar(a, b):  
    return a + b  
  
def restar(a, b):  
    return a - b  
  
def multiplicar(a, b):  
    return a * b  
  
def dividir(a, b):  
    return a / b
```

Por otra parte, implementamos el código del fichero menu.py, donde desarrollaremos un sistema de menú muy similar al que ya vimos en el módulo anterior, cuando se explicaba el bucle *while*:

```
print("Este es nuestro menú infinito")  
while(True):  
    print("""Selecciona una opción:  
1) Saludar  
2) Sumar dos números  
3) Salir""")  
    opción = input()  
    if opción == '1':
```

```

    print("Hola, estamos aprendiendo Python")
elif opción == '2':
    x = float(input("Introduce el primer número: "))
    y = float(input("Introduce el segundo número: "))
    print("El resultado de la suma es: ",
operaciones.sumar(x, y))
elif opción == '3':
    print("Cerramos, nos vemos ;) bye")
    break
else:
    print("Comando desconocido, vuelve a intentarlo")

```

Hay que notar una diferencia: en la segunda opción, se hace referencia a una función “operaciones.sumar(x,y)” que no tenemos definida en ninguna parte del código.

Aquí vemos que está definida la función “sumar(x,y)”. Por lo tanto, solo nos falta conectar un fichero y el otro. Esto se consigue importando en el fichero menu.py la información de operaciones.py.

```
import operaciones
```

Con eso bastaría, ya que esa instrucción importa todo el fichero operaciones y lo incorpora a menú.py. La instrucción que utilizamos para sumar en menu.py es “operaciones.sumar(x,y)”, que responde al esquema: [nombre_de_módulo].[nombre_de_función].

Puede ser que solo nos interese importar una función en concreto del módulo de funciones y esto lo podemos hacer modificando la orden de la importación:

```
from operaciones import sumar
```

De la misma forma, podemos importar no solo una, sino varias funciones en la misma línea de importación:

```
from operaciones import Sumar, restar, multiplicar
```

Para llamar a la función, ya no haría falta la estructura [nombre_de_módulo].[nombre_de_función] sino simplemente [nombre_de_función]. En nuestro caso, podríamos llamarla únicamente con “Sumar(x,y)”.

También puede ser que necesitemos importar todas las funciones que haya en nuestro módulo, con lo cual la regla de importación quedaría así:

```
from operaciones import *
```

Hasta ahora hemos visto cómo importar una función o método desde un módulo. Pero, si tenemos otro módulo en el que estuvieran las clases, la nomenclatura y estructura, la regla para importar la clase y crear objetos sería exactamente la misma.

Por otra parte, podemos hacer que un módulo sea visible por cualquier archivo ubicándolo en la carpeta /Lib, dentro del directorio de instalación de Python.

4.2. Paquetes

En cuanto a los **paquetes**, es como subir un nivel de jerarquía con respecto a los módulos, ya que un paquete es un contenedor de módulos. Dicho de otra forma, un paquete es una carpeta que contiene uno o varios ficheros de módulos.

Sin embargo, para que una carpeta pueda ser considerada como un paquete, hay ciertos requisitos: debe tener un archivo de inicio para que Python interprete la carpeta como paquete, que ha de llamarse “__init__.py”. No hace falta desarrollar ningún código en este archivo.

Si creamos un paquete con nuestro ejemplo anterior, podríamos hacer que se agrupasen bajo un criterio matemático. Por ejemplo, nuestro fichero operaciones.py seguiría siendo de operaciones, pero podríamos añadir un módulo de constantes y otro módulo de clases.

Todo el paquete se llamará “Aritmética”, porque el nombre viene determinado por el nombre de la carpeta contenedora.

```
aritmetica/  
|-- __init__.py  
|-- operaciones.py  
|-- constantes.py  
|-- clases.py
```

Antes de poder llamar a las funciones de los distintos módulos que están en el paquete, hay que importarlo. Por ejemplo, si queremos importar el módulo de operaciones escribiremos:

```
import aritmética.operaciones
```

A la hora de llamar a las funciones desde el código del archivo menu.py, lo haremos de la siguiente forma:

```
print(aritmética.operaciones.sumar(x, y))
```

De la misma forma que en la importación de los módulos, existen otras formas de importar la información:

```
from aritmética import operaciones  
  
print(operaciones.sumar(x, y))
```

Si queremos especificar la función concreta que queremos, la llamaremos de la siguiente forma:

```
from aritmética.operaciones import sumar  
  
print(sumar(x, y))
```

Si dentro de la carpeta hubiera subcarpetas con módulos (dicho de otra forma, si dentro del paquete hubiera subpaquetes) la orden de importación tendría que seguir esa jerarquía. Por ejemplo, si dentro de nuestro paquete “Aritmética”, ponemos un subpaquete que se llame “Gráficas” y dentro un módulo que se llamase “funcionesGráficas”, la forma de importarlo sería:

```
from aritmética.gráficas.funcionesGráficas import *
```

Con eso importaríamos todo el contenido del paquete “funcionesGráficas”. En estructura de carpetas sería algo como:

```
aritmética/  
|-- __init__.py  
|-- operaciones.py
```

```

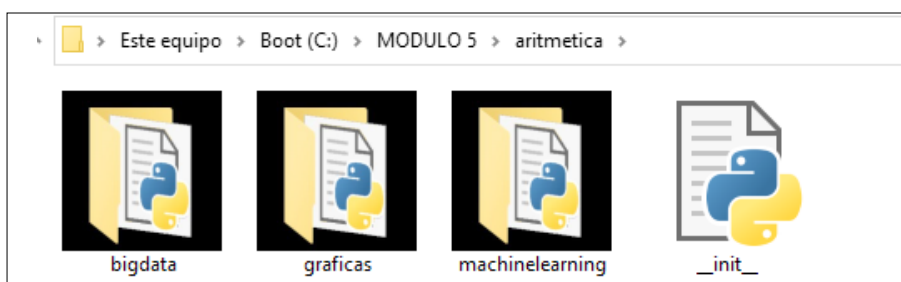
|-- constantes.py
|-- clases.py
| gráficas/
|   |-- __init__.py
|   |-- funcionesGraficas.py

```

4.3. Paquetes distribuidos

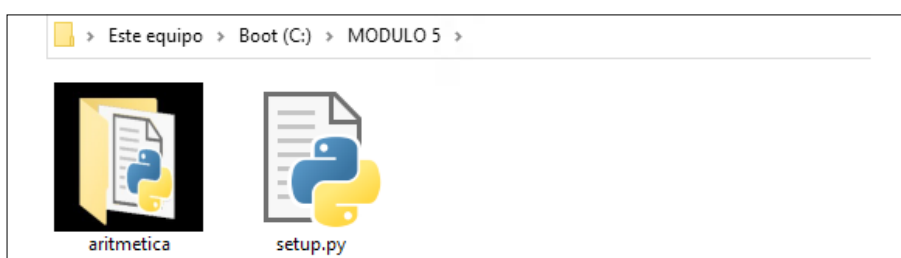
Para crear un paquete que sea distribuible, necesitamos hacer una serie de configuraciones. Pero ¿qué es un paquete distribuible? Es un paquete que contiene módulos y que podemos usarlo en el directorio que sea, sin depender de rutas, también lo podemos copiar-pegar en otras máquinas para instalar los módulos en cualquier entorno y trabajar de una manera más fácil y rápida, sin estar creando la jerarquía de carpetas y paquetes/módulos cada vez en cada entorno.

Hemos modificado un poco el ejemplo anterior, y ahora tenemos un paquete llamado “Aritmética” y dentro, tres subpaquetes, llamados “Big-Data”, “MachineLearning” y el ya conocido “Operaciones”.



Vamos por pasos:

1. Crear un fichero setup.py en el nivel superior del paquete que queremos hacer distribuible. En nuestro caso estará en /Módulo5/setup.py. Dentro de ese fichero, tendremos que desarrollar un código de instalación, donde aparte de varios datos de configuración e identificación de propiedades del paquete, tenemos que especificar el nombre del paquete y los subpaquetes que lo componen dentro del valor “packages”.



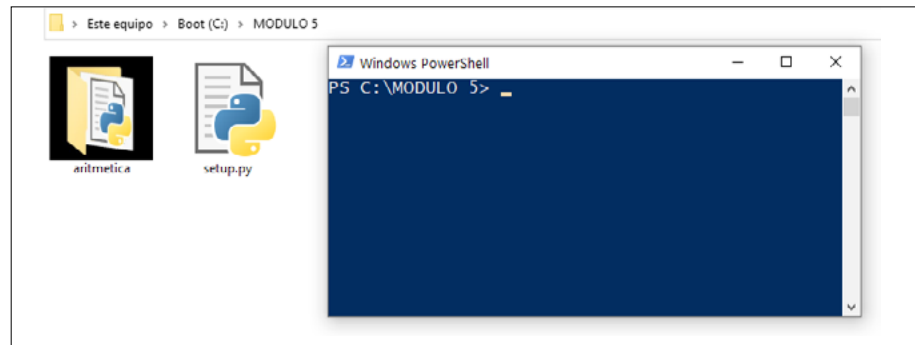
Certificación

Es importante interiorizar sobre paquetes distribuidos, pues en numerosas preguntas de la certificación se hace referencia a estos.

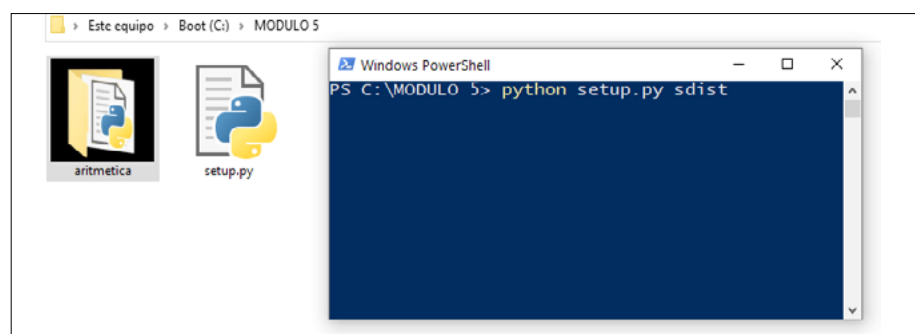
El código del fichero setup.py, en nuestro caso, será el siguiente:

```
from setuptools import setup
setup(
    name="aritmética",
    version="1.0",
    description="Paquete para operaciones aritméticas",
    author="Curso Python",
    author_email="hola@cursopython.com",
    url="http://www.cursopython.com",
    packages=['aritmética', 'aritmética.graficas', 'aritmética.bigdata', 'aritmética.machine-learning']
    scripts=[]
)
```

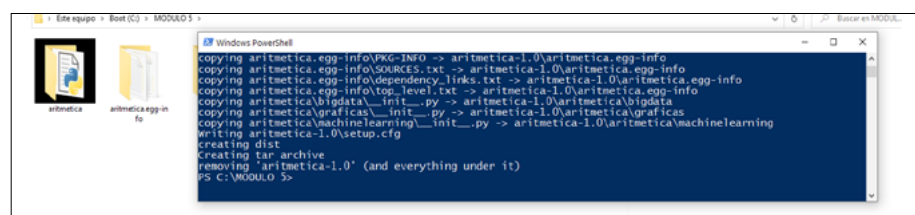
2. Abrimos una línea de comandos en esa carpeta:



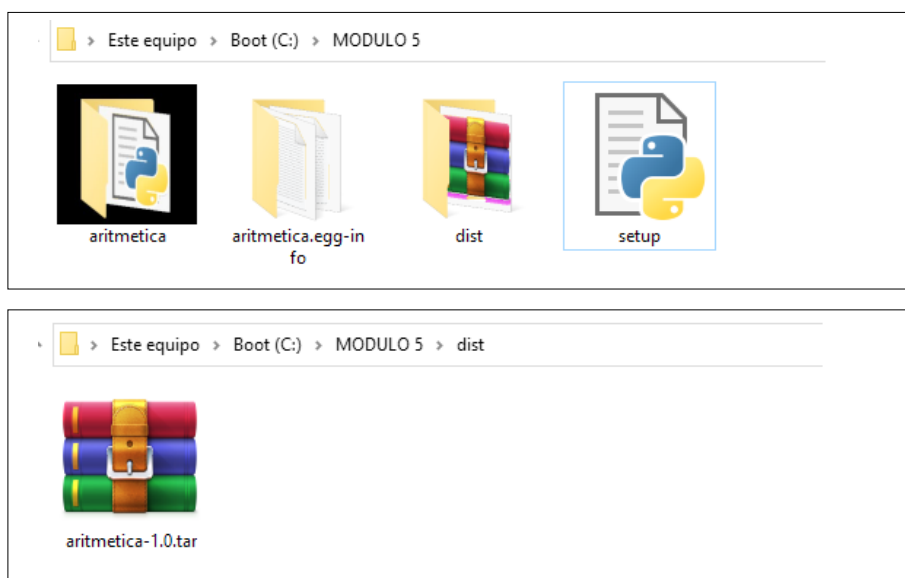
3. Llamamos a la función de Python para convertir un paquete en distribuible: "python [nombre del archivo setup] sdist":



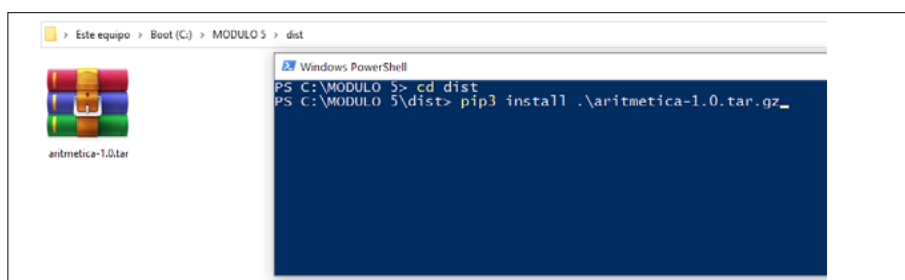
4. Entonces, Python crea nuestro paquete como distribuible:



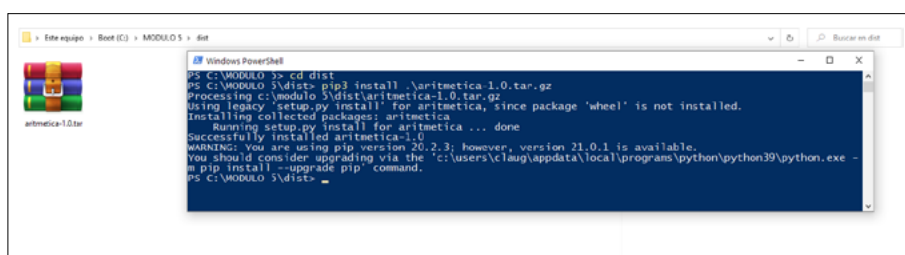
5. Comprobaremos que, en nuestro directorio, hay dos carpetas nuevas. Una de ellas se llama /dist y en su interior, Python ha convertido nuestro paquete en un fichero comprimido. Dependiendo del sistema operativo en el que estemos trabajando, este fichero puede ser un zip o un tar.gz:



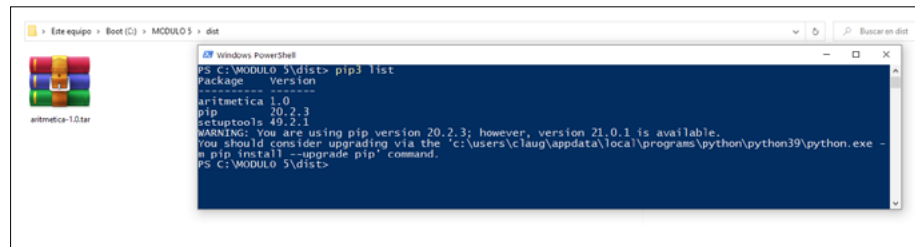
6. Ahora, nos queda la tarea de instalar nuestro paquete distribuible en nuestro entorno de Python. Con ello, podremos llamar a sus módulos independientemente de la ruta o carpeta en la que estemos trabajando. Para eso, por línea de comandos entramos dentro del directorio /dist y ejecutamos la orden de instalar: “pip3 install [nombre_del_paquete.extension]”:



7. Si todo ha ido bien, la línea de comando quedará con el mensaje: “Succesfully installed [nombre_del_paquete]”:



8. Si queremos ver todos los paquetes que tenemos instalados en nuestro entorno de Python, usaremos el comando “pip3 list”. En este caso, vemos que tenemos instalados solo los por defecto, más el que acabamos de instalar.



```
PS C:\MODULO 5\dist> pip3 list
Package Version
-----
aritmetica 1.0
pip 20.2.3
setuptools 49.2.1
WARNING: You are using pip version 20.2.3; however, version 21.0.1 is available.
You should consider upgrading via the 'c:\users\claug\appdata\local\programs\python\python39\python.exe -m pip install --upgrade pip' command.
PS C:\MODULO 5\dist>
```

A partir de este momento, ya podemos usar la información de los módulos del paquete en cualquier fichero de Python de nuestro entorno, independientemente de la carpeta o la ruta en la que nos encontremos.

4.4. Módulos estándar

Algunos módulos ya los incorpora Python, puesto que son módulos con elementos comunes y de uso frecuente. Veamos algunos de ellos:

- **“copy”**: se utiliza para copiar objetos y toda la información en memoria que contienen.
- **“collections”**: añade funcionalidades específicas a las estructuras de lista.
- **“datetime”**: para el manejo de valores de tiempo (fechas, horas, etc.).
- **“doctest”** y **“unittest”**: para la generación de documentación y test unitarios. Son esenciales para la depuración y las tareas de calidad de *software*.
- **“html”**, **“xml”** y **“json”**: permiten manejar estructuras de datos en esos tres formatos de desarrollo web.
- **“pickle”**: para el manejo de ficheros y objetos.
- **“math”**: módulo para trabajar con cálculos matemáticos. Uno de los más importantes, dada la orientación de Python a la eficiencia en los cálculos dentro del ámbito del *Machine Learning* o el *Big Data*.

- **“re”**: orientado a la optimización de comprobaciones y búsquedas. Muy vinculado a la gestión de las cadenas de caracteres.
- **“random”**: para la generación de contenidos aleatorios. Vinculado a su vez con el módulo de la generación de pruebas unitarias y test, así como con el módulo matemático.
- **“socket”**: para el establecimiento de comunicaciones entre máquinas, en base a una serie de protocolos cliente-servidor.
- **“sqlite3”**: sistema de gestión de base de datos relacional, que no necesita un proceso independiente, sino que utiliza ficheros como fuentes de datos.
- **“sys”**: para acceder a la información del sistema operativo y poder actuar sobre él.
- **“threading”**: gestión de hilos, procesos y subprocesos independientes, gracias a la ejecución en paralelo.
- **“tkinter”**: módulo para operar con interfaz gráfica en Python.



Resumen

- Los módulos son ficheros donde guardar funciones y atributos que podremos llamar desde otro fichero mediante su importación.
- Los paquetes son contenedores de módulos, pero si los módulos son ficheros, los paquetes son las carpetas.
- De igual modo que están los paquetes, existe el concepto de subpaquete (que vendría a ser una subcarpeta).
- Se puede exportar un paquete con todo su contenido para poder usarlo donde queramos, mediante la transformación a paquetes distribuidos.

Índice

Esquema de contenido	3
Introducción	5
1. Conceptos generales	7
1.1. Programación estructurada vs. Programación Orientada a Objetos	7
1.1.1. Programación estructurada	7
1.1.2. Programación Orientada a Objetos	11
1.2. Clases	14
1.3. Objetos	19
Resumen	23
2. Objetos, atributos y herencia	24
2.1. Abstracción	24
2.2. Encapsulación	27
2.3. Herencia	29
2.4. Polimorfismo	31
2.5. Herencia múltiple	32
Resumen	34
3. Métodos de colecciones y métodos especiales	35
3.1. Método constructor “__init__”	35
3.2. Método destructor “__del__”	37
3.3. Métodos matemáticos y lógicos	40
3.4. Métodos de colecciones: cadenas	43
3.5. Listas	45
3.6. Diccionarios	47
Resumen	49
4. Uso de módulos y paquetes: definición y uso	50
4.1. Módulos	50
4.2. Paquetes	53

4.3. Paquetes distribuidos	55
4.4. Módulos estándar	58
Resumen	60