

## 1. Clases y objetos.

La programación orientada a objetos (POO) es un paradigma de programación (estilo o forma de programar) que se basa en la utilización de objetos similares a los de la vida real, con sus características y comportamiento.

Python es un lenguaje orientado a objetos. Un objeto se caracteriza por tener propiedades y métodos o dicho de otra forma, datos y código. Las propiedades describen el estado del objeto y los métodos aportan funcionalidad al mismo.

Estudiaremos un ejemplo para entender mejor los conceptos de clase y objeto: “Una aplicación Python necesita trabajar con datos de los **empleados** de una empresa. De cada empleado se debe registrar un identificador, nombre, apellidos, fecha de nacimiento, salario y fecha de alta en la empresa”.

Una **clase define** de forma genérica cómo serán los objetos de un tipo determinado. En nuestra aplicación ejemplo, existirá por tanto una clase llamada “Empleado” que definirá las características o atributos de los que consta cada empleado. Los **objetos** serían cada uno de los empleados reales de la empresa y que se ajustan a la estructura definida, es decir, a la clase. **Podemos imaginar a una clase como una plantilla en base a la cual se construyen objetos.**

**El proceso de creación de objetos se conoce como “instanciación”. Se dice que un objeto es una “instancia” de una clase.**

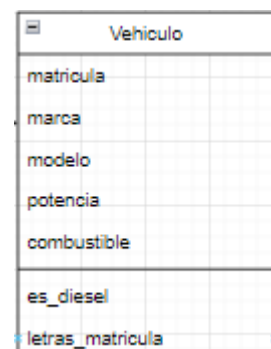
**Las ventajas que aporta esta forma de programar son la reutilización de código, la organización modular de las aplicaciones y el encapsulamiento.**

### 1.1. Propiedades y métodos.

Las **propiedades o atributos** de un objeto determinan el estado del mismo. Los **métodos** son acciones que puede llevar a cabo un objeto y que pueden alterar o no su estado. En la clase “Empleado” del ejemplo anterior, serían propiedades o atributos el identificador, nombre, apellidos, fecha de nacimiento, etc.

Los métodos son funciones definidas en la clase y que pueden servir para instanciar (crear) el objeto, modificar el estado del mismo (cambiar el valor de algún atributo), obtener o retornar el valor de algún atributo o interactuar con otros objetos.

Imaginemos ahora una aplicación que debe trabajar con vehículos. Podríamos crear una clase “Vehículo” como la que se muestra a la derecha. Sus atributos serían matrícula, marca, modelo, potencia y combustible. Sus métodos podrían ser es\_diesel y letras\_matricula, que retornan si es vehículo en cuestión es diesel o no y las letras de su matrícula respectivamente. De esta forma el



Vehículo	
matricula	
marca	
modelo	
potencia	
combustible	
es_diesel	
letras_matricula	

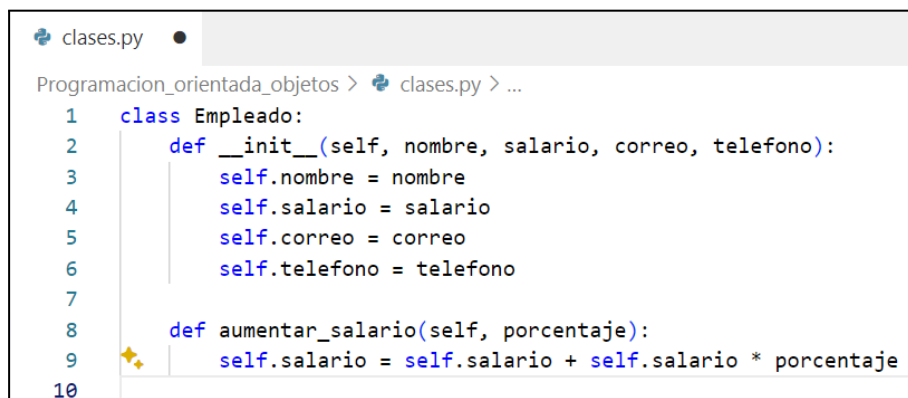
vehículo con matrícula 1290-BMZ, marca Seat, modelo Ibiza, con 100 cv y diésel sería un objeto perteneciente a la clase “Vehiculo” creada.

La **abstracción** es uno de los conceptos básicos vinculados a la POO. Mediante la abstracción, un usuario final de una clase puede utilizar sus métodos sin la necesidad de conocer cómo han sido desarrollados o implementados.

## 1.2. Creación de clases en Python. El método constructor.

Las clases en Python suelen crearse dentro de módulos. Recuerda que un módulo Python es un archivo con extensión “.py” que puede contener definiciones de funciones, clases, variables... En el anexo IV dispones de información sobre los módulos en Python.

En la imagen siguiente puedes observar un ejemplo en el que dentro de un módulo (clases.py) se define una clase y se crea su método “constructor”. El método constructor se utiliza para crear o instanciar un objeto de una clase. En el momento en que se llama al constructor se reserva memoria para el objeto y se asignan valores iniciales a sus atributos.

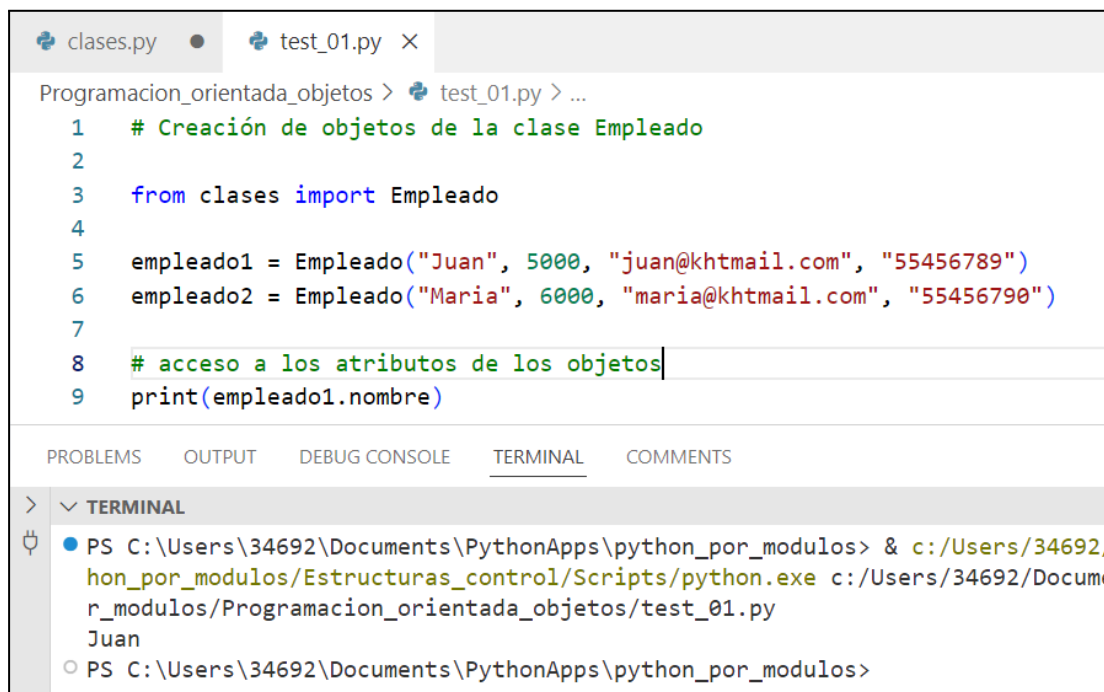


```
clases.py
Programacion_orientada_objetos > clases.py > ...
1 class Empleado:
2     def __init__(self, nombre, salario, correo, telefono):
3         self.nombre = nombre
4         self.salario = salario
5         self.correo = correo
6         self.telefono = telefono
7
8     def aumentar_salario(self, porcentaje):
9         self.salario = self.salario + self.salario * porcentaje
10
```

- Cada clase se define a partir de la palabra reservada “class” seguida del nombre de la clase, “Empleado” en este caso. Observa como el nombre de la clase comienza por una letra mayúscula. **Ver Anexo\_III, “PEP8”.**
- **La función `__init__()` es el método constructor de la clase.** Esta función recibe varios argumentos.
  - El primero, “self”, hace referencia al propio objeto que se está creando.
  - El resto de argumentos guardan los valores que se darán a cada uno de los atributos del objeto que se crea.
  - **Por norma, se utiliza el mismo nombre para argumentos y atributos. Dentro del constructor los atributos van precedidos de la palabra reservada “self”.**
  - En el ejemplo que sigue, se define la clase Empleado, pero el nombre de los argumentos no coincide con el de los atributos, todos tienen como coletilla “\_emp”.

```
class Empleado():  
    def __init__(self, nombre_emp, salario_emp, correo_emp, telefono_emp):  
        self.nombre = nombre_emp  
        self.salario = salario_emp  
        self.correo = correo_emp  
        self.telefono = telefono_emp
```

- En esta otra imagen se muestra como crear objetos de la clase Empleado. Es en este momento de la creación cuando se reserva memoria para los objetos.



The screenshot shows a code editor with two tabs: 'clases.py' and 'test\_01.py'. The 'test\_01.py' tab is active, displaying the following code:

```
1  # Creación de objetos de la clase Empleado  
2  
3  from clases import Empleado  
4  
5  empleado1 = Empleado("Juan", 5000, "juan@khtml.com", "55456789")  
6  empleado2 = Empleado("Maria", 6000, "maria@khtml.com", "55456790")  
7  
8  # acceso a los atributos de los objetos  
9  print(empleado1.nombre)
```

Below the code editor, the 'TERMINAL' tab is selected, showing the execution of the script:

```
> PS C:\Users\34692\Documents\PythonApps\python_por_modulos> & c:/Users/34692/Don  
hon_por_modulos/Estructuras_control/Scripts/python.exe c:/Users/34692/Docume  
r_modulos/Programacion_orientada_objetos/test_01.py  
Juan  
PS C:\Users\34692\Documents\PythonApps\python_por_modulos>
```

- Como se observa en la imagen, ya existen dos archivos asociados al proyecto. "clases.py" que recoge la definición de la clase "Empleado" y "test\_01.py", que es el archivo desde el que se utiliza la clase creada.
- En "test\_01.py", en principio se importa la clase definida en el módulo "clases", ya que se encuentra en otro archivo.
- Más tarde se instancian dos objetos de la clase Empleado, dando valores a sus atributos. Estos objetos son "empleado1" y "empleado2".
- Observa cómo al crear un empleado se escribe el nombre de la clase (Empleado) y entre paréntesis los valores que se asignan a cada atributo, en el mismo orden en el que se especificaron en el constructor. En realidad se está llamando al método constructor.
- **En las llamadas no se le da valor al parámetro "self" explícitamente.**
- Por último, en la sentencia "print" se muestra el nombre del objeto "empleado1".

Existe la posibilidad de asignar **un valor por defecto** a los atributos de una clase, en este caso el constructor no suele llevar argumento para ese atributo.

### Autoevaluación-1.

1. Crear dentro de un módulo Python una clase llamada Partido que servirá para modelar partidos de baloncesto. Sus atributos son: equipo local, equipo visitante, puntos local y puntos visitante.
2. Crear varios objetos de la clase Partido e imprimir sus resultados.

### 2. Atributos de instancia y de clase.

Al trabajar con clases y objetos podemos encontrarnos con dos tipos de atributos: de instancia y de clase. **Los atributos de instancia son propios de cada instancia (de cada objeto)**. Si existen varios objetos de la clase “Empleado”, cada uno de ellos tendrá un valor para el nombre, salario, correo, etc. **Sin embargo, los atributos de clase son compartidos por todos los objetos de esa clase.**

En el ejemplo siguiente, la clase “Empleado” ha sido modificada añadiendo un atributo de clase llamado “contador”. Este atributo se inicializa con el valor cero, y se incrementará en uno cada vez que se cree un objeto de esta clase (cada vez que se llame al constructor). Servirá para conocer cuántos objetos de la clase “Empleado” se crean.

```
1 class Empleado:
2     contador = 0
3     def __init__(self, nombre, salario, correo, telefono):
4         self.nombre = nombre
5         self.salario = salario
6         self.correo = correo
7         self.telefono = telefono
8         Empleado.contador += 1 # incrementa el contador de empleados
9
10    def aumentar_salario(self, porcentaje):
11        self.salario = self.salario + self.salario * porcentaje
```

**Para acceder al valor de un atributo de clase** desde dentro de la clase se utiliza el nombre de la clase y el nombre del atributo (ver línea 8 del ejemplo). Sin embargo, desde fuera de la clase puede accederse al valor de estos atributos de clase utilizando tanto el nombre de la clase como el nombre de cualquier instancia de la misma. Estudia la imagen siguiente.

```
1  # Creación de objetos de la clase Empleado
2
3  from clases import Empleado
4
5  empleado1 = Empleado("Juan", 5000, "juan@khtml.com", "55456789")
6  empleado2 = Empleado("Maria", 6000, "maria@khtml.com", "55456790")
7
8  # acceso a los atributos de instancia
9  print(empleado1.nombre)
10
11 # acceso a los atributos de clase
12 print(empleado1.contador)
13 print(empleado2.contador)
14 print(Empleado.contador)
```

### 3. Métodos de instancia y de clase.

Igual que los atributos, los métodos pueden ser de instancia y de clase. Los primeros permiten el acceso a los atributos de la instancia. Los métodos de clase, por su parte, proporcionan acceso a los atributos de la clase.

Un método de clase va precedido del decorador “**@classmethod**” y no puede acceder a los atributos de instancia. Además, recibe la clase de forma implícita como primer argumento.

En el ejemplo siguiente se ha modificado la clase “Empleado” añadiendo varios métodos.

```
1  class Empleado:
2      contador = 0
3      def __init__(self, nombre, salario, correo, telefono):
4          self.nombre = nombre
5          self.salario = salario
6          self.correo = correo
7          self.telefono = telefono
8          Empleado.contador += 1 # incrementa el contador de empleados
9
10     def aumentar_salario(self, porcentaje):
11         self.salario = self.salario + self.salario * porcentaje
12
13     def datos_empleado(self):
14         return f"Nombre: {self.nombre}, Salario: {self.salario}, Correo: {self.correo}"
15
16     # Método de clase
17     @classmethod
18     def mostrar_contador(cls):
19         return f"El número de empleados es: {cls.contador}"
```

- Los métodos “aumentar\_salario” y “datos\_empleado” son métodos de instancia. El primero aumenta el salario de un empleado en el porcentaje pasado como argumento. El segundo método retorna una cadena con nombre, salario y correo del empleado.

- El método “mostrar\_contador” es un método de clase. Observa cómo lleva el decorador “@classmethod” y el parámetro “cls”, que permite acceder al atributo de clase, “cls.contador”.

En esta imagen se crean dos objetos “Empleado” y se llama a los métodos de instancia y clase. Estudia la salida mostrada en el “Terminal”.

```
1  # Creación de objetos de la clase Empleado
2  from clases import Empleado
3
4  empleado1 = Empleado("Juan", 5000, "juan@khtml.com", "55456789")
5  empleado2 = Empleado("Maria", 6000, "maria@khtml.com", "55456790")
6
7  # llamada a método de instancia
8  print(empleado1.datos_empleado())
9  print(empleado2.datos_empleado())
10 print()
11
12 # llamada a método de clase
13 print(Empleado.mostrar_contador())
14
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

▼ **TERMINAL**

```
Nombre: Juan, Salario: 5000, Correo: juan@khtml.com
Nombre: Maria, Salario: 6000, Correo: maria@khtml.com
○
El número de empleados es: 2
PS C:\Users\34692\Documents\PythonApps\python_por_modulos> █
```

#### 4. Encapsulamiento.

El encapsulamiento es un mecanismo que permite ocultar o limitar el acceso a las propiedades y métodos de un objeto desde el exterior. Este acceso solo estará disponible a través de determinados métodos, con lo que se consigue proteger al objeto frente a posibles cambios no deseados.

En algunos lenguajes de programación este control de acceso se implementa a nivel de clase, mediante **modificadores** que indican cuando una propiedad o método es público, privado o protegido. Un elemento privado es accesible solo desde dentro de la clase. Sin embargo, si está declarado como público puede ser accedido desde el exterior. Si el atributo o método es protegido significa que es accesible desde dentro de la clase y de las subclases que heredan de esta.

**En Python esta posibilidad no existe, todo lo definido en una clase es público por defecto y por tanto accesible desde el exterior.** Para conseguir que algún elemento de una clase sea considerado protegido o privado, Python ha adoptado las convenciones siguientes:

- Si el nombre del atributo/método va precedido de un guión bajo (\_), se le considera protegido.
- Si el nombre del atributo/método va precedido de un doble guión bajo (\_\_), se le considera privado.

Será el propio programador el que decidirá si seguir o no estas convenciones. Es necesario mencionar **que los atributos siempre son accesibles desde el exterior** de esta forma:

- Atributo protegido: objeto.\_atributo.
- Atributo oculto: objeto.\_Clase\_\_atributo.

En el ejemplo siguiente, se crea una clase “Persona” con dos atributos, nombre y edad. La edad se define como un atributo privado (doble guión bajo). El acceso a la edad se realiza mediante el método “edad\_persona”. Sin embargo, como se indica arriba, puede accederse a la edad mediante “objeto.\_Clase\_\_atributo”.

```
1 class Persona():
2     def __init__(self, nombre, edad):
3         self.nombre = nombre
4         self.__edad = edad # Atributo privado
5
6     def edad_persona(self):
7         return self.__edad
```

```
1 from clases_test_02 import Persona
2
3 persona1 = Persona("Juan", 30)
4 persona2 = Persona("Maria", 25)
5
6 # Acceso a los atributos de la clase
7 print(persona1.nombre) # atributo público
8 print(persona1.edad_persona()) # atributo privado, acceso mediante método
9 print(persona1._Persona__edad) # atributo privado, acceso directo
10 print(persona1.__edad) # Genera excepción AttributeError
```

#### 4.1. Métodos getters y setters.

Los getters y setters son métodos especiales que permiten obtener (getters) y asignar (setters) valores a los atributos o propiedades de un objeto. Son muy utilizados en otros lenguajes para implementar el encapsulamiento. En Python no suelen utilizarse tanto,



ya que como se ha indicado anteriormente las propiedades/métodos de un objeto son públicos.

En las imágenes se observa un ejemplo de utilización de estos métodos. Observa como los getters retornan el valor de atributo asociado y los setters reciben un argumento con el que modificar el valor del atributo.

```
1 class Persona():
2     def __init__(self, nombre, edad):
3         self.nombre = nombre
4         self.edad = edad
5
6     def get_nombre(self):
7         return self.nombre
8
9     def get_edad(self):
10        return self.edad
11
12    def set_nombre(self, nombre):
13        self.nombre = nombre
14
15    def set_edad(self, edad):
16        self.edad = edad
```

```
1 # utilización de getters y setters
2 from clases_test_04 import Persona
3 persona1 = Persona("Juan", 25)
4 persona2 = Persona("Karla", 30)
5
6 # Acceso a los atributos: getters
7 print(persona1.get_nombre())
8 print(persona1.get_edad())
9
10 # Modificación de los atributos: setters
11 persona1.set_nombre("Juan Carlos")
12 persona1.set_edad(26)
```

#### 5. Métodos estáticos (@staticmethod).

Puede decirse que un método estático es una función definida a nivel de clase. Está asociado a la clase antes que a los objetos de la misma. Son llamados utilizando el nombre de la clase, y además no pueden alterar el estado de un objeto ni los atributos de clase.

Los métodos estáticos se diferencian de los métodos de clase en que los primeros no operan con los atributos de clase. Se utilizan en alguna de estas situaciones:

- Cuando se quiere implementar una funcionalidad que no depende de ningún atributo de la clase ni de una instancia en particular de la clase.
- Por motivos de organización del código de la aplicación. Puede crearse una clase que contenga una o varias funciones que serán utilizadas en varios puntos de la aplicación.

En Python un método estático se crea con el decorador “@staticmethod”.

En el ejemplo se crea la clase “OperaCadena” que contiene métodos para trabajar con cadenas de caracteres. Los métodos son estáticos (@staticmethod) y son llamados utilizando el nombre de la clase.

```
1 class OperaCadena():
2
3     @staticmethod
4     def primera_letra(cadena):
5         return cadena[0]
6
7     @staticmethod
8     def cadena_mayuscula(cadena):
9         return cadena.upper()
10
11    @staticmethod
12    def cadena_minuscula(cadena):
13        return cadena.lower()
```



```
1 from clases_test_03 import OperaCadena
2
3 print(OperaCadena.primer_letra("Hola Mundo"))
4 print(OperaCadena.cadena_mayuscula("Hola Mundo"))
5 print(OperaCadena.cadena_minuscula("Hola Mundo"))
```

## 6. Funciones especiales para trabajar con atributos.

Dentro de las funciones definidas de Python (built-in functions) existen varias para trabajar con los atributos o propiedades de un objeto. Estas funciones permiten la “manipulación dinámica de atributos” y son las siguientes:

- `getattr(objeto, “atributo”)`. Retorna el atributo referenciado de ese objeto.
- `hasattr(objeto, “atributo”)`. Retorna un boolean que indica si el objeto tiene o no ese atributo.
- `delattr(objeto, “atributo”)`. Elimina el atributo del objeto.
- `setattr(objeto, “atributo”, valor)`. Esta función no solo sirve para asignar un valor a un atributo sino que permite también **añadir nuevos atributos a un objeto en tiempo de ejecución**.

```
1 class Empleado():
2     def __init__(self, nombre, salario):
3         self.nombre = nombre
4         self.salario = salario
```

```
1 from clases_test_06 import Empleado
2
3 empleado1 = Empleado("Juan", 1000)
4
5 # Acceder al nombre del empleado
6 print(getattr(empleado1, "nombre"))
7
8 # Modificar el salario del empleado
9 setattr(empleado1, "salario", 2000)
10 print(getattr(empleado1, "salario"))
11
12 # Comparar si el atributo existe
13 print(hasattr(empleado1, "edad"))
14
15 # Probar si se elimina el atributo
16 delattr(empleado1, "salario")
17 print(hasattr(empleado1, "salario"))
```

## 7. Métodos mágicos o dunder (double underscore).

Son funciones definidas a nivel de clase, y su nombre comienza y termina con doble guión bajo (`__`). Aunque pueden ser invocados, suelen ser llamados por Python en situaciones especiales.

Algunos ejemplos de estos métodos son:

- `__init__()`. Es llamado cuando se construye un objeto.
- `__str__()`. Retorna una cadena destinada a los usuarios, que representa al objeto.
- `__repr__()`. Retorna una cadena destinada a los programadores, que representa al objeto.
- `__eq__()`. Método para comparar dos objetos. “eq” viene de “equals”.
- `__del__()`. Conocido como el destructor de un objeto. Es llamado cuando todas las referencias a un objeto han sido eliminadas.

En la primera imagen se define la clase “Empleado” y varios métodos dunder. En la segunda se realiza la llamada a estos métodos. Ten en cuenta que la funcionalidad del método es la que quiera darle el programador.

Puedes encontrar más información sobre métodos dunder en este enlace, <https://www.geeksforgeeks.org/dunder-magic-methods-python/>.

```
class Empleado:
    def __init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario

    def __str__(self):
        return f"Nombre: {self.nombre}, Salario: {self.salario}"

    def __eq__(self, otro):
        return self.salario == otro.salario

    def __repr__(self):
        return f"Nombre: {self.__dict__}"

    def __len__(self):
        return len(self.nombre)
```

### Autoevaluación-2.

1. Modificar la clase Partido creada en el ejercicio de autoevaluación anterior añadiendo un método (`__str__()`) que retorne el resultado del partido con el formato:  
*equipoA puntosA - puntosB equipoB.*

2. Añadir un atributo de clase que mantenga un contador con los partidos que se van creando. Crear un método que retorne el valor de ese contador.
3. Crear varios objetos de la clase Partido e imprimir sus resultados y muestra el contador de partidos.

```
1  from clases import Empleado
2
3  emp1 = Empleado("Juan", 1500)
4  emp2 = Empleado("Ana", 2500)
5
6  # llamada a __str__
7  print(emp2)
8
9  # llamada a __len__
10 print(len(emp1))
11
12 # llamada a __eq__
13 print(emp1 == emp2)
14
15 # llamada a __repr__
16 print(repr(emp1))
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMENTS

▼ **TERMINAL**

```
● PS C:\Users\jara2\Documents\python_apps\Curso2425\P00\Unidad_01> & C:\
425\P00\Unidad_01\tests\test_03_dunders.py
Nombre: Ana, Salario: 2500
4
False
Nombre: {'nombre': 'Juan', 'salario': 1500}
○ PS C:\Users\jara2\Documents\python_apps\Curso2425\P00\Unidad_01> █
```

- Estudia detenidamente cómo se realiza la llamada en cada caso.
  - Línea 7. Llamada al método `__str__()`.
  - Línea 10. Llamada al método `__len__()`.
  - Línea 13. Llamada al método `__eq__()`.
  - Línea 16. Llamada al método `__repr__()`.

8. Eliminar objetos. El recolector de basura (Garbage Collector).

Un “recolector de basura” es un proceso que se utiliza para eliminar los objetos que no van a ser utilizados más. En Python, este recolector funciona de forma automática, liberando de ese trabajo al programador.

### 8.1. Cómo eliminar objetos.

La sentencia “del” permite eliminar objetos, elementos de listas y claves de diccionarios que no van a ser necesarios. Sería la operación contraria a la creación del objeto.

Escenarios donde puede ser conveniente usar “del”:

- Liberar memoria.
- Prevenir el uso de variables no existentes.
- Evitar conflictos de nombres.

La sintaxis de esta sentencia sería: “**del referencia**”. Dónde referencia puede ser un identificador (de variable, clase, función...), un índice o trozo (slice) de una secuencia mutable, una clave de diccionario o un miembro de una clase (atributo u objeto).

En la imagen puedes observar ejemplos del uso de esta sentencia.

```
a = [1,2,3,4,5,6,7,8,9,10]
b = 10

# Eliminar un elemento de una lista
del a[2]
print(a)
|

# Eliminar una variable
del b
print(b) # Error: NameError: name 'b' is not defined
```

Cuando se intenta hacer uso de un objeto eliminado se obtiene la excepción “NameError”.

La sentencia “del” permite eliminar también atributos/métodos de clase y atributos de instancia.

### 8.2. Cómo funciona el recolector de basura ([texto extraído íntegro de Stackoverflow](#)).

*Aunque el recolector de basura funciona automáticamente en Python y no debería ser algo de lo que te tengas que preocupar, conocer algunos detalles de cómo Python gestiona la memoria pueden serte útiles (aún sin usar nunca la interfaz gc).*

*En python **todos** los datos son en realidad objetos y ocupan un espacio en memoria mayor de lo que cabría esperar en otros lenguajes, debido a que no sólo se*

guarda el dato en sí, sino también meta-información acerca del mismo (su tipo entre otras cosas).

Por ejemplo, cuando usas un entero en tu programa como aquí, `a = 10000`, el intérprete necesita reservar un espacio en memoria donde poder guardar un objeto de tipo `int` que contiene el dato 10000. Lo que ocupe el objeto depende un poco de la implementación de Python con la que estés trabajando. En CPython por ejemplo ocuparía 28 bytes. En otros lenguajes el entero es un dato de 32 bits (o de 64, según la arquitectura) por lo que ocuparía tan sólo 4 bytes (u 8, según la arquitectura).

Una variable no es más que una referencia (un puntero si prefieres llamarlo así) a la dirección de memoria donde está realmente el dato.

La memoria que se ha reservado para contener el entero 10000 permanecerá ocupada **mientras haya alguna referencia** que se refiera a ese dato. Internamente el intérprete mantiene un contador de referencias que va actualizando con cada asignación. Por ejemplo en este momento el contador de referencias sería 1, porque hemos asignado el dato a una variable `a`, y por tanto hay una referencia apuntando a él.

Si seguidamente haces por ejemplo, `b = a`, Entonces el intérprete creará una variable `b` y copiará a ella la referencia que había en `a`, de modo que en este momento hay **dos referencias** al mismo objeto. Tanto `a` como `b` se están refiriendo al mismo objeto. Su contador de referencias vale 2. Si seguidamente haces por ejemplo, `a = a + 1` entonces ocurren varias cosas:

- Se crea un nuevo objeto de tipo `int` para contener el resultado de la operación (otros 28 bytes que son necesarios)
- Se inicializa ese nuevo objeto `int` con el valor 10001 (qué es el resultado de la operación)
- Se cambia la referencia que había en `a` (que apuntaba al entero 10000) para que ahora apunte al entero 10001.
- Como consecuencia de esta asignación, el entero 10001 incrementa su contador de referencias que pasa a valer 1 (la referencia `a` apunta a él) y el entero 10000 decreuenta su contador de referencias, ya que ha dejado de apuntar a él. El contador de referencias de 10000 vale 1 (porque `b` todavía apunta a él).

Si finalmente haces algo como, `b = "Hola"`, eso creará un nuevo objeto en memoria de tipo `str`, inicializado con la cadena "Ho1a", y hará que la referencia `b` cambie su valor para pasar a apuntar a este objeto. Como consecuencia el objeto `str` tiene su contador de referencias a 1, **y el contador de referencias de 10000 pasa a cero**, pues `b` ha dejado de apuntar a él. Entonces el intérprete ejecutará el recolector

de basura para eliminar los objetos cuyo contador de referencias es 0. En este caso se liberará el objeto `int` que valía 10000, liberándose así 28 bytes de memoria.

### **Moralejas de lo anterior:**

1. Las operaciones son más complicadas de lo que aparentaban en la superficie. Si vienes de un lenguaje como C, te habrá sorprendido el follón que se ha montado para un simple `a=a+1`. En C lo que ocurría simplemente es que la posición de memoria en que estaba guardado el 10000 se sobrescribe y pasa a contener un 10001. En Python no, pues los enteros son inmutables. Un 10000 siempre será un 10000 (mientras exista)
2. La memoria no se va a liberar mientras tengas variables que se refieren a ese objeto. Por tanto una posible forma de liberar memoria cuando andes escaso de ella puede ser **borrar la referencia**.

La instrucción **`del a`**, por ejemplo, elimina la referencia llamada `a`, decrementando así el contador de referencias del objeto al que apuntaba. Considera este ejemplo:

```
>>> a = 10000
>>> del a
```

```
>>> a
Traceback (most recent call last): File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

La primera instrucción reservará 28 bytes para el entero 10000 y la referencia `a` apuntará a ese entero. Su contador de referencias será por tanto 1. La línea `del a` elimina la variable `a` (que pasa a ser no definida como se comprueba seguidamente). El entero al que `a` apuntaba pasará a tener un contador de referencias cero, y será por tanto liberado por el recolector de basura.

Es necesario mencionar que el recolector de basura de Python no actúa inmediatamente, tras eliminar la última referencia de un objeto. El procedimiento usado es el de escanear la memoria de forma periódica buscando objetos no referenciados.

9. La palabra reservada `pass`.

La palabra `pass` se utiliza como "placeholder" para un futuro código. Su ejecución no tiene consecuencias y se utiliza para evitar errores cuando no disponemos del código asociado a una función, clase, bucle o sentencia condicional.

```
# método abstracto
@abstractmethod
def get_info(self):
    pass
```

10. Atributos dinámicos.

Python permite alterar el número de atributos de un objeto de forma dinámica, tanto dentro como fuera de la definición de la clase. Este cambio en el número de atributos implica tanto la eliminación (como se vio en el apartado 8) como la adición de atributos nuevos.

Para **añadir** un atributo nuevo a un objeto basta con utilizar la sintaxis “objeto.atributo = valor”.

Llegados a este punto, para conocer todos los atributos de un objeto puede usarse el atributo `__dict__` que posee todo objeto. Este atributo contiene un diccionario con las parejas nombre de atributo y valor.

#### 11. Cómo saber si dos objetos son iguales.

- **Escenario 1.** Partimos de dos referencias y queremos determinar si apuntan a la misma dirección de memoria. Para ello hacemos uso de los operadores “is” y “not is” que determinan si dos referencias apuntan a la misma dirección de memoria.

En este caso, a y b apuntan al mismo dato. La pregunta “a is b” retorna True.

El método `id(objeto)` retorna un identificador único del objeto almacenado en memoria.

```
a = 10
b = a
print (a is b)
print (id(a))
print (id(b))
```

✓ 0.0s

True  
140725748300504  
140725748300504

- **Escenario 2.** Partimos de dos objetos y queremos determinar si sus atributos son iguales. En este caso, debemos definir el método `__eq__` en la clase y dentro de él comparar por parejas los atributos de ambos objetos. Este método retorna True en caso de que todas las comparaciones sean True.

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

    def __str__(self):
        return f'Persona: {self.nombre} + {self.edad}'

    def __eq__(self, value):
        return self.nombre == value.nombre \
            and self.edad == value.edad
```

```
from clases import Persona
x = Persona("Juan", 25)
y = Persona("Juan", 25)

print (x is y)
print (x == y)
```

✓ 0.0s

False  
True

Observa como el operador “is” retorna False y la comparación mediante “==” retorna True. Son objetos distintos pero guardan los mismos valores en sus atributos.



12. Enlaces de interés.

- <https://www.geeksforgeeks.org/dunder-magic-methods-python/>
- <https://es.stackoverflow.com/questions/418644/como-se-usa-un-garbage-colector>.
- <https://realpython.com/python-del-statement/>
- <https://www.digitalocean.com/community/tutorials/python-str-repr-functions>