UNIVERSIDAD
DE MURCIA

Tema 1. Clases y Objetos

Tecnología de la Programación

Material original por L. Daniel Hernández (*ldaniel@um.es*)
Editado por:
- Sergio López Bernal (*slopez@um.es*)
- Javier Pastor Galindo (*javierpg@um.es*)

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

5 de septiembre de 2025

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

Introducción a la Programación

Motivación

¿Qué es programar?

Introducción a la Programación

- Programar es el arte de decirle a un ordenador qué hacer. Utilizamos un **lenguaje de programación** para escribir **instrucciones** que la máquina puede entender.

- A través de la programación, podemos:
  - Automatizar tareas repetitivas.
  - Resolver problemas complejos de manera rápida.
  - Crear herramientas y aplicaciones que impactan nuestra vida diaria.

- ¿Sabías qué?
  - Muchas de las aplicaciones que usas a diario, como redes sociales o videojuegos, fueron creadas por programadores.

¿Por qué aprender programación?

Beneficios de la Programación

- **Pensamiento lógico:** Programar te enseña a pensar de forma estructurada y lógica. Desarrollas la capacidad de descomponer problemas grandes en soluciones más pequeñas y manejables.

- **Creatividad:** La programación es una herramienta poderosa para crear lo que imaginas. Desde aplicaciones hasta robots, las posibilidades son infinitas.

- **Demanda laboral:** En el mundo laboral actual, la demanda de programadores es altísima. Las habilidades de programación abren puertas a múltiples industrias como:
  - Tecnología.
  - Ciencia.
  - Finanzas.
  - Entretenimiento.

UNIVERSIDAD
DE MURCIA

Introducción a la Programación

Tipos de datos y variables

Tipos de datos y variables

- Los programas especifican instrucciones y necesitan datos para funcionar. Se dividen en dos grandes grupos: primitivos y compuestos.
- **Tipos de datos primitivos (o elementales)**:
  - Formados por un único elemento.
  - Tipos: carácter, numérico, booleano, enumerado.
- **Tipos de datos compuestos**:
  - Formados por una agrupación de elementos.
  - Tipos: string, array, registro, conjunto, lista, diccionario.
- Las variables son las ubicaciones de almacenamiento de los datos. Cada variable se caracteriza por su **nombre**, **tipo de dato** y el **valor** almacenado.
- Declarar una variable es establecer el tipo de dato que se almacenará e identificar una zona de memoria con el identificador.
- Asignar valor a una variable es almacenar información en la zona de memoria del identificador. La primera asignación se denomina inicialización.
- Una constante es una variable que solo puede ser inicializada una vez.

Tipos de datos elementales. Variables.

Ejemplo en *Processing*

```
// Tipos de datos primitivos en Processing
char letra;                    // Declaración de un carácter
letra = 'A';                   // Asignación

int numeroEntero;              // Declaración de un entero
numeroEntero = 10;             // Asignación

float numeroReal;              // Declaración de un número real
numeroReal = 3.14;             // Asignación


boolean esVerdadero;           // Declaración de un booleano
esVerdadero = true;            // Asignación

enum Estado { INICIO, EJECUCION, FIN }; // Definición de enumerado
Estado estadoActual;                    // Declaración de la variable
estadoActual = Estado.INICIO;           // Asignación
```

Tipos de datos compuestos. Variables.

Ejemplo en *Processing*

```
// Tipos de datos compuestos en Processing
String texto;                       // Declaración de una cadena de texto
texto = "Hola Mundo";               // Asignación

int[] numeros;                      // Declaración de un array de enteros
numeros = {1, 2, 3};                // Asignación

class Persona {                     // Definición del registro
  String nombre;
  int edad;
}
Persona persona;                   // Declaración de variable de tipo Persona
persona = new Persona();           // Inicialización del registro
persona.nombre = "Javier"          // Agregar un campo

HashSet<Integer> conjunto;                  // Declaración del conjunto
conjunto = new HashSet<Integer>();          // Inicialización
conjunto.add(1);                            // Agregar elementos

ArrayList<String> lista;            // Declaración de una lista de Strings
lista = new ArrayList<String>(); // Inicialización de la lista
lista.add("Elemento 1");            // Agregar elementos

HashMap<String, Integer> diccionario;          // Declaración de diccionario
diccionario = new HashMap<String, Integer>(); // Inicialización
diccionario.put("clave1", 100);                // Agregar clave-valor
```

Tipado dinámico en Python

Características de las variables

- Python es un lenguaje dinámicamente tipado. No se requiere declaración explícita de variables (al realizar una asignación, se declara implícitamente). Si se asigna posteriormente un valor de un tipo diferente, la declaración de la variable cambiaría.

```
numero_entero = 2              # Sin declaración explícita
numero_entero = "hola"         # Se puede asignar cualquier tipo
```

- En cualquier caso, **en esta asignatura siempre declararemos explícitamente las variables**. Así mismo, evitaremos el cambio de tipo de una variable en nuestros programas.

```
numero_entero : int            # Declaración explícita
numero_entero = 4              # Asignación
numero_entero = "hola"         # Funciona, pero lo evitaremos!
```

- Como vemos, en Python la declaración explícita se realiza con el operador **:** mientras que la asignación con el operador =. En la inicialización se usaría la forma nombre_variable : tipo_dato = valor_inicial

Tipos de datos elementales en Python

Convenciones en Python:

- Las variables siguen la convención de nombres snake_case.
- Las *constantes* se escriben en mayúscula (si bien no existen estrictamente).

```
numero_entero: int = 2
numero_real: float = 10.01
caracter: str = '2'                 # Se interpreta como un string
booleano: bool = True
cadena: str = "una cadena"          # Se interpreta como un string

# Asignación múltiple
numero_entero, numero_real, booleano = 2, 10.01, False

# Destrucción de variables
del(numero_entero)                   # No es común en Python

# Las "constantes" se escriben en mayúsculas
PI: float = 3.1415                   # La "constante" PI (puede cambiar)
```

¡En Python no se utiliza el operador ; para diferenciar instrucciones! (aunque no dará error si lo usamos)
¿Entonces cómo separamos las instrucciones...? :-)

Tipos de datos compuestos en Python

Estructuras mutables de datos

Los tipos de datos compuestos definen estructuras de datos que agrupan datos elementales, otros datos compuestos, o ambos a la vez.

- Estructuras mutables. Se pueden cambiar sus términos una vez creadas.
  - Listas: **secuencia** de elementos arbitrarios. Se escriben entre corchetes y se separan con comas.

```
lista: list = [1, "hola", 3, 1]
```

  - Conjuntos: colección de elementos arbitrarios **únicos**. Se escriben entre llaves y se separan con comas.

```
conjunto: set = {1, "hola", 3}
```

  - Diccionario: conjuntos con objetos indexados de la forma **clave: valor**. La clave puede ser cualquier valor inmutable.

```
diccionario: dict[str, object] = {"a": 1, "b": "hola", "c": 3}
```

Tipos de datos compuestos en Python

Estructuras inmutables de datos

- Estructuras inmutables. No pueden cambiar sus términos.
  - Strings: **secuencia** de valores que representan códigos Unicode. Se escriben entre comillas.

```
cadena: str = "hola"
```

  - Tuplas: **secuencia** de elementos arbitrarios. Se escriben entre paréntesis y se separan con comas.

```
tupla: tuple[int, str, int] = (1, "hola", 3)
```

  - Rangos: **secuencias** que se construyen con range([start,] stop [[, step]]).

```
rango: range = range(1, 10, 2) #Valores: 1, 3, 7, 9
```

  - Conjuntos congelados (Frozensets): La versión inmutable de los conjuntos.

```
conjunto_congelado: frozenset = frozenset({1, "hola", 3})
```

Mutabilidad y Casting

Conceptos en Python

Una variable es mutable si se puede cambiar directamente el valor de la variable sin cambiar su referencia en memoria.

- En Python, `id`(var) muestra la referencia de la variable `var`.
- Si se hacen dos asignaciones a una variable e `id`() cambia significa que se ha necesitado crear una nueva referencia para albergar el nuevo valor, pues la variable era inmutable.
- Números, booleanos y strings son **inmutables** (no se pueden modificar el propio valor, aunque sí se les puede asignar un nuevo valor, lo que reemplaza la variable/referencia).

El casting es el proceso donde el valor de una variable se **interpreta** como otro tipo de dato (sin modificarse el tipo).

```
// Casting explícito en Processing
float numeroDecimal = 9.75;
int numeroEntero = (int) numeroDecimal;       // Casting de float a int (9)
float numeroDecimal2 = (float) numeroEntero; // Casting de int a float (9.0)
```

- Python define funciones para el cambio de tipo: `int`(), `float`(), `str`(), `list`(), `tuple`(), `set`(), y `dict`().
- También aplica casting implícito en operaciones entre enteros y reales.

UNIVERSIDAD
DE MURCIA

Introducción a la Programación

Programación estructurada. Secuencias de instrucciones

Estructura Secuencial

Conceptos básicos

Un programa consta de una secuencia de órdenes directas. El conjunto de instrucciones **vienen dadas por el lenguaje** de programación.

- Sentencias de Asignación: Consisten en el paso de valores de una expresión o literal a una zona de la memoria.
- Lectura, `input`(): Recibir desde un dispositivo de entrada algún dato.
- Escritura, `print`(): Mandar a un dispositivo de salida algún valor.
- Tamaño, `len`(): Calcula el número de datos que tiene una secuencia (por ejemplo, un string o una lista).
- Identificación, `id`(): Retorna la referencia en memoria de una variable.
- Tipo, `type`(): Indica el tipo de dato de una variable.

Python define una serie de funcionalidades para su libre uso:

- **Built-in Functions:**
  `https://docs.python.org/3/library/functions.html`

UNIVERSIDAD
DE MURCIA

Introducción a la Programación

Programación estructurada. Condicionales

if-else

Estructura Condicional

Conceptos en Python

Es aquella que ejecuta ciertas órdenes si se cumple una condición booleana.
En Python, se usa la sentencia compuesta con cláusulas if, elif, else.

- Condicional simple.

```
if condicion:
    estructura
```

```
# En una sola línea (Inline)
if condicion: expresion
```

- Condicional doble.

```
if condicion:
    estructura_if
else:
    estructura_else
```

```
# Inline
variable = expresion_si_true if condicion else expresion_si_false
```

- Condicional anidado.

```
if condicion1 [op condicion2 [op condicion3] ... ]:
    estructura_if
elif condicion:
    estructura_else_if
else:  # Casi obligado si se usa elif.
    estructura_else
```

Expresiones booleanas

Conceptos en Python

Las estructuras condicionales están directamente ligadas con las expresiones booleanas. En Python tenemos:

- Comparaciones:
  - x == y: Verifica si x e y son iguales.
  - x != y: Verifica si x e y son diferentes.
  - x > y: Verifica x es mayor que y.
  - x >= y: Verifica si x es mayor o igual que y.
  - x < y: Verifica si x es menor que y.
  - x <= y: Verifica si x es menor o igual y.
- Operaciones:
  - e1 and e2: Evalúa si ambos operandos son verdaderos; el resultado es True solo si ambos e1 y e2 son verdaderos.
  - e1 or e2: Evalúa si al menos uno de los operandos es verdadero; el resultado es True si al menos uno de e1 o e2 es verdadero.
  - not e: Niega el valor de e; el resultado es True si e es False, y viceversa.

UNIVERSIDAD
DE MURCIA

Introducción a la Programación

Programación estructurada. Bucles iterativos *for* y *while*

Estructura Iterativa

Bucle `while`

La estructura iterativa consta de los siguientes pasos:

1. Se parte de una variable de control que se inicializará a cierto valor.
2. Se comprueba una condición booleana donde interviene la variable de control.
3. Si la condición es cierta, se ejecutarán nuevas estructuras.
4. Entre las estructuras se debe modificar la variable de control.
5. Se vuelve al paso 2.

Esto se repite hasta que la variable de control haga falsa la condición booleana.

En Python se utiliza la sentencia while de la siguiente forma:

```
var_de_control = valor_inicial
while expresion_booleana_con_la_var_de_control:
    estructuras
    modificar la variable de control
else: # opcional, no se realiza si se ejecuta la sentencia break
    estructuras
```

¿Te has dado cuenta de que Python no delimita las estructuras condicionales o iterativas abriendo ({) y cerrando (}) con llaves? ¿Cómo distingue bloques?

Estructura Iterativa

Bucle `for` en Python

En Python, se utiliza la sentencia for para iterar sobre secuencias:

```
for var_de_control in secuencia:  # normalmente la secuencia es una lista
    estructuras
```

- Iteración sobre una lista de números:

```
numeros: list[int] = [10, 20, 30, 40, 50]
for num in numeros:
    print(num)
```

- Iteración sobre los caracteres de un String:

```
palabra: str = "Python"
for letra in palabra:
    print(letra)
```

- Iteración con range(start, stop):

```
for i in range(2, 5):           # 2, 3, 4
    print(i)
```

- Iteración con range(start, stop, step) con un incremento específico:

```
for i in range(0, 10, 2):       # 0, 2, 4, 6, 8
    print(i)
```

Sentencias `break` y `continue` (prohibidas en TP)

Uso en Bucles

Sentencia break:

- Solo puede ocurrir sintácticamente en un bucle for o while.
- Terminará el bucle más cercano y omitirá la cláusula opcional else.

```
for i in range(5):   # 0, 1, 2, 3, 4
    if i == 3:
        break
    print(i)         # Solo se imprime 0, 1 y 2
```

Sentencia continue:

- Solo puede ocurrir sintácticamente en un bucle for o while.
- Continúa con la siguiente iteración del bucle más cercano.
- No ejecutará el código que aparezca después de continue dentro de la misma iteración.

```
i:int = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)         # Solo se imprime 1, 2, 4 y 5
```

UNIVERSIDAD
DE MURCIA

Introducción a la Programación

Programación procedimental. Funciones y procedimientos

Funciones. Parámetros Posicionales

- Función: es una **secuencia** de instrucciones bajo un **nombre** que retorna un valor. En Python se usa la palabra reservada def.

```
def nombre_funcion (p1: tipo_p1, p2: tipo_p2, ...) -> tipo_retorno:
    estructuras de la función: secuencial, condicional, repetitiva.
    return valores
```

- En Python una función puede retornar varios valores.
  - Agrupa todos los datos de retorno en una tupla.
- Procedimiento: función que no tiene la instrucción de retorno.
- Si una función/método tiene *n*-parámetros se puede invocar a la función con *n*-argumentos de tal forma que el 1^{er} argumento se sustituya por el 1^{er} parámetro, el 2^{o} argumento por el 2^{o} parámetro, etc ...
  Son parámetros posicionales.

```
def funcion(a: int, b: int, c: int, d: int) -> None: # 4 parámetros
    print(a, b, c, d)

funcion(1, 2, 3, 4) # Invocamos con 4 parámetros posicionales.
```

MUY IMPORTANTE

- Una función no debe tener más de una responsabilidad/propósito.
- Una "función" debe, en la medida de lo posible:
  - o realizar una acción (procedimiento)
  - o retornar un cálculo (función)
  - y no debería "nunca" realizar las dos cosas a las vez.

Ejercicio.

Se quiere hacer un programa que haga lo siguiente:

> *Mostrar si un número entero, dado por el usuario, es un número primo.*

¿Cómo se haría desde un punto de vista procedimental?

Funciones. Valores Por Defecto. Palabras Clave

- Los k-últimos parámetros de un función pueden ser opcionales.
  - Los opcionales determinan un valor literal por defecto.
  - **Primero** los obligatorios **y después** los opcionales (o por defecto).

```
def fun(a:int, b: int, c: int = 3, d: int = 4):
    pass        #  2 posicionales + 2 opcionales.
```

- Python permite invocar por palabras claves (keywords).
  - **Usar keyword** consiste en especificar el nombre del parámetro en la invocación.
  - El orden de los parámetros pueden cambiarse.
  - Los keywords siempre se pondrán al final.

```
# Para la función anterior
fun(b=2, d=4, a=1, c=3) # Invocamos con 4 keywords.
fun(1, 2, d=4, c=3) # Los keywords al final
fun(d=4, 1, 2, c=3) # Incorrecto
```

UNIVERSIDAD
DE MURCIA

Introducción a la Programación Orientada a Objetos

Clases

Programación modular

- La programación modular divide un programa grande en partes más pequeñas y manejables, llamadas módulos. Estos módulos pueden ser reutilizados y mantenidos de manera independiente.
- Las clases son una herramienta fundamental en la programación modular al encapsular datos y funcionalidades en una estructura unificada.
  - Ejemplo de la clase `Rectángulo` en *Processing*:

```
class Rectangulo {                  // Definición de la clase

   int ancho, alto;                 // Atributos

   Rectangulo(int ancho, int alto) { // Constructor
      this.ancho = ancho;
      this.alto = alto;
   }

   void mostrarDimensiones() {      // Método para mostrar atributos
      println("Ancho:", this.ancho, "Alto:", this.alto);
   }

   int calcularArea() {             // Método para calcular el área
      return ancho * alto;
   }
}
```

Clases

- Una clase define un conjunto de datos (atributos) y funciones (métodos) que operan sobre ellos (habitualmente con el operador this).
- El método constructor es especial pues inicializa los atributos de la clase y es invocado al crear un nuevo objeto.
- Un objeto es una instancia particular de una clase, creado en muchos lenguajes mediante la palabra clave new que invoca al constructor.

```
Rectangulo rect1 = new Rectangulo(10, 20);   // Creación rectángulo 1
Rectangulo rect2 = new Rectangulo(5, 15);    // Creación rectángulo 2
```

- El paradigma de programación que modela clases para instanciar objetos recibe el nombre de **Programación Orientada a Objetos (POO)**, cuyos fundamentos se estudian desde el **Tema 1** hasta el **Tema 5**.
- Las clases facilitan la implementación de modelos abstractos, como estructuras de datos o entidades del mundo real, que se definen formalmente como **Tipos de Datos Abstractos (TDA)**[1]. Se profundizará a partir del **Tema 6**.

[1] Esto es, una posible forma de implementar *TDA Persona* o *TDA Lista* es con una clase en POO.

Clases

- Frecuentemente, una clase modela una entidad de la vida real sobre la que trabaja nuestro programa.
- Diseñar una clase es abstraer lo que tienen de común entes parecidos: calculadoras, estudiantes, coches, animales, figuras geométricas...
- Ejemplo. Consideremos dos estudiantes. Ambos parecen tener **coincidencias** en
  - Los mismos atributos: están en un curso, tienen una edad, ...
  - Los mismos comportamientos: se desplazan, estudian, cambian objetos en la mochila, ...

  Realmente dos estudiantes tienen los mismos atributos (aunque con diferentes valores que definen su estado) y mismos comportamientos (con resultados posiblemente diferentes dependiendo de su estado).
- Cuando un conjunto de entidades presentan los mismos atributos y métodos, y se diferencian solo en los estados, dicho conjunto se puede abstraer en una clase en POO.

Declaración de clases en Python

- Informalmente, una clase es una plantilla o molde para construir entidades individuales (objetos).
  - Por ejemplo, con la clase *Usuario* podemos crear los objetos *sergio* y *javier* que permitirán a nuestro programa gestionar la información y las acciones de ambos bajo una misma especificación común.
- En Python, una primera aproximación es el siguiente esquema:

```
class Clase:
    def __init__(self, p1: int, p2: float, ...) -> None: # Constructor
        self._p1: int = p1                                # Atributos
        self._p2: float = p2
        ..
    # Métodos de la clase
    def metodo(self, p: int):
        pass    # instrucción que no hace nada (dummy)
```

1. Se usa la palabra reservada class
2. Se da un nombre a la clase `Clase`.
3. Se especifican los atributos (el guión bajo indica que se deben considerar privados, se explicará en el próximo tema)
4. Se definen los métodos

- Recuerda la importancia en Python de las **tabulaciones** para delimitar bloques de código (condicionales, bucles, funciones, clases...)

Declaración de clases en Python

- En Python, los atributos y métodos de instancia se reconocen porque utilizan la palabla clave self (análogo a *this* en Processing).

```
class Rectangulo:
    def __init__(self, ancho: float, alto: float) -> None:
        self._ancho: float = ancho
        self._alto: float = alto

    def mostrar_dimensiones(self) -> None:
        print(f"Ancho: {self.ancho}, Alto: {self.alto}")

    def calcular_area(self) -> float:
        return self.ancho * self.alto
```

- Los métodos con self son invocados a través de un objeto existente con la notación punto: objeto.nombre_método_instancia()

```
# Creación de dos objetos de la clase Rectangulo
rectangulo_1: Rectangulo = Rectangulo(10, 20)
rectangulo_2: Rectangulo = Rectangulo(5, 15)

# Uso de los métodos de instancia
rectangulo_1.mostrar_dimensiones()
area_2: float = rectangulo_2.calcular_area()
```

- En particular, el parámetro self apunta al objeto que invoca al método de instancia, teniendo acceso también a los atributos de la instancia.

Ejercicio

Ejercicio.

Desarrolla una aproximación inicial para declarar las clases correspondientes a los siguientes tipos de objetos:

1. Libro: Declara una clase `Libro` que contenga atributos como título, autor y número de páginas. Implementa un método que imprima la información completa del libro.
2. Coche: Declara una clase `Coche` que tenga atributos como marca, modelo y velocidad actual. Implementa un método que simule acelerar el coche aumentando su velocidad.
3. Cuenta Bancaria: Declara una clase `CuentaBancaria` que contenga atributos como saldo y titular. Implementa métodos para depositar y retirar dinero, y para mostrar el saldo actual.

Objetivo: Practicar la declaración de clases, atributos y métodos de diferentes tipos de objetos.

UNIVERSIDAD
DE MURCIA

Introducción a la Programación Orientada a Objetos

Objetos

Encapsulamiento

- En POO una clase es una plantilla y define un tipo de dato.
- En POO un objeto representa a una entidad (física o abstracta) que viene dado por la particularización de una abstracción.
- Un objeto queda definido por:
  - Un estado, dado por los valores concretos de sus atributos de la representación.
  - Un comportamiento, representado por los métodos.
- La agrupación se llama encapsulamiento:

Fuente www.lineaysalud.com

- El encapsulamiento es el proceso por el que **un objeto** tiene sus propios datos y métodos.
- El encapsulamiento requiere que su atributos queden ocultos (pero se estudiará con más detenimiento con los modificadores de acceso).

Constructores de Objetos

- Al trabajar con clases hay que crear objetos: se necesitan constructores.
- Un constructor es un método especial que se caracteriza porque:
  - No retorna nunca un valor, crea un objeto generando una referencia de memoria al mismo (ver ciclo de vida).
  - El método suele definirse con el nombre de la clase. No obstante en Python, se define el método `__init__()`.
  - Sus parámetros se identifican con algunos atributos. Por lo tanto, el constructor establece el estado inicial del objeto (inicialización).

```
class Persona:

    def __init__(self, nombre: str, edad: int) -> None:
        self._nombre: str = nombre
        self._edad: int = edad

    def saludar(self) -> str:
        return f"Me llamo {self._nombre} y tengo {self._edad} años."

    def cumplir_años(self) -> None:
        self._edad += 1

juan: Persona = Persona("Juan", 30)
ana: Persona = Persona("Ana", 25)
```

Acceso a los atributos y métodos

Cambiando el estado de un objeto

- Se necesita una referencia, nombre_objeto, para acceder a un objeto.
- Para usar un atributo o método de un objeto se usa la notación punto.
  - nombre_objeto.atributo referencia a un atributo del objeto.
  - nombre_objeto.metodo() referencia a un método del objeto.
- Modificar el estado de un objeto es cambiar los valores de sus atributos.
- En cualquier caso, no se deben modificar ni acceder a los valores de los atributos directamente. Se debe realizar SIEMPRE mediante métodos definidos para ello. **Los motivos, en el siguiente tema**.

Ejemplo.

```
# Lectura
print(ana.saludar())    # Me llamo Ana y tengo 25 años (bien)
juan.cumplir_años()     # Cambia el estado del objeto juan (bien)
print(f"Me llamo {juan._nombre}")    # Funciona pero está mal

# Modificación
juan._edad = juan._edad + 1          # Funciona pero está mal
```

Funciones vs Métodos en Python

- **RECUERDA**: Una función recibe como entrada una lista de parámetros
  - Se invoca como una instrucción más.

```
def funcion (lista de parámetros):
    cuerpo
```

- Un método (comportamiento de un objeto) debe incluir el parámetro self
  - Se invoca a través de un objeto de la clase (notación punto)

```
def metodo (self, lista de parámetros):
    cuerpo
```

- **Ejemplo**:

```
def funcion(x: int) -> int:
    return 2*x

class Prueba:
    def metodo(self, x: int) -> int:
        return funcion(x)

p: Prueba = Prueba()
print(f"{p.metodo(10)}, {funcion(100)}") # Salida: 20, 200
```

El objeto self

- En POO se suele usar un objeto especial (this, self o me según lenguaje)
- self se refiere al objeto que actualmente está ejecutando el código.
  - Recuerda que obj.atributo y obj.metodo() hacen referencia al atributo .atributo y al método .metodo() del objeto obj.
  - Por tanto, self.atributo y self.metodo() hacen referencia al atributo .atributo y al método .metodo() **del objeto que esté ejecutando el código** en ese momento.
- Ejemplo. Considera el siguiente código Python

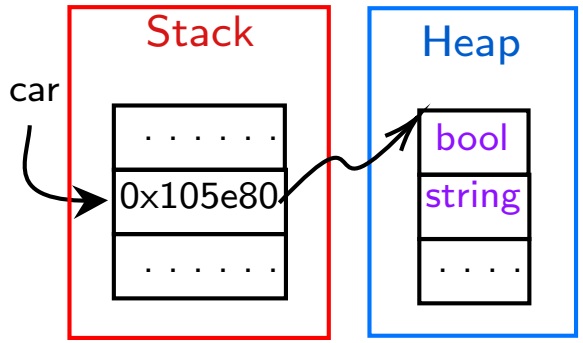
```
class Estudiante:
    ...
    def notaPOO (self, nota: float):
        self._nota = nota   # self.atributo = parámetro

maria: Estudiante = Estudiante()
maria.notaPOO(10)
```

  - `maria` es el objeto que ejecuta el código (línea 7).
  - `maria.notaPOO(10)` referencia al método `notaPOO()` del objeto `maria`.
  - En la ejecución, se realizará la instrucción `self.nota = 10` (línea 4).
  - Como el objeto que invoca es `maria`, el objeto self = `maria` (línea 4). Es como si ejecutáramos: `maria.nota = 10`
  - En consecuencia, *al objeto maria se le asignará un 10 a su atributo nota*.

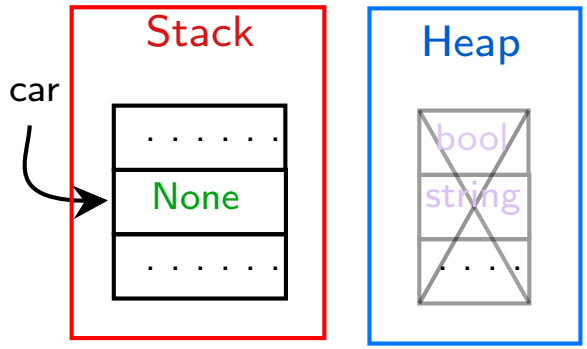
Ciclo de vida de un objeto

- Para la **creación de un objeto** se necesita
  1. **Declaración** de variable indicando la clase del objeto.
  2. **Construir** el objeto, invocando al constructor definido en la clase.
  3. **Almacenar** en la variable la referencia retornada por el constructor.
- La variable es realmente una referencia de memoria al objeto.
- La referencia está en Stack (memoria estática). El objeto en Heap (memoria dinámica).
- Para que un objeto exista, éste necesita al menos una referencia. Entonces se puede modificar el estado mediante métodos.
- Un objeto deja de existir si no tiene variables referenciándolo (None). El *recolector de basura* borrará al objeto de la memoria.

```
// Declaración
car : Car
// Construcción
car = Car()
```




```
car.turnHeadLights()

car = None
```



UNIVERSIDAD
DE MURCIA

Introducción a la Programación Orientada a Objetos

POO para Resolver Problemas

Cómo se usa la POO para resolver problemas

- En todo problema se pueden **ABSTRAER** conceptos o entidades.
  Ejemplo. *Sistema de cajero automático: los usuarios y el cajero.*

- Cada concepto define su clase, **ENCAPSULANDO** su estructura de datos (atributos) y operaciones (métodos).
  Ejemplo.
  - *Clase `Usuario`: Almacena información como el saldo en la cuenta y proporciona métodos para interactuar con el usuario.*
  - *Clase `Cajero`: Almacena la cantidad de dinero disponible y proporciona métodos para interactuar con el cajero.*

- Por lo tanto, en POO la funcionalidad se distribuye entre las clases existentes, proporcionando **MODULARIDAD**.
  Ejemplo. *La funcionalidad del usuario y del cajero está separada y definida en sus respectivas clases.*

Cómo se usa la POO para resolver problemas

Modularidad de un programa en clases

```
# Declaración de clases
class Usuario:
    """
    Representa a un usuario de un cajero
    """
    def __init__(self, nombre: str, saldo: int):
        self._nombre: str = nombre
        self._saldo: int = saldo

    def aumentar_saldo(self, cantidad: int):
        self._saldo += cantidad


class Cajero:
    """
    Representa a un cajero
    """
    def __init__(self, dinero: int):
        self._dinero_disponible: int = dinero

    def sacar_dinero(self, usuario: Usuario, cantidad: int):
        if cantidad <= self._dinero_disponible:
            self._dinero_disponible -= cantidad
            usuario.aumentar_saldo(cantidad)
        else:
            print("No hay suficiente dinero.")
```

Cómo se usa la POO para resolver problemas

- El programa principal usará objetos concretos pertenecientes a las clases.
  Ejemplo. *usuario1 y cajero1 serán instancias de Usuario y Cajero*

```
usuario1: Usuario = Usuario("Javier", 500)  # En el módulo principal
cajero1: Cajero = Cajero(1000)
```

- Los objetos irán invocando métodos de otros objetos para modificar sus estados y resolver el problema (intercambio de mensajes). En particular, la solución es un conjunto de estados deseado.
  Ejemplo.

```
cajero1.sacar_dinero(usuario1, 200)     # En el módulo principal
```

  - En un momento dado, se envía un mensaje al `cajero1` solicitando retirar dinero mediante `cajero1.sacar_dinero(usuario1, 200)`.
  - El objeto `cajero1` ejecuta dicho método, reduciendo su propio dinero disponible y aumentando el saldo del `usuario1` enviándole el mensaje `usuario1.aumentar_saldo(200)`.
  - El objeto `usuario1` se encarga entonces de sumarse el dinero.
  - Solución: El estado deseado es que el `usuario1` reciba el dinero solicitado (aumente su saldo) y el `cajero1` registre la transacción (actualice el dinero disponible).

Beneficios de la POO para resolver problemas

Objetivo. Usar POO para implementar modelos que resuelvan problemas mediante:

- Abstracción. Proceso mental de extracción de las características esenciales de un concepto o proceso descartando los detalles.
  - El programa resuelve el problema definiendo clases (abstracción de tipo) y ejecutando sus métodos (abstracción operacional).
- Encapsulamiento. Proceso por el que se agrupan datos y operaciones, ocultando los detalles internos.
  - Cada objeto protege y maneja internamente su estado, ofreciendo los métodos necesarios para su manipulación.
- Modularidad. Descomposición del sistema en un conjunto de módulos poco acoplados (independientes) y cohesivos (con significado propio).
  - Cada clase define su funcionalidad de la manera más independiente posible, comunicándose con las demás mediante paso de mensajes.
- Jerarquización. Estructurar por niveles jerárquicos los elementos que intervienen en el problema (se verá en próximos temas).
  - Jerarquía de composición (Asociación, agregación). Clases complejas se componen de otras clases más simples.
  - Jerarquía de clasificación (Herencia). Clases generales (superclases) con comportamientos comunes y las clases más específicas (subclases) que añaden o modifican esos comportamientos.

Ejercicio de POO

Ejercicio.

Plantea un programa en Python con las siguientes características:

1. Declara una clase `Perro` que tenga los atributos nombre y energía. Implementa un **método** `jugar()` que disminuya la energía del perro en 1 cuando juega.
2. Declara una clase `Persona` que tenga los atributos nombre y perro (un objeto de la clase `Perro`). Implementa un método `pasear()` que interactúe con el perro, llamando al método `jugar()` del perro y mostrando la energía restante. *¿Debería la persona acceder directamente al atributo de la energía del perro?*
3. Define una **función** que donde la persona juega con el perro hasta que su energía se agota.
4. Escribe el programa principal donde se cree una instancia de `Perro` y de `Persona`. Luego simula el paseo invocando a la función anterior.

Objetivo: Practicar la declaración de clases, la creación de instancias de objetos y el envío de mensajes (llamadas a métodos).