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*) y Javier Pastor Galindo (*javierpg@um.es*)

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

22 de septiembre de 2024

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

Introducción a la Programación

Motivación

¿Qué es programar?

- 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 que?
  - 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: elementales 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 numeroDecimal;           // Declaración de un número decimal
numeroDecimal = 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 de 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
string: 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!
¿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]
```

  - 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)
```

  - 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
```

```
# Inline
if condicion: sentencia
```

- Condicional doble.

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

```
# Inline
var = exp_si_true if condicion else exp_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 paso específico:

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

Sentencias `break` y `continue`

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: type1, p2: type2, ...) -> tipo de dato:
    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} parámetros, etc ...
  Son parámetros posicionales.

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

fun(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 6**.
- 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 7**.

[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):
        pass    # instrucción 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 al *this*).

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

```
# Creación de dos objetos de la clase Rectangulo
rect1: Rectangulo = Rectangulo(10, 20)
rect2: Rectangulo = Rectangulo(5, 15)

# Uso de los métodos de instancia
rect1.mostrar_dimensiones()
area2: float = rect2.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.

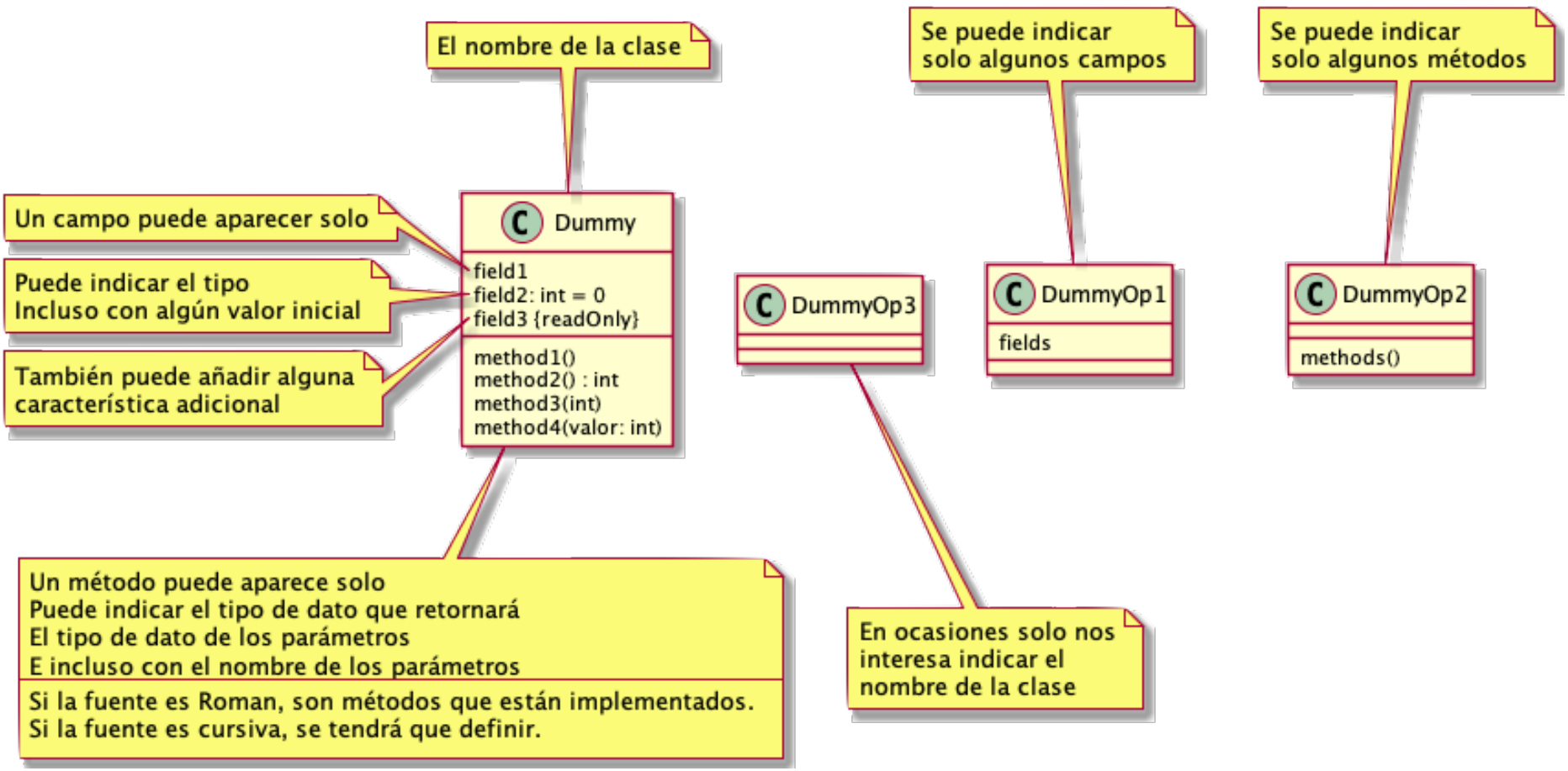
Desarrolla una aproximación inicial para declarar las clases correspondientes a los siguientes tipos de objetos:

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.

Representación visual de una clase

Unified Modeling Language (UML)



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, nombreObjeto, para acceder a un objeto.
- Para usar un atributo o método de un objeto se usa la notación punto.
  - nombreObjeto.atributo referencia a un atributo del objeto.
  - nombreObjeto.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)}")
```

El objeto self

- En POO se suele usar un objeto especial (this, self o Me)
- 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

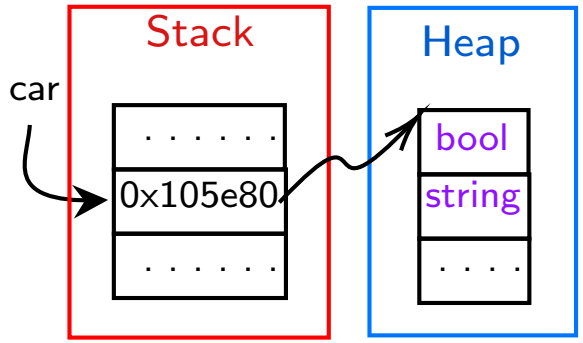
```
class Estudiante:
    ...
    def notaPOO (self, nota: float):
        self._nota = nota   # self.variable = parámetro

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*.

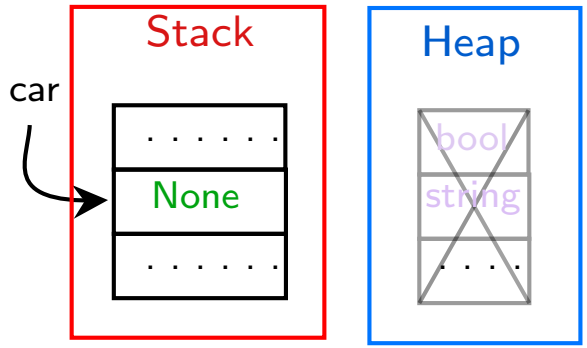
Ciclo de vida de un objeto

- 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, us: Usuario, cant: int):
        if cant <= self._dinero_disponible:
            self._dinero_disponible -= cant
            us.aumentar_saldo(cant)
        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).

UNIVERSIDAD
DE MURCIA

Tema 2. Miembros, Sobrecarga y Visibilidad

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
22 de septiembre de 2024

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

Miembros de una clase

Introducción

- Una clase representa a un conjunto de objetos con la misma estructura.
- Una clase consta de una serie de elementos, llamados miembros.

```
class UnaClase {
   miembros
}
```

- Los miembros de una clase son los elementos de una estructura que la caracterizan y, por ende, permite definir a sus objetos:
  - Las **atributos**, que definirán el estado de cada objeto.
  - Los **métodos**, que establecen el comportamiento (o funcionalidad) de todos los objetos.
  - Pueden existir otros: constantes, eventos, clases internas, etc.
  - En cada lenguaje de programación los miembros pueden variar.
- Los constructores establecen el estado inicial de un objeto recién creado pero formalmente no se considera miembro de una clase.
- Las clases *protegen sus miembros* con restricciones de visibilidad para evitar que cualquier otra clase lo manipule directamente.

UNIVERSIDAD
DE MURCIA

Miembros de una clase

Atributos

Atributos de una clase

- Los atributos miembros definen el estado de la **clases** o sus **objetos**.
- También se llaman campos.
- Pueden ser de tipo simple (entero, booleanos...) o compuesto (listas, objetos...)
- Existen 2 tipos de atributos: de clase y de instancia.
- **Atributos de instancia**. Define el estado particular de un objeto.
  Ejemplo. Sobre la clase `Estudiante`, un alumno tendrá:
  - Color de los ojos
  - Referencia al centro en el que estudia
  - Notas obtenidas
  - ...
- **Atributos de clase.** Define características generales de la clase.
  Ejemplo. Con la clase `Estudiante` podríamos guardar detalles como:
  - Número de estudiantes total (varía con el tiempo)
  - Sistema de calificaciones (constante en el tiempo)
  - ...

Atributos de instancia (u objeto)

Definición y uso interno con self

- Son miembros de una clase que definen el estado de un objeto.
- Se definen dentro del constructor con el prefijo `self.nombreAtributo`. Se utilizan **internamente** siempre con dicho prefijo.

```
class Estudiante:

    def __init__(self, color_ojos: str, notas: list[float]):
        self._color_ojos = centro # Atributo de instancia _color_ojos
        self._notas = notas       # Atributo de instancia _notas

    def imprimir_color_ojos(self):
        print(f'Estudiante con ojos {self._color_ojos}')
```

- Hay, al menos, tantos como objetos o instancias se hayan creado.

```
javi: Estudiante = Estudiante("verdes", [4.5, 6.0, 9.2])
sergio: Estudiante = Estudiante("marrones", [2.0, 8.5, 6.0, 4.4])
# Existen los dos colores y las dos listas, respectivamente
## javi._color_ojos y javi._notas
## sergio._color_ojos y sergio._notas
```

Atributos de instancia (u objeto)

Uso externo con la notación punto

- Los atributos de instancia se pueden acceder **externamente** con la notación `nombreObjeto.nombreAtributo`, pero es una **mala práctica**. La clase debería proveer métodos específicamente para ello.

```
color_ojos_javi : str = javi._color_ojos # no se debe acceder así
javi.imprimir_color_ojos() # método que accede a atributo
```

- **CUIDADO**. En Python se crea un **nuevo atributo de instancia** si se realiza una asignación sobre un atributo no declarado en la clase.

```
javi._primer_apellido = "Pastor"
# crea el atributo _primer_apellido sólo para el objeto javi
```

- Para proteger a los objetos de declaraciones no deseadas o de posibles erratas fuera del constructor, la instrucción `__slots__ =[identificadores]` en una clase fija los identificadores de atributos de instancia permitidos.

```
class Estudiante:

    __slots__ = ["_color_ojos", "_notas"]
    ...

javi._color_hojos = "verdes"     # AttributeError
```

Atributos de clase

- Son miembros de una clase que definen información de la clase en su conjunto (constantes, valores por defecto, contadores, ...).
- Por tanto, definen atributos comunes (no propios) de los objetos. Solo existe uno por clase, independientemente del número de objetos.
- Se definen fuera de los métodos y ubicándose en **memoria estática**, existiendo desde que se carga la clase y hasta el final de ejecución.
- Son accesibles a través del nombre de la clase con la notación punto: `NombreClase.nombreAtributo`

```
class Estudiante:
    num_estudiantes = 0                              # Var. Clase

    def __init__(self,calificaciones: list[float]):
        self._calificaciones: list[float] = calificaciones
        Estudiante.num_estudiantes += 1              # Var. Clase

est: Estudiante = Estudiante([5.5,9.9])
Estudiante.num_estudiantes  # Atributo de clase
est.num_estudiantes         # Atributo de clase, NO se recomienda!
```

Al ser compartidas por todos los objetos de la clase, también se podría acceder a través de un objeto de la clase. **NO** se recomienda este uso. De hecho, la modificación a través de `nombreObjeto.nombreAtributoClase` no modifica el atributo de clase, sino que crea un nuevo atributo de instancia `nombreAtributoClase` para el objeto.

Otro tipo de variables

Variables locales

- No hay que confundir los **atributos de una clase** con otro tipo de **variables**.
- Además de los atributos de instancia y de clase, en los métodos se pueden usar **variables locales** y **globales** (no son miembros de la clase).

- **Variables locales**. Las auxiliares propias de un método o algoritmo, quedando su acceso limitado a un ámbito local.

  Ejemplo. Sobre la clase Estudiante se pueden usar variables locales en el método para calcular la media de las notas de un estudiante.

```
class Estudiante:
    def __init__(self, nombre: str, notas: list[float], altura: float):
        self._nombre: str = nombre
        self._notas: list[float] = notas
        self._altura: float = altura

    def calcular_media(self) -> float:
        suma_notas: float = sum(self._notas)    # Variable local
        num_notas: int = len(self._notas)       # Variable local
        media: float = suma_notas / num_notas   # Variable local
        return media
```

Otro tipo de variables

Variables globales

- **Variables globales**. Definidas en niveles superiores de un programa pues representan conceptos que transcienden a una clase en particular. Son accesibles por cualquier clase o función en un ámbito amplio.

  Ejemplo. La clase Estudiante podría acceder a variables globales como el nombre del planeta donde viven (la Tierra no sólo existe para los estudiantes, es algo más amplio) o el aire que respiran (no es una característica diferenciadora de los estudiantes, pues el aire también es respirado por profesores o animales).

```python
# Variables globales
PLANETA: str = "Tierra"
AIRE: str = "Oxígeno"

class Estudiante:
    def __init__(self, nombre: str, edad: int, altura: float):
        self._nombre: str = nombre
        self._edad: int = edad
        self._altura: float = altura

    def informacion_global(self) -> str:
        return f"{self._nombre} vive en el planeta {PLANETA}
                    y respira {AIRE}."
```

Ámbito de las variables

- El ámbito de una variable hace referencia a las partes del programa en el que una variable es reconocida.
- Una variable es local respecto de un bloque de código si solo es reconocida en ese bloque. Fuera de él, la variable no es reconocida.
- Una variable es global respecto de un bloque de código si se reconoce tanto dentro de dicho bloque como fuera de él.
- En Python se tienen las siguientes situaciones:
  - Variables declaradas "fuera" de una función son globales para ella.
  - Los parámetros de una función son locales para ella.
  - Las variables declaradas en una función son siempre locales.

```
s: str = "global"       # Variable global
def f(param: object): #
    s = "local"         # Nueva variable local
```

  - Para modificar una variable como global dentro de una función debe especificarse la keyword `global`.

```
s: str = "global"     # Variable global
def f(param: object):
    global s          # Indicamos que "s" es global
    s = "global2"     # Modificamos la variable global
```

Ejemplo completo con atributos y variables

```
planeta: str = "La Tierra"                          # Var. Global
class Estudiante:
    num_estudiantes: int = 0                        # Atr. Clase
    def __init__(self, calificaciones: list[float]):
        self._calificaciones = calificaciones       # Atr. Instancia
        Estudiante.num_estudiantes += 1             # Atr. Clase
        print(f'Este vive en {planeta}')            # Var. Global
    def calificacion_media(self) -> float:
        sum: float = 0                              # Var. Local
        for i in range(0,len(self._calificaciones)):
            sum += self._calificaciones[i]          # Var.local y Art. Instancia
        return sum/len(self._calificaciones)

est: Estudiante = Estudiante ([5,10])
est.calificaciones              # Atributo de objeto
Estudiante.num_estudiantes      # Atributo de clase
est.num_estudiantes             # EVITAR: Atributo de clase a través de objeto
```

- Observa que si sólo vamos a **leer** una variable global en una función (p.e., `planeta`), podemos evitar la keyword `global`. Si se quiere modificar (p.e., `planeta='Marte'`), entonces sí se debe usar la keyword `global` (de lo contrario se crearía una variable local `planeta`).
- Por otro lado, ¿te has preguntado qué ocurre si intentamos modificar el atributo de clase `num_estudiantes` a través del objeto `est` de la forma `est.num_estudiantes = N`? ¡Piensa y pruébalo!

Algunas aclaraciones

- Hemos distinguido 2 tipos de atributos y 2 tipos de variables.
- Cada lenguaje de programación implementa los atributos y variables de forma diferente:
  - Java, por ejemplo, trabaja con atributos de instancia y de clase, así como variables locales. Sin embargo, no se pueden definir variables globales fuera de una clase.
  - Hemos visto que Python sí permite trabajar con variables globales y locales, así como atributos de instancia y de clase.
- Recuerda que el ámbito de los parámetros de una función es local (es decir, fuera de la función no se reconocen). No obstante, en Python:
  - Los parámetros de una función se pasan por referencia (y no por copia de valor). La función tiene acceso a las posiciones de memoria originales.
  - Por otro lado, recuerda que sólo pueden modificarse los tipos de datos mutables.
  - Por tanto, las funciones pueden modificar directamente las **listas**, **conjuntos**, **diccionarios** y **objetos** de nuestros programas gracias al paso por referencia (la función recibe el puntero a memoria) y la mutabilidad (el tipo de dato nos permite la modificación).

Ejercicios de atributos

Ejercicio.

- Si tuvieses que definir constantes matemáticas como π, e, ... ¿de qué tipo de atributo serían? ¿qué nombre le pondrías a la clase?
- Si tienes una clase para los empleados públicos ¿el salario base sería clase o de instancia? ¿y los complementos por antigüedad?
- Considera una casa, donde se emplean las clases `Casa`, `Habitación` y `Silla`. Define, para cada clase, atributos de instancia y de clase ¿Qué relación hay entre estas clases?

UNIVERSIDAD
DE MURCIA

Miembros de una clase

Métodos

Métodos de una clase

- Los métodos representan el comportamiento de los objetos.
- Un método está asociado con una acción que puede realizar un objeto.
- Siempre se coloca en "el interior" de la definición de una clase.

  Ejemplo.

```
class Coche:
    def __init__(self, gas: int):
        self._gas: int = gas  # Inicializa los litros de gas

    def recarga(self, n: int):  # MÉTODO
        self._gas += n  # Recarga los litros de gas
```

- Existen tres tipos de métodos:
  - de instancia: uno por cada objeto.
  - de clase: uno para toda la clase y común a todos los objetos.
  - estáticos: es independiente de clases y objetos.

Métodos de Instancia

- Los métodos de instancia son los que están asociados a un objeto.
- Se tiene que invocar a través de un objeto (existente) de la clase.
- Se llaman con esta notación punto: objeto.métodoInstancia().
- Un método de instancia puede invocar a otro con self.métodoInstancia()
- Accede a los atributos de instancia (ver pág. 10).
- En Python se reconocen porque tiene como primer parámetro self que apunta al objeto que invoca al método.

  Ejemplo.

```
class Coche:
    def __init__(self, gas: int):# MÉTODO DE INSTANCIA (CONSTRUCTOR)
        self._gas: int = gas  # Inicializa los litros de gas

    def descargar(self, n: int): # MÉTODO DE INSTANCIA
        self._gas -= n  # Descarga los litros de gas
```

- Los métodos `__init__(self, ...)` o `descargar(self, ...)` son ejemplos de métodos de instancia.

Métodos de Clase

- Los métodos de clase son los que están asociados a una clase.
- Se pueden invocar sin existir ningún objeto de la clase.
- Se usa notación punto: NombreDeLaClase.nombreMétodoClase()
- **No pueden acceder** a las variables de instancia.
- Operan solo sobre variables de la clase (afectará a todas las instancias de los objetos), por lo que también se puede usar objeto.nombreMétodoClase()
- En Python se reconocen porque tienen el decorador `@classmethod` y como primer parámetro cls que apunta a la clase cuando el método es invocado.

Ejemplo.

```
class Rueda:
    def __init__(self, radio: float):
        self._radio: float = radio

    @classmethod
    def descripcion(cls) -> str:
        return "Una rueda es un objeto circular que gira sobre un eje."

# Usa el método de clase para obtener la descripción
print(Rueda.descripcion())
```

Métodos Estáticos

- Los métodos estáticos no están asociados ni a una clase ni a objetos.
- Se pueden invocar sin existir ningún objeto de la clase.
- Se usa notación punto: NombreDeLaClase.nombreMétodoEstático()
- No pueden acceder ni a las variables de clase ni a las de instancia.
- Por tanto son métodos independientes para crear métodos auxiliares o de utilidad (funciones útiles para usar en cualquier momento).
- En Python se reconocen porque tienen el decorador @staticmethod.

  Ejemplo. Imagina que tienes la clase Util que contiene funciones de conversión de medidas. Son útiles porque se podrían usar en cualquier momento (sin depender de un objeto o clase en particular).

```
class Coche:
    @staticmethod
    def convertir_a_galon(litros: int) -> float:  # MÉTODO ESTÁTICO
        return litros * 0.264172  # Convierte litros a galones
```

- Si bien se podría implementar este tipo de operaciones en funciones, los **métodos estáticos** permiten que estas funcionalidades queden agrupadas y organizadas dentro de clases (en lugar de distribuidas por todo un programa) aunque no accedan a atributos de clase o instancia.

Métodos Mágicos en `Python`

- En Python existen métodos especiales llamados métodos mágicos.
- **__new__(cls , ...)** . Se invoca cuando un objeto es creado. **No usar**.
- **__init__ (self)** Debe implementarse siempre
  - Es el método inicializador de instancia que se invoca siempre, una vez construido el objeto con __new__(cls). **SOLO HAY UNO.**
  - Inicializa los atributos de la instancia (objeto) recién creada.
- **__del__ (self)** No se recomienda su uso.
  - Es un método finalizador que se invoca cuando un objeto es recogido por el recolector de basura (garbage collected, GC)
  - Un objeto será recogido (destruido) cuando deja de tener referencias hacia él (es decir, variables apuntándolo).
- **__str__ (self)**
  - Es un método de acceso que retornar un string con información del objeto entendible por el usuario. Se invoca en `print()` y `str()`.
- __lt__ (self , other), __le__ (self , other), __eq__ (self , other), __ne__ (self , other), __gt__ (self , other), __ge__ (self , other).
  - Definen los operadores $<$, $\leq$, $==$, $>$ y $\geq$ entre objetos.
    `is` identidad: True sii a y b son el mismo objeto.
    `==` igualdad: True sii __eq__ es True (por defecto, identidad).

Ejemplo de Métodos Mágicos

```python
class Coche:
    def __init__(self, marca: str, gas: int):
        self._marca: str = marca
        self._gas: int = gas

    def __str__(self) -> str:  # Método mágico __str__
        return f"Coche: {self._marca}, Gasolina: {self._gas} litros"

    def __eq__(self, otro: 'Coche') -> bool:  # Por qué las comillas?
        return self._marca == otro._marca # no recomendado
    def __gt__(self, otro: 'Coche') -> bool:  # Método mágico __gt__
        return self._gas > otro._gas  # acceso a otro._gas no recomendado

coche1: Coche = Coche("Toyota", 50)
coche2: Coche = Coche("Honda", 60)
coche3: Coche = Coche("Toyota", 50)

# Uso de __str__
print(coche1)  # Coche: Toyota, Gasolina: 50 litros

# Comparaciones con __eq__ y __gt__
print(coche1 == coche2)  # False, diferente marca
print(coche1 == coche3)  # True, misma marca
print(coche1 is coche3)  # False, diferente objeto
print(coche2 > coche1)   # True, coche2 tiene más gasolina
print(coche1 > coche3)   # False, ambos tienen la misma gasolina
```

UNIVERSIDAD
DE MURCIA

Sobrecarga de métodos

Sobrecarga

- En ocasiones nos puede interesar que un objeto pueda realizar un mismo método con parámetros diferentes.
- Equivalentemente, nos gustaría mandar el mismo mensaje pero con diferente número o tipo de argumentos.

Ejemplo.

Para sumar dos números con una calculadora no parece razonable tener distintas operaciones de suma según sus argumentos. Todo lo contrario, todos los métodos deberían llamarse igual. Lo que cambian son los argumentos.

```
int    sumar(int a, int b) { return a+b; }
double sumar(double a, double b) { return a+b; }
Fraccion sumar(Fraccion a, Fraccion b) { ... }
Complejo sumar(Complejo a, Complejo b) { ... }
```

Listing 1: Distintas Sumas

Definición de Sobrecarga

- La sobrecarga permite tener un mismo identificador (nombre) para métodos con distinto tipo o número de parámetros.
- Se distinguen los distintos métodos sobrecargados por sus parámetros, ya sea por su **cantidad** o sus **tipos**.

```
class Persona {
   ...
   float distancia(Persona p) { .... }
   float distancia(Casa casa) { ... }
}
```

Listing 2: Sobrecarga de métodos

- Por tanto, dos métodos sobrecargados con el mismo número de parámetros, tipos y órdenes se considerarán iguales.
- La sobrecarga también puede aplicarse a los constructores.

Sobrecarga de constructores

Existen dos tipos de constructores:

- Constructor implícito: Se invoca por defecto si no se define un constructor. El lenguaje garantiza así la construcción de objetos (sin parámetros de inicialización).
- Constructor explícito: Se define como método en la clase, con la parametrización necesaria para dar valor a los atributos de instancia. Siempre suele definirse, sobrecargando así el constructor implícito.

En **Python**, el método __init__() es un **constructor explícito** que inicializa el objeto. Si no se definiera, **Python** invocaría a un constructor implícito (inicialmente el objeto no tendría atributos de instancia).

```
class Biblioteca:
    total_libros: int = 0  # Atributo de clase

class Libro:
    def __init__(self, titulo: str, autor: str): # constructor explícito
        self.titulo: str = titulo  # Atributo de instancia
        self.autor: str = autor    # Atributo de instancia

biblioteca: Biblioteca = Biblioteca() # se invoca constructor implícito

libro1: Libro = Libro("1984", "George Orwell") # constructor explícito
```

Sobrecarga en Python

- Realmente en Python **no existe la sobrecarga** de funciones: no puede haber varios métodos con el mismo nombre (aunque tengan parámetros diferentes).
- Al tener varias funciones con el mismo nombre, será la última función la que sobreescriba la implementación de las anteriores.

```
def f(p1: int):
    print(p1*10)

def f(p1: float, p2: float):  # Sobreescribe la función f
    print(p1*p2)

f(1) #  Error: f() tiene dos parámetros
```

- Este comportamiento se denomina sobreescritura de una función.
- En **Python**, la sobrecarga (tener una función/método con un nombre y varias posibilidades de parámetros) podemos simularla de diferentes formas.

a. Simulando la sobrecarga con parámetros opcionales

- RECUERDA: Los k-últimos parámetros de una 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.
```

- Para el siguiente código solo existirá la última función, la que tiene dos parámetros y no existe la función con un parámetro.

```
def f(p1):
    print(p1*10);

def f(p1, p2):
    print(p1*p2);
```

- Podemos simular la sobrecarga de la función con este código:

```
def f(p1, p2 = None, p3 = None):
    print(p1*p2) if p2 else print(p1*10)

f(1)
f(10, 100)
```

b. Simulando la sobrecarga con parámetros variables

- RECUERDA: Se puede definir una función con número variable de parámetros.
- Deberá considerar un parámetro que empieza con el signo *, colocándose siempre después de los parámetros opcionales de la función.

```
def f(p: str, *args): # args se usa por convención
    print(p)
    for val in otros:
        print (f"\t{val}")

f("El primero", 2, 3, "el cuarto")
```

- Usar * significa empaquetar argumentos variables (*packing arguments*) como un solo parámetro, recibiéndose dentro como una **tupla**.
- De esta forma, podemos obtener otra aproximación a la sobrecarga de funciones con uno o varios parámetros.

```
def f(*args):
    if len(p) == 1:
        print(p[0])
    elif len(p) == 2:
        print(p[0]+p[1])

f(1)
f(10, 100)
```

c. Simulando la sobrecarga con diccionarios

- El problema de la aproximación anterior es que los parámetros no tienen nombre, teniendo que jugar con el orden de aparición.
- Sin embargo, se puede invocar a una función suministrando una lista variable de parámetros con *keywords* con el operador **, colocándose siempre después de los parámetros opcionales de la función.

```
def fun(**kwargs): # kwargs se usa por convención
    print(kwargs) # Muestra el diccionario

fun(a=1, b=2, c=3, d=4)
```

- Usar ** significa empaquetar argumentos variables con keywords como un sólo parámetro, recibiéndose dentro de la función un **diccionario**.

```
def f(**kwargs):
    if 'name' in kwargs:
        print("Nombre:" , kwargs['name'])
    if 'phone' in kwargs:
        print("Teléfono:" , kwargs['phone'])

f(name = 'Luis', phone =  '868931234')
```

- Así también se puede simular la sobrecarga, accediendo al diccionario (clave-valor) y usando condicionales de la forma más adecuada.
- Problema: La función debe conocer las claves (parámetros)

Resumen de tipos de parámetros

- Existen 4 tipos de parámetros en Python
- Posicionales obligatorios.
  - Siempre aparecerán los primeros.
- Posicionales optativos.
  - Aparecerán después de los obligatorios con valores por defecto.
- Variables sin keyword.
  - Los argumentos variables se identifican con un solo parámetro
  - Aparecerá después de los opcionales.
  - Empezará con *.
  - El nombre usual es args, tratándose como una *tupla*.
- Variables con keyword.
  - Los argumentos variables se identifican con un solo parámetro
  - Aparecerá después de los variables sin kyword.
  - Empezará con **.
  - El nombre usual es kwargs, tratándose como un *diccionario*.

Ejemplo.

```
def funcion(ob1, ob2, op1='a', op2='b', *args, **kwargs):
    pass
```

UNIVERSIDAD
DE MURCIA

Visibilidad

Acceso a los miembros de una clase

Interacción entre clases: paso de mensajes

- RECUERDA: La forma en la que deben interactuar las clases entre sí es mediante paso de mensajes:
  - Un objeto envía a otro un mensaje (solicita al otro una acción)
    - El envío de mensaje se realiza con una llamada a un método.
    - El objeto receptor es el responsable de ejecutar el método invocado.
  - El objeto receptor reaccionará (dependiendo del mensaje):
    - Cambiando el estado. Es decir, modificando los atributos.
    - Retornando información sobre su estado.
    - Realizar una rutina concreta.
    - A su vez, puede verse obligado a enviar otros mensajes. Es decir, llamando a otros métodos de otros objetos.
- A continuación, profundizamos en cómo una clase especifica el tipo de acceso a sus atributos y métodos (esto es, indicar quién puede acceder al estado para leerlo o modificarlo).

Acceso a los miembros de una clase

- No se puede permitir que un objeto externo consulte o cambie directamente los valores de un atributo:
  - Podría asignar valores sin sentido. (p.e. una edad negativa)
  - Puede destruir objetos sin control. (p.e. maria.pareja = null)
  - Podría consultar variables auxiliares o detalles de implementación (p.e., consultar una constante interna).
  - No se deben retornar datos ocultos (p.e. obtener PIN de la tarjeta).
- Sólo nos interesa aquellos métodos que la clase exponga, abstrayéndonos del estado, funcionamiento o implementación de esa clase.
  Ejemplo.
  - La llave de un coche es el mecanismo para arrancar un coche.
  - La implementación de cómo se arranca nos da igual (es privado).
  - Además, solo se puede actuar sobre el arranque con la llave.
- **RESUMEN**: No se debe acceder directamente a los atributos de un objeto ni ejecutar libremente cualquier método existente.
- Un objeto **encapsulará** su estado poniendo restricciones de **visibilidad** de sus atributos y métodos mediante **modificadores de acceso**.

Visibilidad de los miembros

- La **visibilidad de un miembro** especifica el alcance o ámbito de un atributo o método a nivel de clase, subclase o paquete (desde dónde son accesibles con la notación punto).
- La interpretación del alcance de estos modificadores **varía en función del lenguaje de programación**. En **Python**:
  - **Alcance público**. El miembro es accesible desde cualquier lugar, tanto dentro de la clase, como desde otras clases y paquetes. Se utiliza cuando quieres que un atributo o método sea accesible de forma abierta.
  - **Alcance protegido**. El miembro de la clase es accesible desde la propia clase y las clases derivadas (subclases). Se utiliza cuando se quiere permitir acceso jerárquico a los miembros en herencia.
  - **Alcance privado**. El miembro de la clase solo es accesible dentro de la misma clase. Ninguna otra clase, ni siquiera las que heredan de esta, puede acceder a este miembro. Se utiliza para encapsular datos y evitar que otros objetos accedan directamente.
- Para una buena **encapsulación** añadiremos **modificadores de acceso** (público, protegido o privado) a los atributos y métodos de la clase.

Modificadores de acceso en Python

- En realidad, Python **no define explícitamente modificadores de acceso o visibilidad**.
- Esto quiere decir que los miembros de una clase son siempre públicos. Todas las clases tienen acceso directo a los miembros de otras clases.
- Ante la visibilidad total, la privacidad se expresa de manera informativa con el nombre que le damos al atributo o método.
- Por convenio, un guión bajo antes del nombre (_identificador) indica que el atributo, método o clase debe tratarse como protegida. No obstante, cualquier otra clase sigue pudiendo acceder a ellas.
- Si se utilizase el doble guión bajo antes del nombre (__identificador) provocará que el intérprete de Python modifique el nombre del miembro de la clase (*name mangling*) a _NombreClase__identificador. Dificulta el acceso, pero _NombreClase__identificador sigue siendo también público.

Buenas prácticas en Python

- **Miembros públicos** sin anteponer guiones (var_publica).
  - Pueden ser accedidos desde otras clases del programa.
  - Los **métodos públicos** son esenciales para permitir la correcta interacción entre clases y el acceso seguro a los atributos.

```
def mostrar_nombre(self): # método público
    return self.nombre    # atributo público
```

- **Miembros protegidos** con un guión bajo (_var_protegida).
  - Pueden ser accedidos por la clase y sus subclases.
  - Los **atributos protegidos** se utilizan por defecto. Los **métodos protegidos** para funcionalidad interna a nivel de herencia.

```
def mostrar_informacion(self): # método público
    return f"{self.nombre}, {self._edad} años" # Atr. público y protegido
```

- **Miembros privados** con doble guión bajo (__var_privada).
  - Realmente usado para ocultar un miembro, incluido subclases.
  - No es común en Python y muy rígido con su *name mangling*.

```
def _get_matricula(self):  # método protegido
    return {self.__matricula} # atributo privado
```

Ejemplo completo en Python

```python
class Vehiculo:
    def __init__(self, llave: bool, tipo: str, motor: bool):
        self.llave: bool = llave          # Atributo público
        self._tipo: str = tipo            # Atributo protegido
        self.__motor: bool = motor        # Atributo privado

    def arrancar(self):                   # Método público
        if self.llave and self.__verificar_motor():
            print(f"El {self._tipo} está arrancado")
        else:
            print(f"Falta llave o el motor no está listo")

    def __verificar_motor(self):          # Método privado
        return self.__motor


# Uso
vehiculo = Vehiculo(True, "coche", True)
vehiculo.arrancar()  # Acceso a método público

# Acceso a atributos
print(vehiculo.llave)  # Atributo público, accesible directamente
print(vehiculo._tipo)  # Atributo protegido, accesible pero desaconsejado

# print(vehiculo.__motor) # Error, el atributo privado no es accesible
print(vehiculo._Vehiculo__motor) # Name mangling: acceso no recomendado
```

IMPORTANTE: Cómo usar los modificadores de acceso

Métodos públicos y protegidos/privados

- Los métodos públicos describen qué pueden hacer los objetos de la clase.
  - Son los métodos con los que los objetos interactúan entre sí.
  - El conjunto de métodos públicos componen su contrato público.

```
class Cafetera: ...
    def preparar_cafe(self, cantidad: int):      # Método público
        if self._puede_preparar(cantidad):       # Método protegido
            self._calentar_agua()                # Método protegido
            print(f"Preparando {cantidad} ml de café.")
            self._set_agua(self._agua-cantidad) # Método setter protegido
            self._set_cafe(self._cafe-cantidad) # Método setter protegido
        else:
            print("No hay suficiente agua o café.")

mi_cafetera: Cafetera = Cafetera(500, 300)
mi_cafetera.preparar_cafe(200) # interacción mediante método público
```

- Los métodos protegidos/privados describen cómo lo hacen.
  - Codifican el funcionamiento interno.

```
    def _puede_preparar(self, cantidad: int) -> bool: # Método protegido
        return self._agua >= cantidad and self._cafe >= cantidad

    def _calentar_agua(self):  # Método protegido
        print("Calentando el agua...")
```

IMPORTANTE: Cómo usar los modificadores de acceso

Métodos Getter/Setter

Todo estado (atributo) de un objeto debería de ser protegido/privado

- Diseño correcto de POO: ocultar los detalles del objeto.
- **Métodos Getter/Setter**. Aquellos definidos para permitir el acceso seguro a los **atributos** de una clase.
  - El **método de consulta** get() permite obtener el valor del atributo. Sólo existirá cuando se permita la consulta del atributo.
  - El **método modificador** set() permite establecer el valor de un atributo. Se implementará si se va a permitir la modificación. Este método controlará que la asignación al atributo es coherente, rechazando o alterando el argumento de entrada si fuera necesario.
  - Estos métodos podrían tener niveles de visibilidad diferentes (público para ser usado por todos, protegido para clase/subclases).
  - Una vez definidos, deberían ser usados no sólo por clases externas, sino también por la propia clase (pues mantienen la coherencia y realizan control de errores).
  - Un uso común es definir **métodos Getter/Setter públicos para ofrecer acceso seguro a atributos protegidos/privados**.

Métodos Getter/Setter en Python

Uso de métodos públicos get() y set() para acceder al estado protegido/privado del objeto

```python
class Persona:
    def __init__(self, dni: str, años: int) -> None:
        self._dni: str = dni    # Atributo protegido
        self._años: int = 14    # Atributo protegido

    # Método público: getter para DNI
    # No hay setter pues no se puede cambiar el DNI
    def get_dni(self) -> str:
        return self._dni

    # Método público: getter para años (todos pueden ver los años)
    def get_años(self) -> int:
        return self._años

    # Método protegido: setter para años (clase o sus subclases)
    def _set_años(self, años: int):
        if 0 <= años <= 100:  # Control de valores válidos
            self._años = años

    # Método público: incrementar los años
    def cumple_años(self):
        self._set_años(self._años + 1) # Método del contrato usando el setter
```

Ejercicio

Ejercicio.

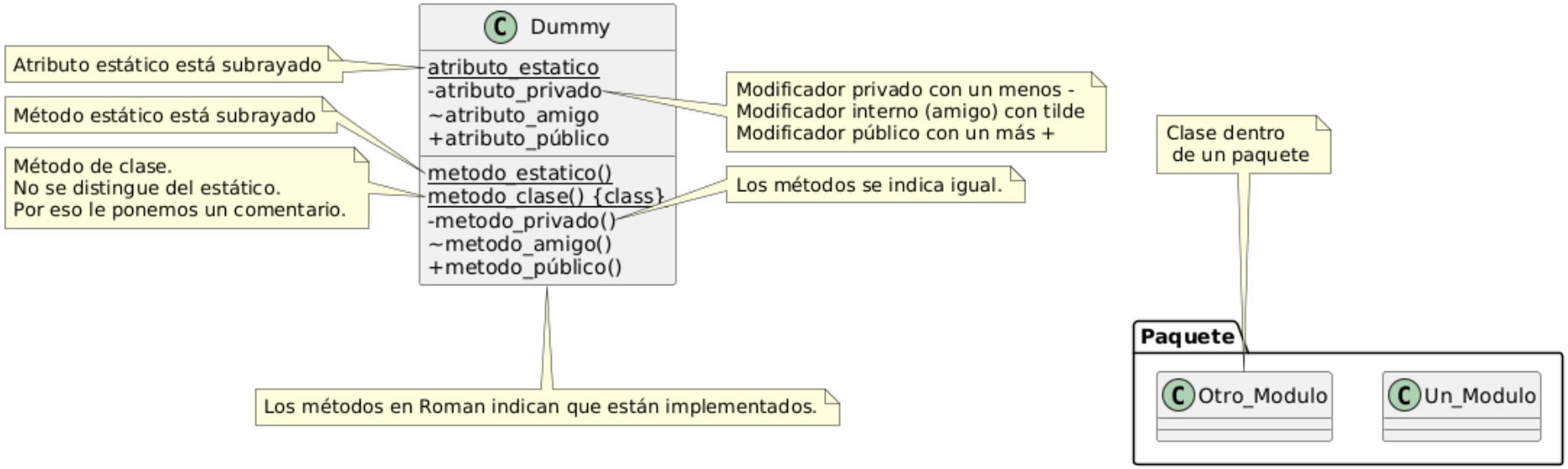
Un personaje se caracteriza por el dinero que posee.

- Se construye suministrando la cantidad de dinero inicial.
- Construye los métodos Getter/Setter sabiendo que un personaje **solo puede añadir/quitar una moneda** cada vez.
- Construye el método para saber si un personaje tiene dinero
- Construye el método por el que otro personaje le dé una moneda de las que tiene.

  ¡Piensa bien qué métodos públicos y protegidos necesitas!

Representación UML

Modificadores de Acceso + Tipos de atributos



UNIVERSIDAD
DE MURCIA

Visibilidad

Módulos, paquetes y espacio de nombres

Paquetes y módulos

Modularidad

- Un programa se divide en partes más pequeñas llamadas **módulos** que facilitan la reutilización y el mantenimiento (**modularidad**), pudiéndose organizar por **paquetes** de manera ordenada.
- Un **módulo** es un fichero que agrupa identificadores relacionados (variables, funciones y clases) que se pueden importar para reutilizar. La variable `__name__` contiene el nombre del módulo.

```python
# módulo: suma.py
def sumar(a: int, b: int) -> int:
    return a + b
```

```python
# módulo principal: calculadora.py
import suma
print(suma.sumar(2, 3))
```

- Un **paquete** organiza una colección de módulos (ficheros) en una jerarquía lógica de carpetas. Para ello hay que crear el fichero `__init__.py` (para nosotros estará vacío).

```
calculadora/          # paquete "calculadora"
    __init__.py       # especifica el paquete "calculadora"
    calculadora.py        # módulo (principal) "calculadora"
    operaciones/      # paquete "operaciones"
        __init__.py   # especifica el paquete "operaciones"
        suma.py           # módulo "suma"
        resta.py          # módulo "resta"
```

Espacio de nombres: namespaces

- Un **namespace** es el entorno abstracto donde quedan definidos los nombres de las variables, funciones y clases.
- Dentro de un *namespace* usamos las variables, funciones y clases sin conflictos de nombres, evitando colisiones.
- En **Python** existen tres tipos de *namespaces*:
  - **Incorporado** (*built-in*): Nombres por defecto de Python (`print()`, `len()`, excepciones, etc.) siempre disponibles.
  - **Global**: contiene los nombres definidos a nivel de un módulo.
    - Los identificadores de los módulos están aislados frente a otros módulos, pero se pueden importar para acceder a ellos.
    - La importación explícita (`import`) permite acceder a los nombres dentro de otro módulo de manera segura.
  - **Local**: nombres definidos dentro de una función o clase.

```python
# modulo_a.py
x = 5
```

```python
# modulo_b.py
x = 10
```

```python
# módulo principal
import modulo_a
import modulo_b
print(modulo_a.x)   # 5
print(modulo_b.x)   # 10
```

Visibilidad de los namespaces en Python

- Una **función** define sus identificadores en su *namespace local*. Además, la función tiene acceso al *namespace global* y *namespace incorporado*.
- Un **módulo** define sus identificadores en el *namespace global*. Puede tener acceso a identificadores de otros módulos mediante la importación (`import`). No tiene acceso a los *namespaces locales* de las funciones pero sí al *namespace incorporado*.
- El **namespace incorporado** define identificadores como `print` o `len` que pueden ser usadas en cualquier lugar del código.

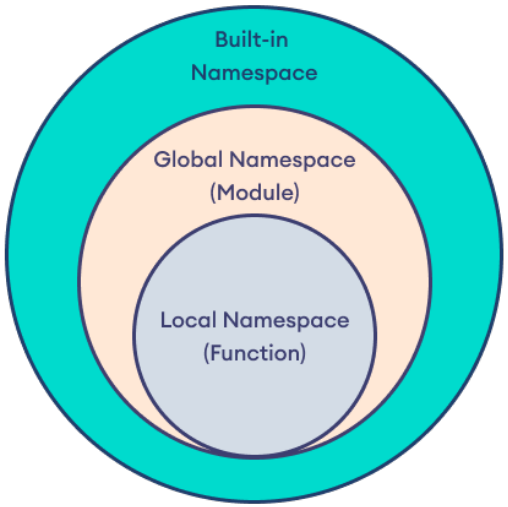


Figura: Ámbito de los *namespaces* en Python

Visibilidad de los namespaces en Python

```python
# --- NAMESPACE GLOBAL DE modulo.py ---
# Identificadores: mensaje, saludar
mensaje = "Hola desde modulo.py"

def saludar():
    return "Saludos desde modulo.py"
```

```python
import modulo # Importamos el módulo

# --- NAMESPACE GLOBAL DE main.py ---
# Identificadores: x, funcion_global, modulo

x = 10

def funcion_global():
    # --- NAMESPACE LOCAL DE funcion_global ---
    # Identificadores: y (diferente al de arriba)

    y = 5; print(f"Variable local y: {y}")  # local

    print(f"Valiable global x: {x}")        # global
    print(modulo.saludar())                 # función saludar() importada

# Llamada a la función print (NAMESPACE INCORPORADO)
print("Llamada a función incorporada")
```

Uso de módulos y paquetes en Python

Ejemplo de uso de módulos

```
import math
print("PI=", math.pi)
```

```
import math as m
print("PI=", m.pi)
```

```
from math import pi, cos
print("cos(PI)=", cos(pi))
```

Ejemplo de uso de paquetes

```
# Usa un modulo del mismo paquete
import unmodulo
# Usa un modulo de otro paquete
import Paquete.unmodulo
# Usa un modulo de un paquete interno
import Paquete.otropaquete.otromodulo

# Usa la función del mismo paquete.
unmodulo.funcA()
# Usa la función de otro paquete.
Paquete.unmodulo.funcB()
# Usa la función del modulo del paquete interno
Paquete.otropaquete.otromodulo.funcC()
```

```
# Usa una función del mismo paquete
from unmodulo import funcA
# Usa una función del modulo de otro paquete
from Paquete.unmodulo import funcB
# Usa una función del modulo de un paquete interno
from Paquete.otropaquete.otromodulo import funcC

funcA()
funcB()
funcC()
```

Modularidad

- Un módulo deber ser Cohesivo.
  - El propósito debe estar bien definido.
  - La cohesión hace más fácil:
    - Entender qué hace una clase y sus métodos.
    - Usar nombres más descriptivos.
    - Reutilizar mejor las clases y métodos.
- Un módulo deber ser Poco Acoplado.
  - El acoplamiento indica la dependencia entre clases.
  - Lo ideal es "independencia" para que una clase conozca lo mínimo esencial de otra clase (lo que indica cohesión).
  - Acoplamiento fuerte implica que las clases relacionadas tienen que conocer detalles unas de otras.
  - El poco acoplamiento hace más fácil:
    - Entender una clase sin leer otras.
    - Cambiar una clase sin afectar a otras.
    - El mantenimiento: se detectan antes los errores.

Principios de la POO

¿Los recuerdas?

- Abstracción[1]. Proceso mental de extracción de las características esenciales de un concepto o proceso descartando los detalles.
- Encapsulación[2]. Proceso por el que se ocultan los detalles de un objeto.
- Jerararquización[3]. Estructurar por niveles (jerarquía) los elementos que intervienen en el proceso.
  - Jerarquía de clasificación (**Herencia**).
  - Jerarquía de composición (**Asociación**).
- Modularidad[4]. Descomposición del sistema en conjunto de módulos poco acoplados (independientes) y cohesivos (con significado propio).

[1] Se trató en el tema anterior con clases y objetos
[2] Se ha estudiado en este tema con la visibilidad de clases
[3] Lo trataremos en el tema siguiente
[4] Se ha estudiado en este tema con la visibilidad de módulos

UNIVERSIDAD
DE MURCIA

Tema 3. Jerarquización

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
13 de octubre de 2024

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

Relaciones

Tipos de Relaciones

Relaciones entre clases

- Cuando se tiene varias clases, pueden existir vínculos entre ellos.
- A los vínculos se llaman relaciones.
- En su versión más sencilla, un objeto manda un mensaje a otro.
- En su versión más compleja, un objeto necesita la información contenida en otro.
- Distinguimos los siguientes tipos de relaciones de menor a mayor dependencia:
  - Relación de uso
  - Asociación
  - Agregación (has-a)
  - Composición (part-of)
  - Herencia (is-a)
- Estos vínculos establecen una relación jerárquica entre clases.
- Cuándo definir uno u otro tipo de relación dependerá del diseño final (interpretación del programador y principios de ingeniería informática).

Relación de uso

- Una **relación de uso** es una relación en la que una clase utiliza objetos de otra clase, pero es una **relación esporádica**.
- Una clase A usa una clase B para que desarrolle un servicio por él.
- Lo más típico es que se pase una instancia de la clase B a un método de la clase A.

```
class B:
    def use(self):
        print('me utilizan')

class A:
    def metodo(self, b):
        b.use()

a = A(); b = B(); a.metodo(b)
```

- **Ejemplos:**
  - *Las personas usan los cajeros (sin que la persona sea cliente del banco)*
  - *Las personas compran en (se relacionan con) los supermercados*
  - *Una persona puede usar un servicio público (vehículos, correos, ...)*
  - *etc ...*

Asociación

- La **relación de asociación** es una **relación de uso** (previamente definido) **estable en el tiempo** pero que puede cambiar.
- No sólo la clase A usa la clase B, sino que un atributo de la clase A es una instancia de la clase B
- La existencia de un objeto de la clase A no depende de la existencia del objeto de la clase B (el atributo puede quedar nulo)

```
class B:
    pass

class A:
    def __init__(self, b):
        # A depende de B
        self.__b = b
```

- **Ejemplos:**
  - *Un cliente web depende de un servidor (cambia en el tiempo)*
  - *Los estudiantes se relacionan con los profesores, y a la inversa (cambian en el tiempo).*
  - *Los propietarios de casas se relacionan con sus aseguradoras (cambian en el tiempo).*
  - *etc ...*

Agregación

- Una **relación de agregación** es un caso de asociación donde hay **más nivel de pertenencia** y es menos probable que cambie en el tiempo.
- Se produce cuando una objeto tiene-un objeto | relación 'has-a' |

```
class B:
    pass

class A:
    def __init__(self, b):
        # A depende de B
        self.__b = b
```

- Al igual que en la relación de asociación, la relación no afecta a sus ciclos de vida: aunque A tiene un miembro del tipo B, si destruyes A sigue existiendo B.
- Los objetos pueden existir independientemente
- **Ejemplos:**
  - *Una factura está asociada a un cliente (y el cliente puede aparecer en varias facturas)*
  - *Una persona puede tener un coche/jersey (y el coche/jersey puede existir sin la persona)*
  - *Una habitación puede tener una o varias sillas (y éstas pueden existir sin la habitación)*

Composición

- La **relación de composición** es más restrictiva que la agregación.
- Se produce cuando una objeto es parte-de otro objeto | relación 'part-of' |
- Esta relación **sí afecta a sus ciclos de vida**: un objeto NO puede existir sin el otro.

A tiene un miembro del tipo B. Si destruyes A, también se destruye B.

Además: si B no está en A, el objeto A está incompleto (sin definir).

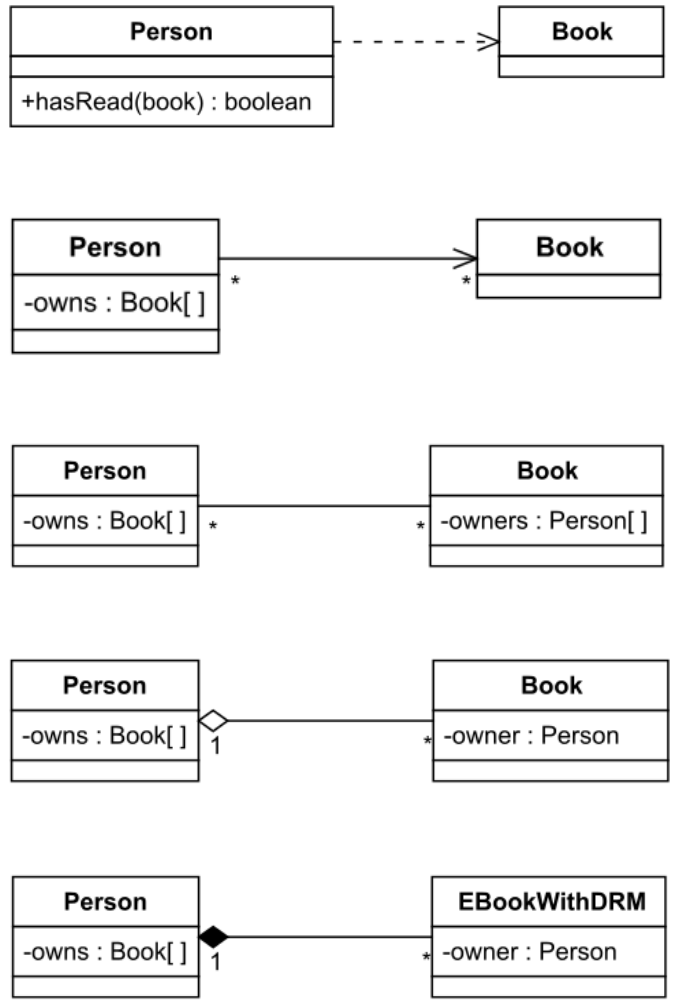
- El atributo de tipo B suele crearse e inicializarse en el constructor de A
- **Ejemplos:**
  - *Una persona tiene corazón (sin él la persona está incompleta).*
  - *Un rabo forma parte de los perros y gatos (y sin él la mascota está incompleta).*
    Si dejan de existir las mascotas, dejan de existir los rabos.
  - *Una biblioteca tiene libros (pero sin éstos, la biblioteca deja de existir).*
  - *Las habitaciones forman parte de un casa (aunque sea una).*
    Sin habitaciones no tiene sentido que eso sea una casa.
    Si deja de existir la vivienda, dejan de existir las habitaciones.
  - *etc ...*

Resumen

- A veces la diferencia entre relaciones no está clara (depende del diseño).
- Estarás en una situación de **relación de uso** cuando un objeto de B no se almacena en ningún atributo de A.
- Si el objeto se almacena en algún campo, un objeto B será un atributo del objeto A.
  - Si no afecta a sus ciclos de vida (destruyes A entonces el objeto B seguirá existiendo):
    - Será una relación de **Asociación** cuando no sea de Agregación.
    - Será una relación de **Agregación** cuando:
      - A tiene o posee otro objeto B, y/o B es parte de A
  - Si sí afectan a sus ciclos de vida (destruyes A entonces el objeto B deja de existir):
    - Será una relación de **Composición**.
- Ejemplo. Imagina un pirata con una pata de palo, espada al cinto y disparando un cañón de su barco pirata:
  - relación de uso con el cañón (no puede llevar el cañón encima)
  - una asociación con la espada (no forma parte del pirata)
  - una pata de palo agregada (sobrevive si matan al pirata)
  - una pierna "normal" y brazos (no sobrevivirán si matan al pirata)

Ejemplos de relaciones entre Persona y Libro

- **Uso**: Persona tiene un método que usa un objeto de la clase Libro.
- **Asociación**:
  - Unidireccional. Una persona tiene el campo owns que almacena una lista de libros que usa. A su vez un libro lo puede usar muchas personas.
  - Bidireccional. Un libro también tiene un campo con los nombres de las personas.
- **Agregación**. Se considera que el libro tiene como único propietario a una persona. De alguna forma, la persona tiene "su ejemplar".
- **Composición**. Se considera que el libro tiene DRM (es la única persona que puede usarlo).



Agregación vs Composición (con código)

- Observa bien las diferencias.
- La mesa tiene dos referencias, pero cada habitación tiene solo una.

```
class Habitacion:
    ...
    def setMesa(self, mesa: Mesa):
        """
        AGREGACIÓN. La mesa ya existe.
        Sintácticamente igual que ASOCIACIÓN.
        """
        self._mesa: Mesa = mesa;

class Casa:
    def __init__(self, numHabitaciones: int):
        """
        COMPOSICIÓN. Los objetos "part-of" se crean en el constructor
        """
        self._habitaciones: list[Habitacion] = []
        for i in range(0, numHabitaciones):
            self._habitaciones.add(Habitacion())
            # Se crean las habitaciones
```

UNIVERSIDAD
DE MURCIA

Relaciones

Delegación

Delegación: ¡Debe aplicarse siempre!

- Delegación: Cuando una clase contiene una o más instancias de otras clases, entonces la clase **delega** su funcionalidad a los atributos.
- Un objeto recibe una petición y **delega** la ejecución del método a otros objetos.
- Es una buena costumbre que la acción y la acción delegada tengan **el mismo nombre**.

Ejemplo. La clase Rectángulo tiene el método *trasladar* una distancia que, a su vez, delega en el método *trasladar* de la clase Punto que se encarga de modificar las coordenadas *x* e *y* del punto *origen*.

```
class Punto:
    ...
    def trasladar(self, distancia: Punto):
        self.set_x(self.get_x() + distancia.get_x())
        self.set_y(self.get_y() + distancia.get_y())

class Rectangulo:
    def __init__(self, largo: float, ancho: float, origen: Punto):
        self._largo = largo
        self._ancho = ancho
        self._origen = origen

    def trasladar(distancia: Punto):
        self._origen.trasladar(distancia) # Delegación
```

UNIVERSIDAD
DE MURCIA

Relaciones

Clonación/Copia de un Objeto

¿Cómo clonar un objeto?

- Cuando se tienen dos objetos, la asignación obj1=obj2 produce aliasing sobre el mismo objeto. No se copia el objeto (sino su referencia).
- Hacer una copia conlleva crear una nueva instancia manteniendo las relaciones de asociación, agregación y composición que tiene el objeto.
- **Copia Superficial**
  - Crea una nueva instancia de la misma clase, clonando los valores de los atributos inmutables (numéricos, caracteres, booleanos...) pero referenciando a los atributos mutables (listas, diccionarios, conjuntos y objetos).
  - Es decir, existe **aliasing** en los atributos mutables que quedan compartidos entre los dos objetos clonados.
  - Hay que llevar cuidado, esos atributos con aliasing se pueden modificar a través de ambos objetos.
- **Copia Profunda**
  - Además de la copia superficial, se clonan también los atributos mutables.
  - Se generan dos objetos completamente independientes, sin aliasing.
  - Puede llegar a ser muy compleja.

Clonación de objetos en Python

- El módulo copy permite hacer copias.
- Tiene los siguientes métodos:
  - copy.copy(x) para realizar una copia superficial de x.
  - copy.deepcopy(x[,memo]) para realizar una copia profunda de x.

```
import copy

class Persona:
    def __init__(self, nombre: str, amigos: list[str]) -> None:
        self._nombre = nombre  # atributo inmutable (string)
        self._amigos = amigos  # atributo mutable (lista)

persona1 = Persona("Juan", ["Carlos", "Ana"])
persona_copia_superf = copy.copy(persona1) # Copia superficial
persona_copia_superf._amigos.append("Luis") # Modificar lista (ALIASING)
print(persona1._amigos)                    # ["Carlos", "Ana", "Luis"]
print(persona_copia_superf._amigos)  # ["Carlos", "Ana", "Luis"]
# PRUEBA A MODIFICAR EL NOMBRE DE LA PERSONA...

persona_copia_prof = copy.deepcopy(persona1) # Copia profunda
persona_copia_prof._amigos.append("Miguel")
print(persona1._amigos)                # ["Carlos", "Ana", "Luis"]
print(persona_copia_prof._amigos) # ["Carlos", "Ana", "Luis", "Miguel"]
```

- Para que una clase tenga su propia implementación de copy, puede definir métodos mágicos __copy__() y __deepcopy__().

UNIVERSIDAD
DE MURCIA

Relaciones

Ejercicios de Relaciones

Ejercicios - I

Ejercicio.

Programa las clases y relaciones necesarias:

- Un usuario tiene dinero en efectivo y una tarjeta. Esta última tiene dueño y saldo. Hay un cajero que permite sacar una cantidad dada de dinero en efectivo con la tarjeta si esta tiene saldo suficiente.
- Una familia se entiende como una lista de personas. Las personas tienen un nombre y pertenecen a una familia, uniéndose a la misma cuando nacen.
- Triángulo cerrado en el plano, definido por tres puntos. En este programa, los puntos sólo pueden formar parte de un triángulo.

Ejercicios - II

Ejercicio.

Si el estado de una persona queda determinado únicamente por su nombre y su pareja:

- ¿cómo se construye que una persona A es soltera?
- Y si A es pareja de B, ¿cómo se pueden construir A y B?

Ejercicios - III

Ejercicio.

Un usuario sabe que está inscrito en una biblioteca y en una tienda de libros. Tanto la biblioteca como la tienda contienen una cantidad ingente de libros y tienen registrado al usuario como cliente.Además, todo libro agrega bien la biblioteca en la que se encuentra depositado o la tienda en la que está disponible.

Programa clases y métodos para que un usuario consulte la disponibilidad de un libro

Ejercicios - IV

Ejercicio.

En un videojuego de acción en tercera persona los jugadores controlan a un personaje. El personaje debe tener siempre su arma, que puede disparar si esta tiene munición. También tiene la capacidad de recoger objetos del entorno del juego que se añadirán a su inventario.

Programa las clases y métodos que permiten simular las acciones de disparar y recoger objetos.

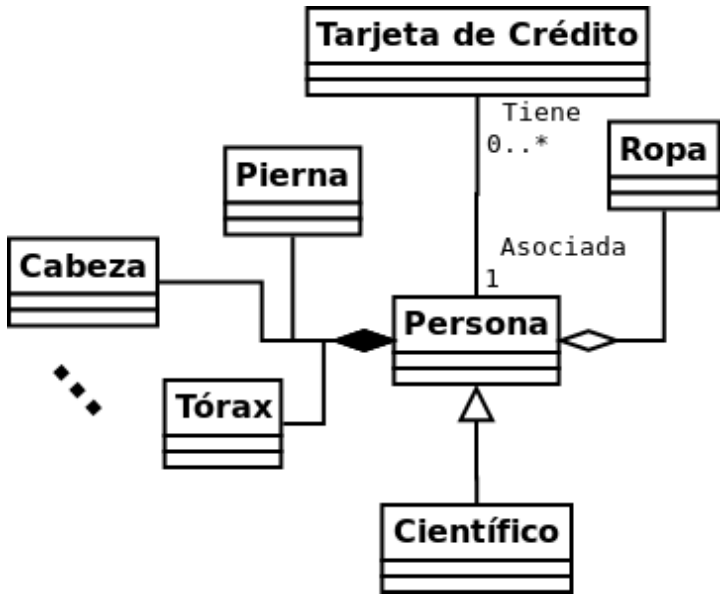
UNIVERSIDAD
DE MURCIA

Herencia

Herencia

- Ya sabes que las clases se pueden relacionar entre ellas por asociación:
  - La asociación de agregación se corresponde con `has-a`
  - La asociación de composición se corresponde con `part-of`.
- La asociación de **herencia** se corresponde con `Is-a`
- Se usa cuando una clase es una generalización de otra o, si se prefiere, cuando la otra es un caso particular de la una.

Ejemplo.

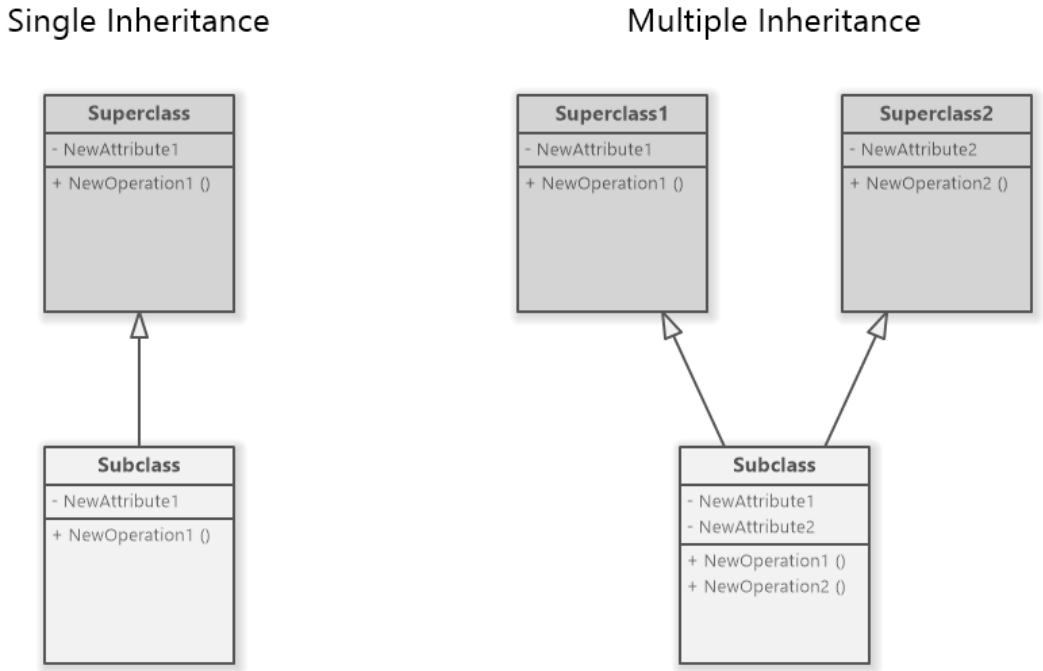


Subclases

- La herencia crea clases nuevas a partir de una clase existente.
  - La clase nueva es un caso particular de la clase que ya existe.
- A la clase nueva se le denomina subclase y a la existente superclase.
  - Responden a una especialización y a una generalización.
- Nombres alternativos
  - Para la **subclase**: clase hija, derivadas o subtipos.
  - Para la **superclase**: clase padre o base.
- La herencia es el proceso por el que una clase hija reconoce a los miembros de la clase padre.
  - Los miembros reconocidos se llaman miembros heredados.
  - No tienen que reconocerse todos (sólo los que se indiquen).
- En la subclase se DEBEN definir nuevos miembros (atributos y/o métodos)
  - Por eso se dice que la clase derivada extiende a la clase base.
  - Los nuevos miembros serán desconocidos por la clase padre.

Tipos de Herencia

- Hay **herencia simple** cuando la subclase solo puede heredar de una clase padre (Java, C#, etc.).
- Hay **herencia múltiple** cuando la subclase hereda de dos o más padres (C++, Python, etc.).



- Un objeto con "tipo de dato" de una subclase también es un "tipo de dato" de las superclases. *P.e. si Hombre es subclase de Omnívoro y Animal, entonces un objeto de tipo Hombre también es de tipo Omnívoro y Animal.*

Herencia en Python

- En Python la herencia es múltiple.
- La clase hija hereda todos los **miembros públicos y protegidos** de la clase padre. Realmente los privados también, pero se ocultan mediante el *name mangling*.
- Definición de subclases:

  class Subclase (Superclase1, SuperClase2, ...)

- Una subclase puede (debe) llamar al __init__() del padre con super().
  - super() retorna un objeto "proxy"que representa a la clase padre.

```
class Mascota: # Una clase
    def __init__(self, nombre: str):
        self._nombre: str = nombre

    def get_nombre(self) -> str:
        return self._nombre

class Perro(Mascota): # Subclase
    def __init__(self, nombre: str):  # Invoco al constructor padre
        super().__init__(nombre)
        self._nombre = "Perro " + self.get_nombre()  # modificación

perro = Perro("Toby")
print (perro.get_nombre()) # Imprime "Perro Toby"
```

Herencia y Sobreescritura de métodos

- En una superclase se indicará siempre qué miembros serán heredados.
  - Si bien en Python se hereda todo, son los miembros protegidos y públicos de la superclase los únicos que deben ser accedidos.
- Si una subclase hereda un método puede hacer dos cosas:
  - Usar el método de la superclase como si fuera suyo.
  - Sobreescribir (*Overriding*) el método para tener un comportamiento diferente (pudiendo usar super() para "extender" el método padre).

La sobreescritura de un método es construir un nuevo método con la misma declaración (signatura) pero donde cambia la definición. Si no se sobreescribe, se mantiene la definición del método padre.

```
class Mascota: # Una clase
    def __init__(self, nombre: str):
        self._nombre: str = nombre

    def get_nombre(self) -> str:
        return self._nombre

class Perro(Mascota): # Subclase, constructor se hereda si no se redefine

    def get_nombre(self) -> str: # sobreescritura de get_nombre
        return f"Perro se llama {self._nombre}" # redefinición completa
        return f"El perro se llama {super().get_nombre()}" # o extensión
```

Herencia y Ocultación de atributos

- Si una subclase hereda un atributo puede hacer dos cosas:
  - Usar el atributo de la superclase como si fuera suyo.
  - Ocultar el atributo heredado mediante una nueva definición.
    La ocultación de un atributo es definir una atributo con el mismo identificador que el atributo de la superclase. Si no se oculta/redefine, se mantiene el valor del atributo padre.
- Un atributo se comparte hacia abajo por la jerarquía de herencia. En el constructor, con super() podemos heredar los atributos de la clase padre.

```
class Mascota:
    def __init__(self, sonido: str ="Sonido genérico"):
        self._sonido: str = sonido

class Perro(Mascota):
    def __init__(self, tipo: str):
        super().__init__(sonido="Guau") # Usar el atributo padre sonido
        # self._sonido = "Guauguau"     # Ocultaría el atributo padre
        self._tipo = tipo         # Nuevo atributo, solo en subclase

perro = Perro(tipo="Chihuahua")
print(perro._sonido)     # Imprime "Guau"
print(perro._tipo)       # Imprime "Chihuaua"

animal = Mascota()       # No tiene tipo
print(animal._sonido)    # Imprime "Sonido genérico"
```

Relación entre clases y subclases

- RECUERDA: Una subclase es la particularización de la clase padre.
- La **subclase reconoce a los miembros de la superclase** y añade (o redefine) nuevos miembros.
  - Para poder acceder a los métodos de la superclase se deberá usar la función super().
  - Es común que en la **sobreescritura de un método** `metodo(...)`, la primera instrucción invoque al método padre `super().metodo(...)` y las siguientes instrucciones modifiquen o extiendan con nueva funcionalidad lo definido en la clase padre.
  - Es común que en la **ocultación de atributos**, la clase hija `__init__(...)` invoque al constructor padre `super().__init__(...)` para inicializar los atributos heredados y después extender con nuevos atributos (u ocultar los heredados). Es decir, cada __init__():
    - pase a super() los argumentos no son propios,
    - se creen nuevos atributos con el resto de argumentos.
- La **clase padre no puede acceder a los miembros de la subclase**.
  - obj.__dict__ es un diccionario que muestra los miembros propios.
  - `dir`(obj) es una lista que los miembros del objeto, tanto propios como recursivamente heredados.

Relación entre clases y subclases

Ejemplo

```
class Mascota: # Una clase

    def __init__(self, nombre):
        self._nombre = nombre

    def get_nombre(self):
        return self._nombre

class Perro(Mascota): # Subclase sin constructor, usará el de Mascota
    cardinal: int = 0

perro = Perro("Toby")       # en Mascota se asigna "Toby" a self._nombre
print(perro.get_nombre())   # se muestra el valor de self._nombre

### Miembros únicos en las clases y el objeto perro
print([ m for m in Mascota.__dict__ if not m.startswith('__')],
      [ m for m in Perro.__dict__ if not m.startswith('__')],
      [ m for m in perro.__dict__ if not m.startswith('__')])
# ['get_nombre'] ['cardinal'] ['_nombre']

### Miembros reconocidos en las clases y el objeto perro
print([ m for m in dir(Mascota) if not m.startswith('__')],
      [ m for m in dir(Perro) if not m.startswith('__')],
      [ m for m in dir(perro) if not m.startswith('__')])
# ['get_nombre'] ['cardinal', 'get_nombre']
# ['_nombre', 'cardinal', 'get_nombre']
```

Relación entre clases y subclases

Un ejemplo más claro donde los atributos se añaden al objeto

```python
class Mascota: # Una clase

    def __init__(self, nombre):
        self._nombre = nombre

    def get_nombre(self):
        return self._nombre

class Perro(Mascota): # Subclase

    def __init__(self, nombre):

        # Añade atributo _nombre
        super().__init__(nombre)

        # Modifica atributo _nombre
        self._nombre = "EL " + nombre

mascota = Mascota("Toby")
print(mascota.get_nombre()) # Toby

perro = Perro("Toby")
print(perro.get_nombre()) # EL Toby
```

El atributo **protegido** self._nombre es compartido en las dos clases.

```python
class Mascota: # Una clase

    def __init__(self, nombre):
        self.__nombre = nombre

    def get_nombre(self):
        return self.__nombre

class Perro(Mascota): # Subclase

    def __init__(self, nombre):

        # Añade atributo en el padre
        super().__init__(nombre)

        # Nuevo atributo
        self.__nombre = "EL " + nombre

perro = Perro("Toby")
print(perro.get_nombre()) # Toby

# Equivale a lo siguiente (NO HACER)
print(perro._Mascota__nombre)
```

Hay un atributo **privado** __nombre en cada clase. Habría que sobreescribir el método get_nombre().

Sobreescritura sobre la clase *Object*

- En Python realmente hay una clase raíz Object de la que heredan implícitamente todas las clases existentes o que definimos.
- Cuando implementamos métodos mágicos, realmente estamos sobreescribiendo métodos ya definidos en la clase raíz Object:
  - *__init__ ()*. El método de inicialización de variables de un objeto.
  - *__str__ ()*. El modo en el que un usuario vería la información de la clase. Retorna una cadena "informal".
  - *__gt__ ()*, *__ge__ ()*, *__lt__ ()*, *__le__ ()*: Métodos que define las desigualdades lógicas al comparar dos objetos con $>$, $>=$, $<$ y $<=$.
  - __eq__(self , objeto). Método que define el operador de igualdad ($==$)
  - etc...
- Todos estos métodos ya tienen una implementación previa en la clase padre Object, lo que hacemos en los métodos mágicos es sobreescribirlos para personalizar su funcionamiento.

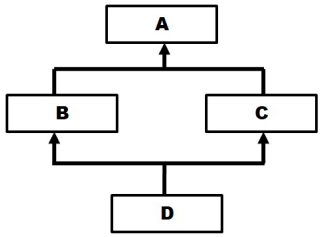
UNIVERSIDAD
DE MURCIA

Herencia

Problema del diamante

Sobreescritura: Problema del diamante

- En herencia múltiple surge el problema de la estructura del diamante.



- Si B y C heredan y sobreescriben un método de A, y la clase D lo hereda (sin sobreescribir) de B y de C, ¿la clase D utiliza el método heredado de B o de C?
- Python usa el Method Resolution Order (MRO): crea una lista recursiva de clases, de izquierda a derecha y de abajo a arriba (D, B, A, C, A), eliminando las clases repetidas salvo la última ocurrencia[1]:
  - El orden es aquel especificado en la definición de la subclase (class Subclase (Superclase1, SuperClase2, ...))
  - La clase raíz siempre será Object.
  - El **orden de resolución** del método es: D, B, ~~A~~, C, A, Object.
- El árbol de ancestros puede obtenerse con el atributo de clase __mro__. Si un método no está en la superclase, se pasa a la siguiente (__mro__).

[1] Para no visitar antes la superclase (A) de una subclase (C)

Problema del diamante

Method Resolution Order (MRO)

Python resuelve super() siguiendo el árbol de ancestros definido por __mro__ del **objeto invocador**.

```
class A:
    def __init__(self):
        print("A")

class B(A):
    def __init__(self):
        print("B")
        super().__init__()

class C(A):
    def __init__(self):
        print("C")
        super().__init__()

class D(B,C):
    def __init__(self):
        print("D")
        super().__init__()

a = A()   # imprime A
b = B()   # imprime B, A
c = C()   # imprime C, A
d = D()   # imprime D, B, C, A
```

```
class A:
    def __init__(self):
        print("A")

class B(A):
    None

class C(A):
    def __init__(self):
        print("C")

class D(B, C):
    def __init__(self):
        print("D")
        super().__init__()

a = A()   # imprime A
b = B()   # imprime A
c = C()   # imprime C
d = D()   # imprime D, C
```

- ¿Qué pasaría si C tampoco tuviera __init__()?
  D imprimiría D y A.
- ¿Puede usar D el método de A sin pasar por B/C?
  Sí, usando super(C, self).metodo() o A.metodo(self).metodo(). **NO HACER.**

UNIVERSIDAD
DE MURCIA

Herencia

Polimorfismo

Polimorfismo

- Dada una clase padre y dos clases hijas, éstas pueden heredar un método que pueden sobreescribir (para reemplazarlo o refinarlo).
- Cada clase tiene el *mismo método*, pero lo ejecutan de diferente forma.
  - No confundir con la sobrecarga, donde un mismo método puede tener diferentes parámetros para comportamientos diferentes.
- El Polimorfismo es la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de clases diferentes.

```
class Animal:
    def sonido(self):
        return "El animal hace ruido"

class Perro(Animal):
    def sonido(self):
        return "El perro ladra" # reemplazo

class Gato(Animal):
    def sonido(self):
        return "El gato maúlla" # reemplazo

animales: list[Animal] = [Perro(), Gato()]

for animal in animales:
    print(animal.sonido()) # poliformismo del método sonido
```

Métodos para identificar la clase o subclase en Python

Un objeto no pertenece a varias clases, pero se comporta como si fuese también del tipo de las clases predecesoras.

- `type(obj)`. Devuelve la **clase exacta** de un objeto.

```
class Animal:
    pass

class Perro(Animal):
    pass

mi_perro = Perro()
print(type(mi_perro) == Perro)  # True
print(type(mi_perro) == Animal) # False
```

- `isinstance(obj, clase)`. Verifica si un objeto es **instancia de una clase o subclase**.

```
print(isinstance(mi_perro, Perro))   # True
print(isinstance(mi_perro, Animal))  # True
```

- `issubclass(clase1, clase2)`. Verifica si **una clase es subclase de otra**.

```
print(issubclass(Perro, Animal))  # True
print(issubclass(Perro, Gato))    # False
```

UNIVERSIDAD
DE MURCIA

Herencia

Uso de la herencia

Cuándo usar herencia

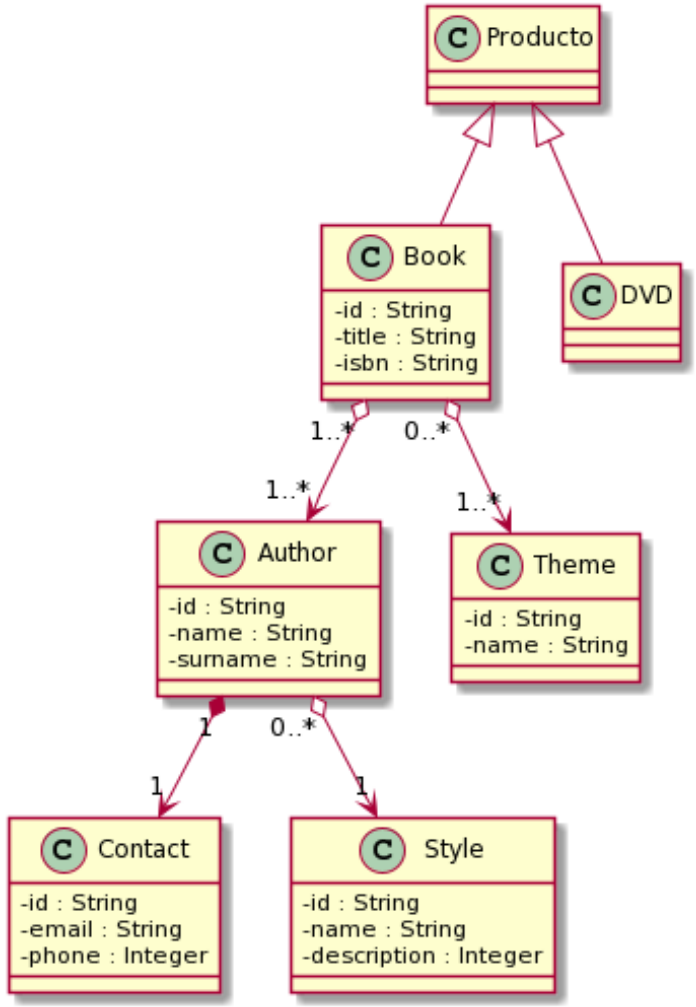
- Usa herencia cuando
  - Relaciones ES-UN. Si se rompen por algún motivo... ¡Mal asunto!
  - TODOS los métodos públicos de la clase A lo son también de la clase B: (1) La subclase B está siempre basada en la superclase A y (2) la implementación de la superclase A es necesaria para B.
  - La subclase es candidata a
    - sólo añadir nueva funcionalidad (nuevos métodos/atributos)
    - no sobrescribir nada: dejarlo como está.
- Principios **SOLID**:
  - **S - Responsabilidad Única:** Cada clase debe tener una sola función o responsabilidad.
  - **O - Abierto/Cerrado:** Puedes extender el comportamiento de las clases sin modificar el código existente.
  - **L - Sustitución de Liskov:** Las subclases deben poder reemplazar a la clase base sin alterar la funcionalidad.
  - **I - Segregación de Interfaces:** Las clases deben implementar solo los métodos que necesitan y no depender de métodos innecesarios.
  - **D - Inversión de Dependencia:** Las superclases no deben depender de detalles específicos de implementación (importancia de clases genéricas, abstractas o interfaces). La lógica específica debe ir en las subclases.

Cuándo NO usar herencia

- **No uses herencia** cuando cause más desventajas que beneficios:
  - Cuando el acoplamiento entre clases es demasiado fuerte.
  - Cuando las clases base crecen y requieren constantes modificaciones (un pequeño cambio afecta a muchas clases).
  - Cuando las subclases sobrescriben demasiados métodos (esto puede indicar que no se trata de una relación de herencia).
  - Cuando las subclases heredan métodos que no necesitan.
  - Cuando la superclase es heredada por solo una subclase.
- **Alternativa**: Asociación, agregación o composición:
  - En lugar de que B herede de A, A puede estar asociada a B. Esto es una relación diferente con cambios de diseño importantes.
  - Modelan una relación con menor acoplamiento.
  - Los cambios en una clase afectan mínimamente a otras clases.
  - No tendrás beneficios como la reutilización o polimorfismo.
- ¿Qué pasa si quiero **objetos del mismo tipo pero sin acoplamiento**?
  ¡Usa interfaces! Lo explicaremos en el próximo tema.

Ejemplo con diferentes tipos de relaciones entre clases

Los libros y los DVDs son productos culturales.
En concreto, los libros tienen un título y un ISBN. Están asociados a un conjunto de temas (un tema puede estar asociado a muchos libros). Y también están asociados a un listado de autores (un autor podría asociarse a otros libros).
Cada autor tiene un nombre y su apellido. Se compone con sus datos de contacto (un email y un teléfono), que se destruyen cuando el autor muere. Un autor se asocia con un estilo (que puede ser utilizado por muchos autores). De un estilo se guarda el nombre y la descripción.



Ejercicios

Ejercicio.

En un sistema de simulación están los **agentes** que permiten **decidir** y aquellos que además se pueden **mover**.

Ejercicio.

Todo **cliente** se caracteriza por tener un DNI y una cuenta bancaria, que puede ser de ahorro o de crédito. Si es de ahorro, genera unos intereses; pero si es de crédito permite tener un depósito. Una cuenta bancaria se crea con un saldo inicial. En toda cuenta se puede depositar y retirar dinero. Un DNI consta de un identificador junto con el nombre, dirección y edad al que pertenece.

Ejercicios

Ejercicio.

Toda **alarma** tiene un **umbral** de sensibilidad de intrusos. También consta de un **sensor** al que consulta y que le indica cual es el valor actual de intrusión. En el caso de que se supere el umbral, puede ocurrir lo siguiente.
Si es una alarma **sonora**, pondrá en marcha un timbre incorporado que se puede activar o desactivas. Pero si es una alarma **luminosa**, encenderá una luz. En el caso de que sea **sonora y luminosa** hará las dos cosas.

Ejercicio.

Para el ejercicio anterior de la alarma ¿qué modificaciones tendrías que hacer para que una alarma completa encendiera la luz ante cierto nivel de intrusión, pero que hiciera sonar también el timbre si el nivel fuera aún mayor?

UNIVERSIDAD
DE MURCIA

Tema 4. Clases Abstractas e Interfaces

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
20 de octubre de 2024

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

Clases Abstractas

Clases Abstractas

- La **abstracción** es el proceso por el cual extraemos las características esenciales de un objeto, tanto en atributos como en comportamiento.
- Se enfoca en lo que hace un objeto, no en cómo lo hace.
- **Ejemplo**: Podemos hablar del concepto de "Vehículo" (sin especificar si es un coche, moto, etc.), donde sabemos que tiene ciertas características (ruedas, motor) y ciertos comportamientos (arrancar, detenerse), pero los detalles concretos varían.
- Una **clase abstracta** representa objetos que tienen algunas características y funcionalidades conocidas, pero no están completamente definidos.
- Las clases abstractas contienen métodos que **pueden estar declarados pero no definidos**.
- **Ejemplo**: Una clase abstracta `Vehículo` puede tener métodos como `acelerar()` o `frenar()`, pero no se especifica cómo funcionan.

Clases Abstractas

- Un método abstracto es aquel que tiene declaración pero no está definido. Una clase abstracta es la que tiene (al menos) un método abstracto.
- Como una **clase abstracta no está completamente definida**, no se puede instanciar. Existirán subclases de una clase abstracta que implementarán los métodos abstractos.
- **Ejemplo**: No podremos tener objetos de la clase abstracta `Vehículo` (pues tiene métodos abstractos `acelerar()` o `frenar()` sin definir), pero sí de subclases como `Coche` o `Moto` que definirán dichos métodos.
- Una subclase puede ser abstracta si la clase padre lo es.
- Por tanto, tendremos un método (abstracto) que se ejecutará de forma diferente en cada subclase (**polimorfismo**):
- Una clase abstracta puede tener constructores:
  - Pero no sirven para construir instancias (por definición de clase abstracta).
  - Las subclases no abstractas deberán de sobreescribir el constructor explícito y llamar al constructor del padre con super().

Ejemplo de Clase Abstracta

Ejemplo.

- La clase Animal tiene los métodos caminar() o comer() pero no se sabe cuáles son las acciones concretas que deben realizarse.
- Por tanto, la clase Animal **debe de ser** abstracta.
- Las clase Ave y Mamífero son subclases de Animal.
- Ave y Mamífero heredan el método caminar():
  - Una ave camina con **dos** patas pero un mamífero con **cuatro**.

```
from abc import ABC, abstractmethod

class Animal(ABC):

  @abstractmethod
  def caminar(self):
    pass

class Mamifero(Animal):

  def caminar(self):
    print("... con 4 patas")

class Ave(Animal):
  pass
```

```
perro = Mamifero()
perro.caminar()
gallo = Ave()
```

Si una clase hija no define el método abstracto sigue siendo una clase abstracta.

UNIVERSIDAD
DE MURCIA

Duck Typing

Duck Typing en Python

- **Duck Typing** en Python:
  - *"Si camina como un pato y suena como un pato, probablemente sea un pato"*
  - Si un objeto tiene los métodos que se esperan (por ejemplo, `volar()`), podrá ejecutarse independientemente de su tipo.

```
class Pato:
    def volar(self):
        print("El pato vuela.")

class Avion:
    def volar(self):
        print("El avión vuela.")

...
objeto.volar() # No importa el tipo, solo que exista el método
```

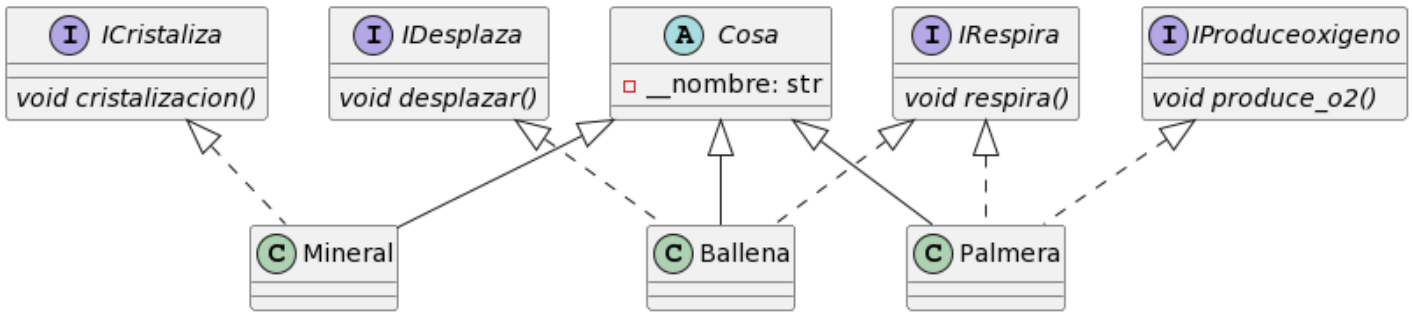
- En Python, el **Duck Typing** se basa en la presencia de métodos sin forzar a que el objeto tenga un tipo concreto.
  - Los lenguajes de programación con *tipado nominal* (como Java) verificaría si el objeto es de una clase antes de ejecutar el método.
- Si queremos forzar o asegurar que una clase implemente ciertos métodos, se deben usar **interfaces formales**.

UNIVERSIDAD
DE MURCIA

Interfaces

Interfaces

- Una **interfaz** es una abstracción que se centra **solo en la funcionalidad** y define un tipo de objeto.
- Contiene solo **métodos**, no incluye atributos o datos. Es un conjunto de acciones que los objetos que la implementan deben poder realizar.
- Los métodos de una interfaz son necesarios para que los objetos que la implementen cumplan con los **requisitos mínimos de funcionalidad**.
- Los objetos que implementan una interfaz son considerados del tipo definido por la interfaz. Esto se conoce como *subtipado estructural*.
- Las interfaces añaden una **especialización** o una **extensión** a los objetos, permitiéndoles ofrecer funcionalidad adicional.
- Cada objeto puede implementar la interfaz de manera diferente:
  - Los métodos de las interfaces son abstractos (sin implementación por defecto) y polimórficos (pueden comportarse de manera distinta).



Interfaces formales en Python

- En Python, se pueden usar **interfaces formales** para definir un conjunto obligatorio de métodos.
- Python es demasiado versátil para trabajar con registros de interfaces (`https://realpython.com/python-interface/`).
- Estas interfaces formales se definen usando la clase base 'abc.ABC':

```python
from abc import ABC, abstractmethod

# Definimos la interfaz formal "Volador"
class Volador(ABC):

    @abstractmethod
    def volar(self):
        """Método que debe implementarse."""
        pass
```

- Se dice que **una clase implementa una interfaz** cuando se desea que tengan obligatoriamente todos los métodos declarados en dicha interfaz.
  - Si quiere que mi clase tenga la funcionalidad de Volador, implemento dicha interfaz para definir métodos personalizados.
  - No tengo tanto acoplamiento como en herencia, donde las superclases ya definen todo el estado y comportamiento de las subclases.

Interfaces formales en Python

- Para que un objeto implemente una interfaz formal, la clase hereda directamente de la interfaz.
  - Al ser métodos abstractos, están obligados a definirlos.
  - Una clase también es del tipo de la interfaz que implementa.

```
# Clase Pato que implementa la interfaz Volador
class Pato(Volador):

    def volar(self):
        print("El pato está volando.")

# Clase Avion que implementa la interfaz Volador
class Avion(Volador):

    def volar(self):
        print("El avión está volando.")

pato = Pato()    # tipo Pato y también Volador
pato.volar()

avion = Avion() # tipo Avion y también Volador
avion.volar()

print(isinstance(pato, Volador))    # Salida: True
print(issubclass(Avion, Volador))   # Salida: True
```

UNIVERSIDAD
DE MURCIA

Clases Abstractas vs Interfaces

Intefaces vs Clases Abstractas

- Una clase abstracta se caracteriza por:
  - **Constructores**: Tiene al menos uno (el implícito)
  - **Atributos de clase**: Se admiten (con cualquier modificador de visualización)
  - **Atributos de instancia**: Se admiten (con cualquier modificador de visualización).
  - **Métodos estáticos, de clase o instancia**:
    - Pueden tener signatura y cuerpo
    - **Al menos uno** tendrá el modificador @abstractmethod que tendrá signatura pero no cuerpo.
    - Cualquier modificador de visualización.
  - **Sobreescritura de métodos:** está permitida (y obligatoria en al menos un caso).
  - Admiten herencia.

    En esto casos se habla de subclase y de superclase.

Intefaces vs Clases Abstractas

- Una interface se caracteriza por:
  - **Constructores**: NO tiene.
  - **Atributos de clase**: Se admiten.
  - **Atributos de instancia**: NO tiene. Es decir, no existen atributos de objeto y son solo de clase. Algunos lenguajes relajan esto.
  - **Métodos estáticos, de clase o instancia**:
    - Contendrán solo la signatura, sin cuerpo.
    - Tendrán el modificador @abstractmethod.
    - Todos los métodos serán públicos.
  - **Sobreescritura de métodos: Todos** se deben sobreescribir.
  - Admiten herencia.

    En esto casos se habla de subinterface y de superinterface.

Intefaces vs Clases Abstractas

	Clase Abstracta	**Interface**
Constructor	Sí	**No**
Atributos de clase	Sí	Sí
Atributos de instancia	Sí	**No**
Métodos	Al menos un abstracto	**Todos abstractos**
Sobreescritura	Sí	Sí. **Todos**.
Herencia	Sí	Sí

UNIVERSIDAD
DE MURCIA

Polimorfismo

Polimorfismo de un objeto

Ahora, un mismo **objeto** puede ser tratado como cualquiera de sus tipos, según lo necesite el contexto:

- **Tipo original**: Tipo de la clase concreta con la que se creó el objeto.
- **Tipos heredados**: El objeto también es del tipo de todas las clases de las que hereda, directa o indirectamente.
- **Interfaces**: Si implementa interfaces, el objeto es de esos tipos también.

```
class Volador(ABC):
    @abstractmethod
    def volar(self):
        pass

class Animal:
    def respirar(self):
        print("El animal respira.")

class Pato(Animal, Volador):
    def volar(self):
        print("El pato vuela.")

pato = Pato()

print(isinstance(pato, Pato))     # True (tipo original)
print(isinstance(pato, Animal))   # True (superclase)
print(isinstance(pato, Volador)) # True (interfaz)
```

UNIVERSIDAD
DE MURCIA

Ejercicios

Ejercicio

Ejercicio.

Los artefactos dispone de un motor capaz de indicar el número de revoluciones al que va. Existen muchos tipos de motores cada uno con distintos tipos de atributos. ¿Cómo modelar entonces la situación?

Ejercicio.

1. Implementa los métodos Getter/Setter en la clase SerieTV con el atributo número de episodios y clase VideoJuego con el atributo horas estimadas de juego.
2. Se considera ahora que ambas clases pueden ser **productos prestados**. ¿qué se debería de hacer para que tengan las siguientes acciones?
   - entregar(): cambia el atributo prestado a true.
   - devolver(): cambia el atributo prestado a false.
   - esEntregado(): devuelve el estado del atributo prestado.
3. Se considera ahora que son **comparables**: Si dos videojuegos tienen el mismo número de horas estimadas se consideran que son iguales. Si dos series de tv tienen el mismo número de episodios se consideran que son iguales.
   Ten en cuenta que pueden existir otros tipos de ocio que no pueden ser prestados, como los canales de streaming, pero si pueden ser comparables, por ejemplo, por el precio.

Ejercicio

Ejercicio.

Cada recurso particular tiene un nombre y un precio; pero no se pueden construir instancias de esta clase.

Los recursos forman los grupos de los coches, el de la carne, o el del pan. Son recursos los coches (de los que sabe su velocidad), la carne (de lo que se sabe su origen) y el pan (se conoce sus calorías).

Cada grupo tiene un IVA diferente, siendo capaz de calcular su precio y obtener el precio final de una lista de productos.

Hay grupos que son comestibles (de los que se sabe cómo se comen y el sonido que hacen al masticarlos) y movibles (acelerando y frenando).

UNIVERSIDAD
DE MURCIA

Control de errores. Entrada y salida de datos

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
26 de octubre de 2024

Índice de Contenidos

Excepción = Error

- Lo que menos le gusta a un programador son los errores
- Considera los siguientes códigos y observa las excepciones que genera:

```
print(a)
print(55/0)
print('2' + 2)
lista = None
for i in range(2):
    print(lista[i])

lista = []
for i in range(2):
    print(lista[i])
```

- Las excepciones incorporadas en Python en
  https://docs.python.org/es/3/library/exceptions.html

Objetos de Excepciones

- Los errores **en tiempo de ejecución** se manejan usando excepciones.
- Las excepciones son objetos que se crean cuando el programa hace algo que se considera irregular.
- En los ejemplos anteriores cada xxxxError indica que se ha creado un objeto-error.
- Como objetos, siguen una jerarquía de clases.
- En Python, los usuales son las subclases de Exception: ¡¡Leelos!! `https://docs.python.org/es/3/library/exceptions.html#exception-hierarchy`
- Cuando se crea una excepción, se dice que se ha "lanzado" una excepción.
  - En Java lo llaman throw (lanzar, arrojar).
  - En Python lo llaman raise (elevar, levantar).
- Para tratar conveniente el error, la excepción debe ser "atrapada"
- **Estudiaremos** cómo
  - Capturar excepciones
  - Lanzar/Elevar excepciones
  - Crear nuestros propios objetos excepción

UNIVERSIDAD
DE MURCIA

Excepciones

Capturar Excepciones

Capturar excepciones

- Para manejar excepciones se usan la sentencias try/except/else/finally

```
try:

  // Código que se desea ejecutar

except [zzzError [as nombre]]:

  // Código a ejecutar si se produce un error en try
  // Código que controla la situación de error. Puede lanzar otro error.

else:

  // Bloque a ajecutar si try no tuvo errores

finally:

  // Código que se ejecutará siempre. Ocurra lo que ocurra en el programa.
  // P.e. cerrar los recursos
```

Ejemplo de Captura de Excepciones

try/except/finally

- Veamos antes el error sin capturas.

```
print(un_mensaje)
```

- Con las 3 componentes; pero a except genérico y le da el igual el **tipo de error**.

```
try:
    print(un_mensaje)  # Código que se desea ejecutar
except:                # Si hay error, se genera una excepción.
    print('Opppss ... hubo un error')   # Cualquier tipo de excepción
finally:               # Siempre se puede obligar a cierto código
    print('Siempre muestro este mensaje')
```

Ejemplo de Captura de Excepciones

try/except

Ejemplo.

- except solo se activa si el error es NameError.

```
try:
    print(un_mensaje)  # Código que se desea ejecutar
except NameError:                  # Salvo que se genere una excepción
    print('Opppss ... hubo un error')
```

- except asigna el error a una variable para poder ser manipulado.

```
try:
    print(un_mensaje)  # Código que se desea ejecutar
except NameError as error: # Salvo que se genere una excepción
    print(f'Opppss ... hubo un "{error}"')
```

Ejemplo de Captura de Excepciones con `else`

https://www.programiz.com/python-programming/exception-handling

```
def funcion(num):
  try:
    print(f"Número introducido {num}")
    assert num % 2 == 0
  except:  # Si try SÍ ha generado excepciones
    print("No es un número par")
  else:    # Si try NO ha generado excepciones
    reciproco = 1/num
    print(reciproco)
```

- Se activa except

```
funcion(1)
```

- Se activa else

```
funcion(4)
```

Varios `except`

- Si se conoce la jerarquía de clases de las excepciones se pueden usar varios "except".
- Las capturas "except" se ordenarán desde las subclases a las superclases
- Código correcto.

```
try:
    print(100/0)
except ZeroDivisionError:    #  ZeroDivisionError es SUBclase de ArithmeticE
    print("Error: divides por cero")
except ArithmeticError:
    print("Error aritmético")
```

- Código incorrecto

```
try:
    print(100/0)
except ArithmeticError:    #  ArithmeticError es una SUPERclase de ZeroDivis
    print("Error aritmético")
except ZeroDivisionError:
    print("Error: divides por cero")
```

Nunca se capturará un objeto de `ZeroDivisionError`.

UNIVERSIDAD
DE MURCIA

Excepciones

Lanzar Excepciones

Lanzar excepciones

- El programador también puede crear excepciones
- El programador puede lanzar una excepcion utilizando raise o assert.

Ejemplo. Para indicar que un método aún no ha sido implementado.

```
def metodo():
    raise NotImplementedError("Not supported yet.")

metodo()
```

Ejemplo. Para indicar que un parámetro no es el adecuado.

```
def metodo(param):
 assert param > 0, "Deber ser positivo"

metodo(-1)
```

```
def metodo(param):
 if param <= 0:
  raise ValueError("Deber ser positivo"

metodo(-1)
```

Excepciones en Setter

- La interface Setter permite asignar valores a los atributos de instancia de una clase.
- Los métodos Setter de una atributo deben lanzar errores/excepciones cuando los valores a asignar sean incorrectos.

  Ejemplo. Para indicar que un valor debe ser entero positivo.

```
class Dia:
    def __init__(self, d: int):
        self._dia = d

    def get_dia(self):
        return self._dia


    def set_dia(self, valor: int):
        try:
            self._dia = int(valor)
            assert self._dia > 0
        except Exception as e:
            print(f"#ERROR {e.__class__}")
```

UNIVERSIDAD
DE MURCIA

Excepciones

Definir Excepciones del Usuario

Creando Excepciones Nuevas

- Pueden crearse subclases a partir de la clase Exception
- Usa la notación usual de creación: `class` MiError(Excepcion)
- Otra opción es

```
class MiError(Exception):
  """Mi clase base para mis excepciones"""
  pass  # Lo correcto es definir __init__(self, *args)

raise MiError   # Alza un error sin texto
```

- Se pueden construir clases hijas: `class` ConcretoError(MiError)

```
class ConcretoError(MiError):
  """Una clase concreta para mis excepciones"""
  pass   # Lo correcto es definir __init__(self, *args)

raise ConcretoError("Esto es un error concreto")  # Con texto
```

  - Se llama al constructor de Exception, creándose una tupla como atributo de instancia con el listado de longitud indefinida de todo lo que se le pase. En este caso, sólo el mensaje.
  - Exception define el método *__str__()*, que imprime dicha tupla.
- Es una buena práctica colocar todas las excepciones en algún fichero separado como exceptions.py o errors.py. Pero esto no es obligatorio.

Personalizando las Excepciones Nuevas

- Las nuevas clases, como cualquier otra, pueden tener su propio constructor y métodos.
- Esto nos permite crear excepciones más personalizadas.

```
class NotInRangeError(Exception):
  def __init__(self, valor: int, mensaje="No está en el rango (10, 40)"):
    self._valor = valor
    self._mensaje = str(valor) + " -> " + mensaje
    super().__init__(self._mensaje)

valor = 50
if valor < 10 or 40 < valor:
  raise NotInRangeError(valor)
```

- Si no se crea un constructor, o si se invoca a super() con un listado indefinido de parámetros, se crea como una tupla en Exception.
- **RECUERDA**: Este tipo de errores, a su vez, pueden capturarse con las cláusulas *try-except*. En POO, un método o función encapsula en *try-except* las porciones de código o llamadas a funciones que pueden lanzar (*raise*) una excepción.

Ejercicio

Ejercicio.

- Crea la clase SintaxisError y sus subclases SinArrobaSintaxisError y CadenaVaciaSintaxisError
- Sobre la clase Persona se tienen los atributos eMail y nombre.
- Se quieren detectar los errores de que el correo electrónico no tiene arroba @ y que el nombre dado no es ni nulo ni tiene una longitud inferior a 3 caracteres.
- Escribe las clases indicadas con sus constructores, que deberán de lanzar las correspondientes excepciones si fuera necesario.
- Escribe el programa para una prueba

UNIVERSIDAD
DE MURCIA

Entrada y Salida de Datos

Entrada y Salida

- Un stream (secuencia o flujo) es una representación abstracta de un dispositivo físico de entrada o salida.
- Un stream **puede imaginarse** como un tubo por el que fluyen bytes de datos y dónde una vez transmitido el dato éste no se volverá a enviar.
- Hay dos tipos de flujos atendiendo a "su dirección":
  - Flujos de **entrada**: teclado, archivo de disco, ...
  - Flujos de **salida**: archivo de disco, una consola, impresora ...
- En Python se distinguen los siguientes tipos:
  - E/S de texto. En el flujo los datos serán objetos de la clase str. Representa los objetos como cadena de caracteres.
  - E/S binario (o *buffered IO*). En el flujo se esperan binarios como imágenes o archivos, y produce objetos de tipo bytes. Cada carácter corresponde a un valor entre 0 y 255.
  - E/S sin formato (o *unbuffered IO*). Trata los datos en crudo, a bajo nivel y sin buffer para aplicaciones específicas de baja latencia.

UNIVERSIDAD
DE MURCIA

Entrada y Salida de Datos

Entrada y Salida Estándar

Entrada y Salida Estándar

- Se trabaja con E/S de texto.
- **Mostrar objetos en la pantalla.**
  `print`(*objects, sep='', end='n', `file`=sys.stdout)
  - Todos los argumentos de *objects se convierten a cadenas de caracteres.
  - Los objetos se mandan al flujo separados por sep.
  - Se termina el envío con end.
  - file debe ser un objeto que implemente un método write(string). Si no se indica se usará sys.stdout. También puede usar sys.stderr (donde se envían las indicaciones del intérprete y mensajes de error).
- **Lee datos del teclado.** `input`([prompt])
  - Si se indica prompt se mostrará en la salida estándar sin nueva línea.
  - Lee una línea de la entrada sys.stdin, la convierte en una cadena (eliminando la nueva línea), y retorna eso.

```
>>> s=input("Dime tu edad: ")
Dime tu edad: 22
>>> print("Hola", "mundo", sep='X', end="!")
HolaXmundo!
```

UNIVERSIDAD
DE MURCIA

Entrada y Salida de Datos

Entrada y Salida a Ficheros

Esquema General

Pasos a seguir siempre

1. Abrir un fichero. f = `open`(`file`, mode=`'r'`)
   - file es un string indicando la ubicación del fichero.
   - mode es el modo de E/S. Por defecto es de lectura con flujo de texto.
   - Retorna un file object, que tiene una interface para tratar archivos (pasos siguientes) junto con algunos atributos:
     - f.closed: True si el fichero está cerrado. Falso en otro caso.
     - f.mode: Indica el modo de acceso de cómo se abrió el fichero.
     - f.name: Indica el nombre del fichero.
2. Leer/Escribir en el fichero.
   - f.write(string): Escribe un string en el fichero abierto.
   - f.read([n]): Vuelca en un string el contenido del fichero abierto. Opcionalmente se puede indicar el número de bytes que se quieran leer.
   - f. readline (): Lee una sola línea del archivo. Deja n al final de la cadena.
   - f. readlines (): Vuelca en una lista de strings el contenido del fichero
   - f. writelines (lista): Escribe una lista de líneas en el archivo.
   - f. tell (): Indica la posición actual del puntero.
   - f.seek(offset [, `from`]): Cambia la posición según offset desde una posición dada. Si `from` es 0 indica el inicio, 1 la posición actual y 2 la posición final.
3. Cerrar el fichero.
   - f. close (): Cierra el fichero

Modos de Apertura

- Al abrir un fichero con open() se debe indicar el **modo de apertura**.
- Un fichero se puede abrir para leer, sobreescibir sobre él o hacer las dos cosas simultáneamente. Se define el comportamiento inicial:
  - r: Abre el fichero para **lectura**. El fichero debe existir. El puntero del fichero se coloca al **principio**. Este modo es por defecto.
  - w: Abre el fichero para **escritura**. Sobreescribe el fichero si existe. El puntero del fichero se coloca al **principio**.
  - a: Abre el fichero para **añadir**. Si no existe el fichero lo crea. Si existe, coloca el puntero del fichero al **final**.
- A la vez, la E/S puede ser de texto o binaria. Y se puede especificar los usos del fichero una vez abierto.
  - b: Si se especifica se tratará el fichero como **binario**.
  - t: Será una E/S de texto. Valor por defecto.
  - +: Se puede usar el fichero para **leer y escribir**.
  - x: Abierto para **creación** en exclusiva, falla si el fichero ya existe.
- Por lo tanto, podríamos combinarlos, por ejemplo:
  - rb+: Para leer. Puntero al principio. Será una E/S binaria. También se podrá escribir.
  - a+: Para añadir. Puntero al final. Será una E/S texto. También se podrá leer.

Ejemplos Básicos

```
# Crear un fichero y escribe 3 líneas
f = open("test.txt", "w")
for num in range(3):
  f.write(str(num)+'\n') # \n para nueva línea
f.close()

# Lo abre de nuevo para añadir una cuarta línea
f = open("test.txt", "a")
f.write(str(4)+'\n')
f.close()

# Mostramos el contenido en pantalla
f = open("test.txt")
for linea in f:
  print(linea)
f.seek(0)  # Para empezar desde el principio
lista= f.readlines()
f.close()
```

Ficheros con `try/excepts`

- Si no existe el fichero para una lectura se generará un error.

```
f = open("testR.txt")
```

- En estos casos se debe usar try/excepts

```
try:
  f = open("testR.txt")
except:
  print("Opppps ! ... y no se cierra el fichero.")
else:
  f.read()
  f.close()
  print("Tarea realizada con éxito")
```

Ficheros con `with`

- Es importante que no se generen errores ANTES de cerrar el fichero

```
f = open("testW.txt", "w")
f.read()
print(f"Está cerrado? {f.closed}")
```

- Es fundamental cerrar el fichero f.close()
- Python garantiza el cierre usando with

```
try:
  with open("testW.txt", "w") as f:
    f.read()              # Genera error, pero se cerrará
except:
  print(f"Está cerrado? {f.closed}")
```

Notar que no se usa f.close() en el código anterior.

UNIVERSIDAD
DE MURCIA

Entrada y Salida de Datos

Ficheros con JSON

JSON

- JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos.

```
{
    "departamento": 8,
    "nombredepto": "Ventas",
    "director": "Juan Rodriguez",
    "empleados": [
        {
            "nombre": "Pedro",
            "apellido": "Fernández"
        },
        {
            "nombre": "Jacinto",
            "apellido": "Benavente"
        }
    ]
}
```

- En lo esencial es una diccionario cuyos valores pueden ser: (1) un valor, (2) una lista de valores, (3) una lista de diccionarios.

Ejemplo - I

Definimos en **una variable** toda la información:

```
empresa = {
  "departamento": 8,
  "nombredepto": "Ventas",
  "director": "Juan Rodríguez",
  "empleados": [
  {
    "nombre": "Pedro",
    "apellido": "Fernández"
  }, {
    "nombre": "Jacinto",
    "apellido": "Benavente"
  }
  ]
}
```

Ejemplo - II

```
import json    # Hay que importar esta librería

with open("json.txt", "w") as f:          # Escritura texto
  json.dump(empresa, f, indent=4)

with open("json.txt", "r") as f:          # Lectura texto. Retorna un Diccionario
  empresa = json.load(f)

print(json.dumps(empresa, indent=4)) # Lo convierte a str con formato JSON
```

UNIVERSIDAD
DE MURCIA

Entrada y Salida de Datos

Ficheros con pickle

Serialización de objetos Python

- Los pickles de Python representan un objeto Python como una cadena de bytes.
- Se pueden transmitir por red o guardar en un archivo.

```
>>> import pickle    # Hay que importar esta librería

>>> datos = [list().append(2), set().add(100)]

>>> with open("datos.pickle", "wb") as f:          # Escritura *binario*
...  pickle.dump(datos, f)
...
>>> with open("datos.pickle", "rb") as f:          # Lectura *binario*
...  datos_p = pickle.load(f)
...
>>> print(pickle.dumps(datos_p)) # Pickle como cadena binaria (bytes)
b'\x80\x04\x95\x15\x00\x00\x00\x00\x00\x00\x00]\x94(]\x94(K\x01K\x02e\x8f\x94(K

>>> print(datos_r) # Lo convierte a str
[[1, 2], {3, 4}]
```

No reinventes la rueda

Algunas librerías para manipular ficheros.

- **csv**: Leer y escribir a ficheros .csv (Excel)
- **xlwings**: Leer y escribir a ficheros .xls (Excel)
- **wave**: Leer y escribir a ficheros WAV (audio Microsoft)
- **aifc**: Leer y escribir a ficheros AIFF y AIFC (audio)
- **tarfile**: Leer y escribir a ficheros tar (compresión)
- **zipfile**: Leer y escribir a ficheros ZIP (compresión)
- **xml.etree.ElementTree**:] Leer y escribir a ficheros XML (marcas)
- **msilib**: Leer y escribir a ficheros de instalación de Microsoft.
- **PyPDF2**: Manipulación de ficheros .pdf (Adobe)
- **Pillow**: Manipulación de ficheros de imágenes.

¡**IMPORTANTE** leer la documentación oficial para saber como usar sus funcionalidades!

UNIVERSIDAD
DE MURCIA

Entrada y Salida de Datos

Más sobre Ficheros Locales. El módulo os

No reinventes la rueda

El módulo os proporciona métodos que ayudan a la gestión de ficheros.

- Relacionados con ficheros:
  - os.rename('current_file_name', 'new_file_name') Renombra un fichero.
  - os.remove('file_name') Borra un fichero.

- Relacionados con directorios:
  - os.mkdir("newdir") Construye un nuevo directorio.
  - os.chdir("newdir") Cambia el directorio actual.
  - os.getcwd() Muestra el directorio de trabajo actual.
  - os.rmdir("dirname") Borra el directorio actual

UNIVERSIDAD
DE MURCIA

Entrada y Salida de Datos

Ficheros Lejanos. El módulo urllib

Puedes acceder a ficheros externos

Proporciona métodos que ayudan a la gestión de ficheros externos vía URL.
Es muy útil para análisis de datos (estadística, machine learning, ...)

```
def words_file(url):
    from urllib import request
    from urllib.error import URLError
    try:
        file = request.urlopen(url)
    except URLError:
        # Observa que retorna, NO imprime
        return('La url ' + url + ' no existe')
    else:
        content = file.read()
        # split() divide un string por espacios.
        # Construye una lista con cada una de l
        return len(content.split())

print(words_file('https://www.gutenberg.org/files/2000/2000-0.txt'))
print(words_file('https://no-existe.txt'))
```

Fuente: `https://tinyurl.com/27wjb5td` (visitar)

Ejercicio

Ejercicio.

A veces querrás leer un fichero y escribir en el otro a la vez.

- Crea la función `def guarda(fichero: str, datos: list)` para guardar una lista de strings en un fichero.
- Crea la función `def backup(fichero_in: str, fichero_out: str)` para hacer un backup de un fichero en otro. Haz dos versiones de la función:
  1. En cada linea de respaldo añade el número de línea.
  2. En el respaldo, las líneas se colocan en el orden inverso al del fichero original.
     - Usa la función reversed() que retorna un iterador en el orden inverso del iterable dado.

```
for num in reversed([1, 2]):
  print(num)
```

     - .readlines() usa un iterador para guardar las líneas.

Ejercicio

Ejercicio.

Se tiene un fichero .csv con las calificaciones de los alumnos. Es un fichero de texto donde cada fila constará de 3 datos separados por comas. Puedes generar el fichero con una hoja de cálculo o directamente con un editor que guarde el contenido sin formato.

```
,Teoría,Prácticas
Alumno A,6,4
Alumno B,3,5
....
```

Haz un programa que lea el fichero y muestre en pantalla (1) el nombre del alumno junto con el promedio de las dos notas, (2) el número de alumnos aprobados.

UNIVERSIDAD
DE MURCIA

Tema 7. Abstracción. Tipos de Datos Abstractos

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
3 de noviembre de 2024

Índice de Contenidos

Abstracción

- La **abstracción** es el proceso de identificar las **características principales** de un concepto, descartando detalles no esenciales para centrarse en lo relevante.
- Este proceso categoriza elementos en grupos, donde cada grupo es una abstracción que destaca aspectos importantes e ignora características innecesarias.
- La abstracción permite estudiar sistemas complejos a diferentes niveles de detalle, formando un **modelo jerárquico**.
- *Ejemplo: Una aplicación de gestión de tickets se puede dividir por capas de abstracción en el sistema, eventos, usuarios, reservas y tickets. A su vez, cada modelo se construye de variables que abstraen de valores concretos y funciones que generalizan un comportamiento particular.*
- En programación, la abstracción permite:
  - Simplificar sistemas complejos
  - Diseñar de manera estructurada
  - Separar responsabilidades

UNIVERSIDAD
DE MURCIA

Tipos de Abstracción

Tipos de Abstracción

- **Abstracción procedimental.** Oculta un conjunto de operaciones en un **procedimiento**, de modo que el programador lo usa sin preocuparse de cómo funciona.
  - **Funciones** y **procedimientos** responden a esta abstracción.
  - *Una función calcularPromedio() recibe una lista de números y devuelve su promedio. El usuario sólo necesita saber qué hacer, sin conocer la lógica exacta.*
- **Abstracción de iteración.** Simplifica el proceso de iterar sobre **colecciones**, **pasando de un elemento al siguiente** sin saber cómo se organizan de forma interna.
  - Dos elementos: **iterable** (colección) e **iterador** (variable).
  - *En Python, con for item in lista el programador no se preocupa por gestionar el índice o fin de recorrido de la lista, sino de tratar cada elemento.*
- **Abstracción de datos.** Oculta la **estructura interna de un tipo de dato y sólo expone las operaciones** necesarias para que el programador interactúe (sin tener que conocer la estructura interna).
  - Modelo de datos + operaciones (clase)
  - *Una clase Pila que expone solo métodos como apilar() o desapilar() sin que el usuario sepa si los datos se almacenan en un array, una lista enlazada o alguna otra estructura.*

Abstracción Procedimental

- Consiste en crear procedimientos y funciones como abstracción de operaciones.
- Un procedimiento/función **debe responder sin ambigüedad a qué hace obviando el cómo lo implementa.**
- Abstraemos un conjunto de operaciones (cómo se realiza) como una única operación (qué hace), expresada como función o procedimiento.
- Las técnicas de programación procedimental y modular usan este tipo de abstracción para romper el problema principal en problemas más pequeños.
- Toda operación se debe especificar de la siguiente forma:

```
operacion nombre (id1:tipo1, id2: tipo2, ....) return tipo
    Descripción: Descripción textual del comportamiento
    Precondiciones: Condiciones que se deben cumplir
    Retorna: Indica el tipo de dato que retorna
    Excepciones: Indica las excepciones (opcional)
```

- Este tipo de especificaciones los incorporan los editores de programación.
- Permiten generar la documentación (API) para los programadores. Un buen ejemplo es `https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#substring(int)`

Abstracción de Iteración

- Consiste en tener algún mecanismo que permita acceder a los elementos de un contenedor sin tener en cuenta su representación interna.
- **Representación interna**: Estructura de datos que se utiliza para representar al contenedor.
- Los bucles con contadores tienen en cuenta la representación de los datos.

```
for i: int = 0...top:
    acción sobre elemento[i] # Considera estructura indexada
```

- Es más adecuado tener una abstracción de la forma:

```
for cada elemento P de Contenedor
    acción sobre P
```

- La responsabilidad del recorrido se traslada a un objeto que se llamará iterador.

Abstracción de Datos

- Existen 3 tipos o **niveles de abstracción**.
- Tipos de datos integrados. Simples (enteros, reales y booleanos) o Compuestos (string, rangos, listas, tuplas, diccionarios y conjuntos)

```
vector: tuple = (a,b)   # Vector 2D (tupla)
print(vector[0])        # Coord X (índice 0, representación interna)
```

- Tipos de datos definidos por el usuario. Conceptos más abstractos agrupando datos integrados. Estructuras (en Python, clase sin operaciones)

```
vector: Vector2D = Vector2D(x=a,y=b)   # Vector 2D (estructura)
print(vector.x)       # Coord X (atributo x, representación interna)
```

- Tipos de datos abstractos (TDA). Construye modelos usando agrupación de datos y operaciones asociadas. Posible implementación con Clases.
  - Se trabaja con las operaciones del TDA sin importar la estructura interna de los datos.

```
vector: Vector2D = Vector2D(x=a,y=b)   # Vector 2D (TDA)
vector.imprimir_x()   # Coord X (método sin saber estructura interna)
```

UNIVERSIDAD
DE MURCIA

Tipos de Datos Abstractos

Tipos de Datos Abstractos

- Los Tipos de Datos Abstractos (TDA) son **modelos** que constan de
  - un nombre público para
  - identificar a un conjunto de datos (valores), junto con
  - un conjunto de operaciones bien definidas sobre los datos
- Ejemplos de TDAs:
  - *TDA Usuario* donde se definen *nombre*, *amigos*, *publicaciones* y operaciones como *agregarAmigo()* y *hacerPublicacion()*.
  - *TDA Transacción* con *monto*, *fecha*, *cuentaOrigen*, *cuentaDestino* y operaciones como *procesarTransaccion()* y *verificarFondos()*.
- Todo TDA abarca **diseño** e **implementación** en tres tareas:
  - 1. **Especificación:** Describe el TDA y define sus operaciones (qué hace) para que los usuarios entiendan el modelo.
  - 2. **Representación:** Estructura interna del TDA para caracterizar el modelo.
  - 3. **Implementación:** Cómo implementar la estructura y operaciones en un lenguaje de programación. Un TDA se puede implementar de muchas maneras y en muchos lenguajes de programación.
    Por ejemplo, un TDA puede implementarse en una clase en Python.

UNIVERSIDAD
DE MURCIA

Tipos de Datos Abstractos

Especificación de TDAs

Ejemplo de especificación de TDA

Diseño de tipo de dato independiente de representación o lenguaje de programación

TDA Polinomio

- Es una expresión algebraica $p(x) = c_0 + c_1x + c_2x^2 + \ldots + c_nx^n$ compuesta por la **suma** de dos o más monomios (es un **coeficiente** y una **variable** con **exponente**). No existen dos monomios con el mismo exponente. Todos los monomios usan la misma variable. El k-ésimo monomio es el que tiene una variable con coeficiente $\neq 0$.
- Usa: naturales (exponentes), reales (coeficientes)
- Operaciones:
  - `Crear (real[] coef) : Polinomio`
    El k-ésimo coeficiente representa al k-ésimo monomio.
  - `suma (Polinomio f, Polinomio g) : Polinomio`
    Retorna la suma de polinomios
  - `termino (Polinomio f, int k) : real`
    Retorna el coeficiente del k-ésimo monomio
  - *etc* $\cdots$
- **NOTA.** Aquí se muestra una versión simplificada. Hay que seguir la especificación procedimental (descripción, precondiciones, valor de retorno y excepciones)

Especificación de TDAs

- La especificación de un TDA (Datos+Operaciones) consta de 3 partes: nombre y descripción; definición de datos; y definición de operaciones.
- Nombre y descripción.
  - Se le dará un nombre que identifica al modelo y las operaciones.
  - Se indicará qué representa.
- Especificación de los datos.
  - Puede usar notaciones matemáticas conocidas.
    Ejemplo. Conjuntos conocidos ($\mathbb{N}$, $\mathbb{R}$, ...), conjuntos $\{s_1, s_2, \ldots\}$, intervalos $[a, b]$, ...
  - Nunca se usan tipos o estructuras concretas de un lenguaje de programación.
- Especificación de las operaciones (abstractas).
  - Se indicará tanto la sintaxis (cabecera) como la semántica (especificación) de cada una (abstracción procedimental).
- **IMPORTANTE.** Un TDA define un valor de tipo T:
  Todos los entes/valores que respondan al TDA T son de tipo T.

Tipos de Operaciones de los TDAs

- Las operaciones de un TDA se pueden dividir por su objetivo:
  - Constructores. Los que indican cuáles son los datos necesarios para construir un valor de tipo T.
  - Modificadores. Los que construyen un nuevo valor de tipo T a partir de un valor de tipo T dado.
  - Consulta. El conjunto de operaciones que a partir de un valor de tipo T retornan un valor, que no es de tipo T.
- Las operaciones de un TDA se pueden dividir por su importancia:
  - Fundamentales, también llamadas primitivas:
    - **No se pueden quitar:** Son funciones atómicas necesarias para resolver un problema. *Por ejemplo, sumar dos polinomios.*
    - **Todas deben poder usarse:** todas deben estar visibles.
  - No fundamentales.
    - **Apoyan la definición de una operación fundamental,** pero no se puede considerar como tal. Deben estar ocultas. *Por ejemplo, comprobar si existe o no un término particular.*
    - Las que **aumentan el conjunto de operaciones** y se construyen a partir de fundamentales. *Por ejemplo, método para sumar tres polinomios*.

UNIVERSIDAD
DE MURCIA

Tipos de Datos Abstractos

Representación de TDAs

Representación de TDAs

- Se debe de elegir una estructura **rep** indicando qué datos de un valor de tipo T se almacenan en la estructura **rep**.
- Función de abstracción. Una función sobreyectiva $Abst$: **rep** $\longrightarrow \mathcal{A}$ (conjunto de objetos que pueden representarse con el TDA)
- Invariante de la representación. Es un predicado I : **rep** $\longrightarrow \mathbb{B}$ que es cierto para los objetos de **rep** que sean legítimos.

- Ejemplo. Representación de Polinomios
  - Para los polinomios $p(x) = c_0 + c_1x + c_2x^2 + \ldots + c_nx^n$ se opta por:

```
struct rep {
    entero grado
    real[] coef
}
```

  - `grado` es un entero para representar el grado del polinomio, y
  - `coef` es una lista indexada de reales (no ARRAY)
  - $Abst(r) =$ `r.coef[0]+r.coef[1]`x`+r.coef[2]`x^2 `+ ...`
    `... + r.coef[r.grado]`$x^{\texttt{r.grado}}$
  - $I(r)=$ `(r.grado`$\not<$`0) AND r.grado = len(r.coef)`

UNIVERSIDAD
DE MURCIA

Tipos de Datos Abstractos

Implementación de TDAs

Implementación de TDAs

- lenguaje: lenguaje de programación en el que se implementa el TDA.
- Se deberá de implementar considerando:
  - Encapsulamiento, agrupando en un objeto atributos (variables) y métodos (operaciones).
  - Ocultación de información, pues establecer qué atributos y métodos pueden permanecer visibles u ocultas.
- La POO aparece de forma natural para resolver esta situación.
- Nuestro Objetivo:
  Usar la POO para implementar TDAs (que son modelos) en Python.
- Lo que **se necesita para usar** un TDA en ese lenguaje es:
  - su nombre,
  - su dominio (conjuntos de datos con los que trabaja y su tipo),
  - su interface (los nombres de las operaciones asociadas).

  Estos 3 elementos caracterizan a un TDA: parte pública usable por cualquiera.
- Lo que **no necesita conocer** el usuario del TDA en ese lenguaje es:
  - la **estructura de datos** usada (cómo está codificado la representación)
  - cómo se han **implementado** los algoritmos (operaciones)

  Estos elementos reciben el nombre de parte privada, responsabilidad del autor del TDA.

Implementación de TDA Polinomio en una clase en Python

- **Parte privada** (clase desarrollada por el autor del TDA):

```
class Polinomio:

    def __init__(self, coeficientes: list[int]):
        """El k-esimo coeficiente representa al k-esimo monomio."""

        # REPRESENTACIÓN PRIVADA DE TDA POLINOMIO
        self._coeficientes: list[int] = coeficientes
        self._grado: int = len(coeficientes)

    def suma(self, otro: 'Polinomio') -> 'Polinomio':
        """Retorna la suma de polinomios"""
        {implementación de algoritmo en Python, omitido por longitud}

    def termino(self, grado: int) -> int:
        """Retorna el coeficiente del k-esimo monomio"""
        {implementación de algoritmo en Python, omitido por longitud}
```

- **Parte pública** (información para que un tercero pueda usar dicho TDA):
  - Nombre: `Polinomio`
  - Dominio: enteros (`int`), reales (`float`)
  - Interface documentada/especificada:
    - `crear(coeficientes: list[float]) -> Polinomio`
    - `suma(Polinomio a, Polinomio b) -> Polinomio`
    - `termino(Polinomio a, int indice) -> float`

UNIVERSIDAD
DE MURCIA

Tema 8. Tipos de Datos Abstractos Sin Orden

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
15 de noviembre de 2024

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

Colección de datos

Introducción

Colección de datos

- Una colección **agrupa datos** de una forma que permite el acceso y manipulación básica, como **agregar**, **eliminar** o **buscar elementos**.
- **Ejemplos:** Conjuntos, diccionarios, listas, colas, pilas, grafos, árboles, etc... Cada uno de ellos con operaciones y comportamientos diferentes.
- Por ejemplo, **Python** ya implementa sus propias colecciones que están optimizadas para ciertas operaciones:
  - *Range*: secuencia de números enteros (útil en bucles for)
  - *Sets*: colección de elementos únicos y desordenados
  - *Listas*: colección ordenada de elementos que permite duplicados
  - *Tuplas*: similar a una lista, pero inmutable.
  - *Diccionarios*: colección de pares clave-valor, con claves únicas.
- Pero, *¿cómo se especifican, representan e implementan internamente estas colecciones?*
- En los siguientes temas, aprenderemos a diseñar los TDAs básicos para trabajar con colecciones **sin usar** estos tipos de datos integrados en Python.

UNIVERSIDAD
DE MURCIA

Colección de datos

Estructuras de datos

Tipos de estructura de datos

- Una estructura de datos define **cómo los datos están organizados y almacenados en memoria** para que puedan ser accedidos/manipulados.
- Una colección se puede **representar** principalmente dos formas:
  - **Estructura contigua**
    - Los elementos se almacenan en **posiciones de memoria consecutivas**, lo que permite un acceso rápido por índice.
    - Generalmente permite una mayor eficiencia en el acceso directo a elementos, frente a eliminaciones menos eficientes.
    - *Ejemplo genérico*: Arrays (Python no tiene...)
  - **Estructura enlazada**
    - Los elementos no se almacenan en posiciones consecutivas, sino que cada elemento contiene una **referencia al siguiente**.
    - Facilita la inserción y eliminación de elementos en cualquier posición sin necesidad de reorganizar toda la estructura.
    - *Ejemplo genérico*: Lista enlazada.
- Sobre estos dos tipos de estructura se podría conocer la longitud, buscar un elemento, añadirlo o eliminarlo, entre otros.
- El **tipo de colección** viene determinado por la **especificación** de sus operaciones. *Por ejemplo, en una Pila sólo se puede insertar y extraer elementos de la estructura por el principio. En un Conjunto, no se pueden repetir elementos...*

Estructura contigua vs enlazada

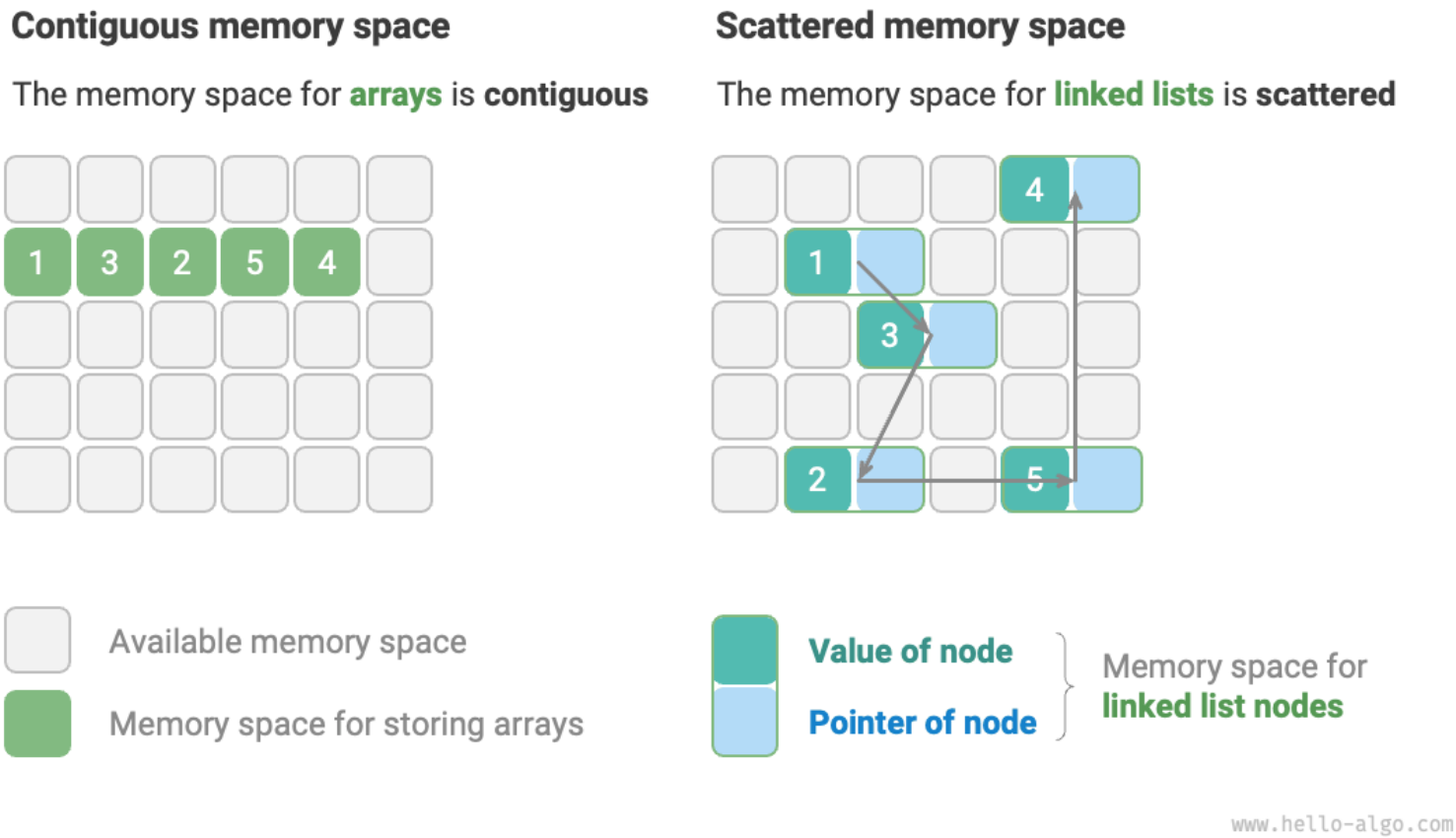


Figura: Almacenamiento en memoria de ambos tipos de estructuras

UNIVERSIDAD
DE MURCIA

Colección de datos

Estructura contigua

Estructura contigua: array

- Se guardan los elementos secuencialmente en memoria, en un espacio fijo reservado de antemano. Por tanto, hay acceso directo a las posiciones.
- En esta estructura, los elementos se guardan en un array.

```
struct EstructuraContigua {
    capacidad: int                          # capacidad
    elementos: array[CAPACIDAD_MINIMA]      # array
    longitud: int                           # longitud actual
}
```

- Se inicializa con un tamaño con las posiciones disponibles.
  - Inicialmente son posiciones vacías/inválidas.
  - Los elementos se guardan o mueven a lo largo de la estructura.
  - En el caso de llenarse, se debe redimensionar la estructura.
    - Esto conlleva crear un array más grande y copiar los elementos.
- Python no tiene arrays, pero podemos simularlo con el tipo list.
- Hay acceso directo a los elementos. No usar funciones ni métodos de list.
- La capacidad actual, las posiciones disponibles o la manipulación de la estructura se realiza internamente, de manera **transparente** al usuario.

Estructura contigua: operaciones genéricas

- *Crear*. Inicializa la estructura vacía con una longitud por defecto.
- *Consultar longitud*. Devuelve el número de elementos guardados.
- *Verificar si existe un elemento dado*. Recorre la estructura, devolviendo True si se encuentra, False en caso contrario.
- *Devolver el elemento de una posición dada*. Devuelve el elemento de una posición dada. Hay acceso directo.
- *Modificar el elemento de una posición dada*. Sobreescribe el elemento que haya en una posición dada. Hay acceso directo.
- *Insertar un nuevo elemento en la posición dada*. Añade un nuevo elemento en una posición, desplazando antes el resto de elementos a la derecha. Si se ha llegado a la capacidad máxima, el array debe redimensionarse (por ejemplo, al doble de tamaño).
- *Eliminar el elemento de una posición dada*. Elimina y libera una posición dada, desplazando el resto de elementos a la izquierda. El array podría reducir el tamaño si se vacía mucho.
- *Devolver iterador de la estructura*. Devuelve una referencia para que de manera externa se pueda recorrer la estructura.

Estructura contigua: algunas indicaciones

- *Representación*. Importante la simulación del array.

```
clase EstructuraContigua:

  CAPACIDAD_MINIMA = 2                        # capacidad minima

  EstructuraContigua():
    capacidad = CAPACIDAD_MINIMA              # capacidad actual
    elementos = array[CAPACIDAD_MINIMA]       # array de elementos
    longitud = 0                              # longitud actual
```

- *Consultar una posición (posicion_target)*

```
elementos[posicion_target]  # Acceso directo. Faltan comprobaciones.
```

- *Establecer un elemento (nuevo) en una posición (posicion_target)*.

```
elementos[posicion_nuevo] = nuevo  # Faltan comprobaciones
```

- *Buscar un elemento* (target). Uso del while para la búsqueda.

```
i = 0
mientras i < longitud y elementos[i] != target:
  i = i + 1
si i < longitud:
  trabajar con elementos[i] encontrado
```

Estructura contigua: algunas indicaciones

- *Insertar un elemento (nuevo) en una posición dada (posicion_nuevo)*. No sobreescribe. Mover elementos a derecha y sobredimensionar si se llena.

```
si longitud == capacidad:
    crear elementos_nuevo[capacidad*2]
    copiar elementos[0] hasta elementos[posicion_nuevo-1] en elementos_nuevo[]
    insertar nuevo en elementos_nuevo[posicion_nuevo]
    copiar desde elementos[posicion_nuevo] hasta elementos[longitud]
                                                    en elementos_nuevo[]
    elementos = elementos_nuevo
sino:  # no se llena
  mover a derecha elementos[longitud] hasta elementos[posicion_nuevo]
  insertar nuevo en elementos[posicion_nuevo]

longitud=longitud+1
```

- *Eliminar el elemento de una posición dada (posicion_target)*. Elimina y libera una posición dada, desplazando elementos a izquierda. Análogo al isertar, se podría reducir la capacidad si se detecta el vaciado.

```
for i desde posicion_target hasta longitud-1:
    elementos[i] = elementos[i+1]
    longitud = longitud-1
```

Estructura contigua: implementación

¡IMPLEMENTA UNA ESTRUCTURA CONTIGUA EN PYTHON!

```
class EstructuraContigua:

    # creación
    def __init__(self):

    # operaciones
    def len(self):

    def contains(self, elemento: object) -> bool:

    def get_item(self, posicion: int) -> object:

    def set_item(self, elemento: object, posicion: int):

    def insert_item(self, elemento: object, posicion: int):

    def remove_item(self, posicion: int):

    def iterator(self) -> iter:
```

UNIVERSIDAD
DE MURCIA

Colección de datos

Estructura enlazada

Estructura enlazada: nodos con referencias

- Se guardan los elementos dispersos en memoria, construidos de manera dinámica bajo demanda.
- Una estructura enlazada es aquella que se construye utilizando como estructura básica un nodo:

```
struct Nodo {
    elemento: object
    siguiente: Nodo
}
```

- En una colección de nodos, cada uno tiene una referencia a otro nodo (cuyo campo llamaremos siguiente o next). El último no apunta a nada.
- Una estructura enlazada referencia al primer nodo, a partir de él se puede recorrer todos los elementos por las referencias de siguiente.

```
struct EstructuraEnlazada {
    longitud: int
    primer_nodo: Nodo
}
```

  - Se inicializa vacío y con primer nodo vacío/inválido.
  - No hay acceso directo a las posiciones, pero las inserciones o eliminaciones son eficientes pues no afectan a toda la estructura.

Estructura enlazada: operaciones genéricas

- *Crear*. Inicializa la estructura enlazada vacía con una longitud de cero y primer nodo nulo.
- *Consultar longitud*. Devuelve el número de elementos.
- *Verificar si existe un elemento dado*. Recorre la estructura desde el primer nodo hasta el final, devolviendo True si encuentra el elemento, False en caso contrario.
- *Devolver el elemento de una posición dada*. Recorre la estructura hasta llegar a la posición, devolviendo el elemento. No hay acceso directo.
- *Modificar el elemento de una posición dada*. Recorre la estructura hasta llegar a la posición, sobreescribiendo el elemento. No hay acceso directo.
- *Insertar un nuevo elemento en una posición*. Se recorre la estructura hasta llegar a la posición. La inserción implica ajustar las referencias del nodo anterior y posterior, sin desplazar elementos en memoria.
- *Eliminar un elemento específico*. Busca y elimina el elemento, actualizando las referencias en los nodos adyacentes para mantener la conexión sin mover elementos en memoria.
- *Devolver iterador de la estructura*. Devuelve una referencia para que de manera externa se pueda recorrer la estructura.

Estructura enlazada: algunas indicaciones

- *Representación*. La estructura inicialmente está vacía y sin nodos. Cada elemento que se inserte estará en un nuevo nodo. El último nodo no apuntará a otro nodo, es decir, su campo *siguiente* es None.

```
clase Nodo:
  elemento: object
  siguiente: Nodo

clase EstructuraEnlazada:
  longitud: int = 0                    # longitud actual
  primer_nodo: Nodo = None             # referencia al primer nodo
```

- *Consultar un elemento en una posición (posicion_target)*. Se recorre la estructura hasta encontrar el nodo en la posición deseada. No hay acceso directo. **IMPORTANTE: Usamos una variable de tipo Nodo "actual", que va iterando a través de "siguiente".**

```
# antes comprobar que la posicion es válida
actual = primer_nodo
i = 0
while i < posicion_target:
  actual = actual.siguiente
  i = i + 1
trabajar con actual.elemento
```

Estructura enlazada: algunas indicaciones

- *Establecer un elemento (nuevo) en una posición (posicion_target)*. Se recorre hasta la posición y luego se reemplaza el elemento. No hay acceso directo.

```
# antes comprobar que la posicion es válida
actual = primer_nodo
i = 0
while i < posicion_target:
  actual = actual.siguiente
  i = i + 1
actual.elemento = nuevo
```

- *Buscar un elemento específico (target)*. Uso de un bucle 'while' para recorrer y buscar el elemento. Importante comprobar si hemos terminado con el último nodo (actual==None).

```
actual = primer_nodo
while actual != None and actual.elemento != target:
  actual = actual.siguiente
si actual != None:
  trabajar con actual.elemento encontrado
```

Estructura enlazada: algunas indicaciones

- *Insertar un elemento (nuevo) en una posición dada (posicion_nuevo)*. Se recorre hasta la posición anterior, luego se ajustan las referencias para insertar el nuevo nodo. No hay desplazamientos en memoria ni sobredimensionado.

```
nuevo_nodo = Nodo(elemento=nuevo)
si posicion_nuevo == 0:
    nuevo_nodo.siguiente = primer_nodo
    primer_nodo = nuevo_nodo
sino: # comprobar que es una posición válida...
    actual = primer_nodo
    i = 0
    while i < posicion_nuevo-1:
        actual = actual.siguiente
        i = i + 1
    nuevo_nodo.siguiente = actual.siguiente
    actual.siguiente = nuevo_nodo

longitud = longitud + 1
```

IMPORTANTE: Añadir un nuevo nodo (nuevo_nodo) detrás de otro (actual):

1. Crear nodo con el elemento: *nuevo_nodo = Nodo(nuevo)*
2. Conectarlo con su siguiente: *nuevo_nodo.siguiente = actual.siguiente*
3. Ser conectado con el anterior: *actual.siguiente = nuevo_nodo*

Estructura enlazada: algunas indicaciones

- *Eliminar un elemento en una posición dada (posicion_target)*. Se recorre hasta el nodo anterior, ajustando referencias para eliminar el nodo correspondiente. No hay desplazamientos en memoria.

```
si posicion_target == 0:
    primer_nodo = primer_nodo.siguiente
sino:  # comprobar que es una posición valida...
    actual = primer_nodo
    i = 0
    while i < posicion_target - 1:
        actual = actual.siguiente
        i = i + 1
    borrar = actual.siguiente
    actual.siguiente = borrar.siguiente

longitud = longitud - 1
```

IMPORTANTE: Eliminar un nodo (borrar) que está detrás de otro (actual):

1. Obtener el nodo a borrar: *borrar = actual.siguiente*
2. Desconectarlo del actual: *actual.siguiente = borrar.siguiente*
3. Liberar el nodo: *borrar se elimina pues no es apuntado por nadie*
   Python lo hace automáticamente, en otros lenguajes se debe realizar explícitamente.

Estructura enlazada: implementación

Cabeceras de los métodos (parte pública) son iguales a la Estructura contigua

```
class EstructuraEnlazada:

    class Nodo:
        # creación
        def __init__(self, elemento: object):

        def siguiente(self) -> Nodo:

        def elemento(self) -> object:

    # creación
    def __init__(self):

    # operaciones
    def len(self):

    def contains(self, elemento: object) -> bool:

    def get_item(self, posicion: int) -> object:

    def set_item(self, elemento: object, posicion: int):

    def insert_item(self, elemento: object, posicion: int):

    def remove_item(self, posicion: int):

    def iterator(self) -> iter:
```

Estructura contigua vs enlazada

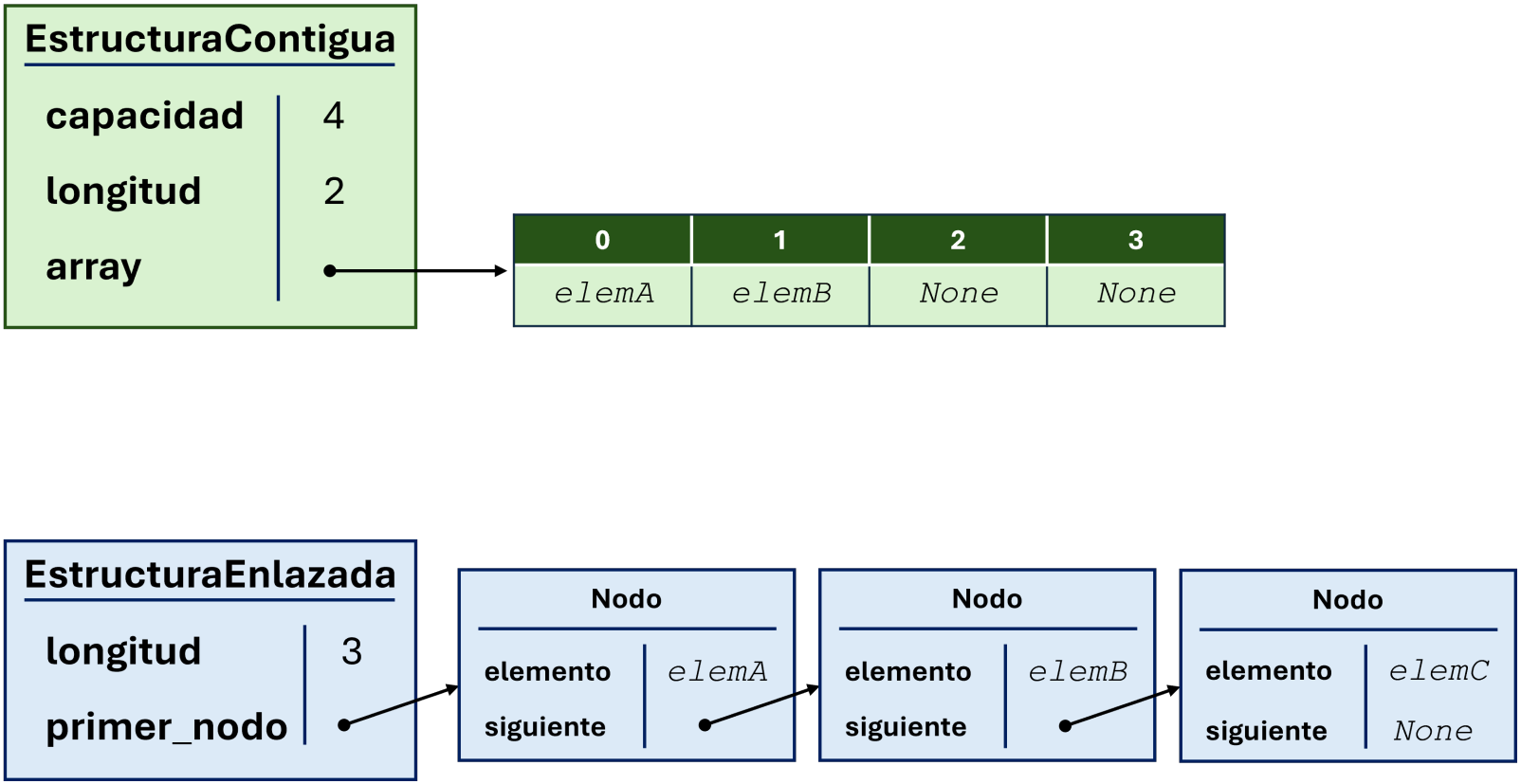


Figura: Representación de estructuras de datos genéricas

UNIVERSIDAD
DE MURCIA

Colección de datos

TDAs para colección de datos

Colecciones de datos - TDAs

- Una colección de datos puede abstraerse, diseñarse e implementarse mediante un Tipo de Dato Abstracto (TDA):
- **Especificación**: Nombre, dominio y operaciones posibles.
- **Representación**: Principalmente mediante una estructura contigua o estructura enlazada.
- **Implementación**:
  - Crear una clase en Python con la representación (constructor) y operaciones (métodos).
  - *RECUERDA*: Python ya incorpora sus propios TDAs (rangos, sets, listas, tuplas y diccionarios). **NO se pueden usar**, hay que:
    a. Programar operaciones usando una estructura contigua con apoyo de una lista (a modo de **array**). Por lo tanto, no se pueden usar los métodos o funciones de una lista en Python.
    b. Programar operaciones en una estructura enlazada con nodos.
- Por lo tanto:
  - Un TDA no es una estructura de datos. La estructura de datos es una posible forma de **representación**.
  - Un TDA no es una clase. Una clase es una posible forma de **implementación**.

Colecciones de datos - TDAs

- **TDAs sin orden** (*Tema 8*): No imponen un orden en los elementos.
  - TDA Bag: Colección de elementos sin orden ni unicidad; permite duplicados.
  - TDA Set: Colección de elementos únicos, sin orden.
  - TDA Map: Colección de pares clave-valor, donde las claves son únicas.
- **TDAs con orden lineal** (*Tema 9*): Los elementos están en una secuencia lineal.
  - TDA Lista: Secuencia de elementos accesibles por posición, permite duplicados.
  - TDA Pila: Colección LIFO (último en entrar, primero en salir).
  - TDA Cola: Colección FIFO (primero en entrar, primero en salir).
- **TDAs con orden no lineal** (*Tema 10*): Organización jerárquica o en red.
  - TDA Árbol: Estructura jerárquica de nodos con un nodo raíz y múltiples niveles de subnodos.
  - TDA Árbol Binario: Árbol donde cada nodo tiene a lo sumo dos hijos (izquierdo y derecho).

UNIVERSIDAD
DE MURCIA

TDAs sin orden

Qué son los TDAs sin relación de orden

- Los TDAs sin relación de orden hacen referencia a contenedores donde no establecemos ningún criterio de ordenación entre los elementos que almacena.
- Ejemplos:
  - La cesta de la compra en un supermercado.
  - Las personas que están en un sala de espera, identificadas con un boleto.
- Podemos distinguir las bolsas (bag), conjuntos (set) y mapas (map).
- Las operaciones generales que podemos definir en este tipo de contenedor, donde los objetos simplemente se almacenan, son:
  - Consultar el número de objetos que tiene el contenedor,
  - Determinar si el contenedor está vacío,
  - Añadir un nuevo objeto en el contenedor,
  - Informar si un objeto está en el contenedor (pertenencia),
  - Retirar un objeto del contenedor, y
  - Quitar todos los objetos (limpiar) del contenedor.

UNIVERSIDAD
DE MURCIA

TDAs sin orden

TDA Bag

TDA Bag: Especificación

- Contenedor que almacena una colección de elementos (objetos o items) donde todos son del mismo tipo y no importa la posición donde se almacenó cada uno ni el número de veces que aparecen.
- Operaciones particulares
  - Bag(): Bag. Crea un nuevo *bag*, inicialmente vacío.
  - len(): int. Retorna el número de elementos en el *bag*.
    Para un bag cualquiera $< a_0, a_1, \ldots, a_{n-1} >$ retornará el valor n.
  - contains(element): bool. Indica si el elemento element se encuentra en el *bag*. Retorna True si está contenido y False si no está contenido.
  - add(element): None. Modifica el *bag* añadiendo el elemento element al *bag*.
  - remove(element): None. Elimina y retorna la ocurrencia element del *bag*. Lanza un error si el elemento no existe.
  - iterator(): IteratorBag. Crea y retorna un iterador para el *bag*.

TDA Bag: Representación e Implementación

- **REPRESENTACIÓN**:
  a. Estructura contigua.
  b. Estructura enlazada.
- **IMPLEMENTACIÓN**: Programar los métodos con la estructura de datos seleccionada, atendiendo a las particularidades de las operaciones de TDA Bag. En este caso, **no hay operaciones que usen una posición dada, pues no hay orden establecido.**

```
class Bag:

    # creación
    def __init__(self):

    # operaciones
    def len(self) -> int:

    def contains(self, elemento: object) -> bool:

    def add_item(self, elemento: object):

    def remove_item(self, elemento: object):

    def iterator(self) -> iter:
```

UNIVERSIDAD
DE MURCIA

TDAs sin orden

TDA Set

TDA Set: Especificación

- Un Bag donde **no se permite** la repetición de elementos.
- Operaciones particulares:
  - Set(): Set. Crea un nuevo conjunto, *set*, inicialmente vacío.
  - len(): int. Retorna el número de elementos en el conjunto. Para una lista cualquiera $< a_0, a_1, \ldots, a_{n-1} >$ retornará el valor n.
  - contains(element): bool. Indica si el elemento element se encuentra en el conjunto. Retorna True si está contenido y False si no está contenido.
  - add(element): None. Modifica el conjunto añadiendo el elemento element al conjunto, si no existe ya.
  - remove(element): None. Elimina el elemento element del conjunto. Lanza un error si el elemento no existe.
  - iterator(): IteratorSet: Crea y retorna un iterador para el conjunto.
- Notar que estas operaciones del *TDA Set* son análogos a los operadores del *TDA Bag*, salvo que en *add()* hace falta comprobar que el elemento no existe ya.

TDA Set: Especificación

Otras operaciones bien conocidas de los conjuntos matemáticos:

- isSubsetOf(setB): bool. Determina si un conjunto es subconjunto del conjunto dado. Un conjunto A es subconjunto de B si todos los elementos de A están en B.
- equal(setB): bool. Determina si el conjunto es igual al conjunto dado. Dos conjuntos son iguales si ambos contienen el mismo número de elementos y todos los elementos del conjunto está en el conjunto B. Si ambos están vacíos entonces son iguales.
- union(setB): Set. Retorna un nuevo conjunto que es la unión del conjunto con el conjunto dado. La unión del conjunto A con el conjunto de B es un nuevo conjunto que está formado por todos los elementos de A y todos los elementos de B que no están en A.
- difference(setB): Set. Retorna un nuevo conjunto que es la diferencia del conjunto con el conjunto dado. La diferencia del conjunto A con el conjunto de B es un nuevo conjunto que está formado por todos los elementos de A que no están en B.
- intersect(): Set. Retorna un nuevo conjunto que es la intersección del conjunto con el conjunto dado. La intersección es un nuevo conjunto que está formado por todos los elementos que están en A y también en B.

TDA Set: Representación e Implementación

- **REPRESENTACIÓN**: a Estructura contigua, b Estructura enlazada
- **IMPLEMENTACIÓN**: Equivalente a TDA Bag, salvo que no hay repetidos. **No hay operaciones con posiciones pues no hay orden.**

```
class Set:

    def __init__(self): # creación

    def len(self) -> int:    # operaciones

    def contains(self, elemento: object) -> bool:

    def add_item(self, elemento: object):

    def remove_item(self, elemento: object):

    def iterator(self) -> iter:

    def is_subset(self, other: 'Set') -> bool:

    def equal(self, other: 'Set') -> bool:

    def union(self, other: 'Set') -> 'Set':

    def difference(self, other: 'Set') -> 'Set':

    def intersect(self, other: 'Set') -> 'Set':
```

UNIVERSIDAD
DE MURCIA

TDAs sin orden

TDA Map

TDA Map: Especificación

- Colección de registros no repetidos donde cada uno consta de una clave y un valor. La clave debe ser comparable, se utiliza para acceder al valor.
- *Ejemplos*: tener las notas de los estudiantes por DNI, datos fiscales por número de identificación, datos de un conductor por matricula...
- Operaciones:
  - Map(): Map. Crea un nuevo map vacío.
  - len(): int. Retorna el número de registros clave/valor que existen.
  - contains(key): bool. Indica si la clave key se encuentra en el contenedor. Retorna True si la clave está contenida, False si no.
  - get(key): TipoValue. Retorna el valor asociado a la clave dada. La clave debe de existir en el Map.
  - set(key, value): bool. Modifica el map añadiendo el par key/value al contenedor. Si existiera un registro con la clave key se sustituye el par key/value existente por el nuevo par key/value. Retorna True si la clave es nueva y False si se realiza una sustitución.
  - remove(key): None. Elimina el registro que tiene como clave el valor key. Lanza un error si el elemento no existe.
  - iterator(): IteratorMap: Crea y retorna un iterador para el conjunto.

TDA Map: Representación e Implementación

- **REPRESENTACIÓN**: a Estructura contigua, b Estructura enlazada
  ¿Cómo representaríamos los pares clave-valor en estas estructuras?
- **IMPLEMENTACIÓN**: No hay posiciones pues no hay orden. Pero **se introduce la clave como parámetro en algunas operaciones**.

```
class Map:

    def __init__(self): # creación

    def len(self) -> int:       # operaciones

    def contains(self, key: object) -> bool:

    def get_item(self, key: object) -> object:

    def set_item(self, value: object, key: object) -> bool:

    def remove_item(self, key: object):

    def iterator(self) -> iter:
```

UNIVERSIDAD
DE MURCIA

Tema 9. Tipos de Datos Abstractos con Orden Lineal

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
1 de diciembre de 2024

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

TDAs con Orden Lineal

Qué son los TDAs con relación de orden lineal

- Los TDA con ordenación lineal son aquellos que contienen a un conjunto de objetos pero donde cada objeto tiene exactamente un predecesor inmediato o un sucesor inmediato.
- Destacan el primer objeto (que no tiene predecesor) y el último objeto (que no tiene sucesor).
- **Matemáticamente** quedan definidos mediante una relación binaria $a \preceq b$, que será una ordenación lineal sii cumple las siguientes propiedades:
  Completa o total. $\forall\ a, b$, o bien $a \preceq b$ o bien $b \preceq a$,
  Transitiva. si $a \preceq b$ y $b \preceq c$, esto implica que $a \preceq c$, y
  Antisimétrica. Si se cumple $a \preceq b$ y $b \preceq a$, se sigue que $a = b$
- Algunas operaciones que se pueden lleva a cabo en este tipo de contenedores son:
  - **Acceder** al k-ésimo objeto del orden. En particular, al primer o último objeto en el orden lineal,
  - **Recorrer** todos los objetos del contenedor en dicho orden.
  - **Buscar un objeto** a que puede, o no, estar en el contenedor.
  - **Insertar** un nuevo objeto o reemplazar el objeto en la k-ésima posición.

UNIVERSIDAD
DE MURCIA

TDA Lista

Especificación

TDA Lista: Especificación - I

- La lista se define para objetos que se quieren ordenar explícitamente. Hay un primer elemento que se coloca al inicio del contenedor y un último elemento que está al final de la lista. Los elementos centrales se colocan de acuerdo al orden lineal definido.
- Operaciones
  - List(): Lista. Crea una nueva lista, inicialmente vacía.
  - len(): int. Retorna la longitud o número de elementos en la lista. Para una lista cualquiera $< a_0, a_1, \ldots, a_{n-1} >$ retornará el valor n.
  - contains(value): bool. Indica si el valor value se encuentra na la lista.
  - getItem(pos): value. Retorna el elemento o valor almacenado en la posición pos. Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ retornará el valor a_{pos}.
  - setItem(pos, value): None. Sustituye el pos-ésimo valor de la lista por value.
    Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, value, \ldots, a_{n-1} >$.

TDA Lista: Especificación - II

- insertItem(pos, value): None. Inserta el pos-ésimo valor de la lista por value.
  Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, value, a_{pos}, \ldots, a_{n-1} >$.
- removeItem(pos): None. Elimina el elemento de la posición dada, si existe en la lista.
  Dada una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, a_{pos-1}, a_{pos+1}, \ldots, a_{n-1} >$.
- clear(): None. Limpia la lista. Se convierte en una lista vacía.
  Modifica cualquier lista a la lista $<>$.
- isEmpty(): bool. Indica si la lista está vacía o no.
- first(): value. Retorna el primer elemento de la lista. Si la lista está vacía retornará un error o valor especial None.
  Dada una lista $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se retornará a_0.
- last(): value. Retorna el último elemento de la lista. Si la lista está vacía retornará un error o valor especial None. Dada una lista $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se retornará el elemento a_{n-1}.
- iterator(): Lista: Crea y retorna un iterador para la lista.

UNIVERSIDAD
DE MURCIA

TDA Lista

Representación con estructura contigua

TDA Lista: Representación con estructura contigua

- Estructura contigua: cada elemento del array contiene un elemento de la lista, indexados/ordenados de acuerdo al orden lineal.

```
struct rep
   int longitud; # la primera posición vacía
   int capacidad; # capacidad de la lista
   object[] array;  # la lista
```

  - La función de abstracción: $Abst$: **rep** $\longrightarrow \mathcal{A}$ la definiremos como

$Abst$(r) =<r.array[0], r.array[1], ..., r.array[r.capacidad-1]>

  - El invariante de la representación I : **rep** $\longrightarrow \mathbb{B}$, lo definimos de esta manera:

$$I(\text{r}) = \begin{cases} \text{r.longitud} < \text{r.capacidad } \textit{and} \\ \forall p, p < \text{r.longitud} \Rightarrow \text{r.array[p]} \neq \textit{NULL } \textit{and} \\ \forall p, p \geq \text{r.longitud} \Rightarrow \text{r.array[p]} = \textit{NULL } \textit{and} \\ \text{r.array} \neq \textit{NULL } (\text{r.capacidad} > 0) \end{cases}$$

UNIVERSIDAD
DE MURCIA

TDA Lista

Representación con estructura enlazada

TDA Lista: Representación con estructura enlazada

- Una lista se puede representar mediante una estructura simplemente enlazada de nodos con un valor (dato) y referencia al siguiente nodo.

```
struct Nodo:  # Estructura del nodo
    TD dato
    Nodo siguiente

struct Lista:  # Lista simplemente enlazada
    node primer_nodo    # Apunta al primer nodo
    entero longitud
```

- Función de abstracción: $Abst$: **Nodo** $\longrightarrow \mathcal{A}$

$Abst(\texttt{r}) = \texttt{<r.dato, r.sig.dato, ..., r.sig.....sig.dato>}$

- El invariante de la representación I : **Nodo** $\longrightarrow \mathbb{B}$, que se define así:
  - Consta de una colección de nodos donde cada uno tiene una referencia a otro nodo (cuyo campo llamaremos siguiente o next).
  - En dos nodos diferentes la referencia next son diferentes.
  - Se tiene una variable externa que hace referencia a aquel nodo tal que a partir de él se puede recorrer todos los elementos de la lista pasando por la referencia de next de cada uno.

Recordatorio: Operaciones con Estructuras enlazadas - I

1. Recorrer los nodos.

```
actual = primer nodo de la lista
mientras actual is not None:
  trabajar con actual
  actual = actual.siguiente
```

2. Buscar un nodo.

```
actual = primer nodo de la lista
mientras actual is not None y actual.dato != target:
  actual = actual.siguiente

si actual is not None:
    trabajar con actual
```

Recordatorio: Operaciones con Estructuras enlazadas - II

3. Añadir un nodo como siguiente de otro.

```
pos = referencia # Pondremos uno delante de este
nuevo_nodo = Nodo(DATO)
nuevo_nodo = pos.siguiente
pos.siguiente = nuevo_nodo
```

4. Eliminar el nodo siguiente de otro.

```
pos = referencia # Eliminaremos el siguiente a este
borrar = pos.siguiente
pos.siguiente = borrar.siguiente
del borrar
```

Modificaciones sobre una Estructuras Enlazada Simple I

- Si se quiere añadir elementos al final, tener un campo `ultimo` al último elemento de la lista.

```
struct Nodo  # Estructura del nodo
    TD dato
    Nodo siguiente

struct Lista  # Lista simplemente enlazada
  node primer_nodo
  node ultimo_nodo # con referencia al último elemento de la lista
```

- Si hay que hacer muchos recorridos, considerar una lista doblemente enlazada.

```
struct Nodo
   TD dato
   Nodo siguiente
   Nodo anterior
```

UNIVERSIDAD
DE MURCIA

TDA Lista

Búsqueda y Ordenación en Listas

Búsqueda en listas

- Un **algoritmo de búsqueda** es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos. Por ejemplo, encontrar el registro correspondiente a cierta canción en un array de canciones.
- Dependiendo de cómo se encuentren ordenados los elementos de la colección, podemos aplicar uno de los dos siguientes algoritmos:
  - o búsqueda secuencial, cuando los elementos de la colección no están ordenados
  - o búsqueda binaria, para cuando los elementos están ordenados.

- Un **algoritmo de ordenación** es aquel diseñado para ordenar los elementos de una lista. Es importante definir vec[pos1]<vec[pos2].
- Algoritmos conocidos de ordenación, y que se consideran **ya aprendidos**:
  - Ordenación de Burbuja. Se revisa la lista una y otra vez, se intercambian dos posiciones consecutivas si no están ordenados.

    ```
    4 5 3 2
    4 3 5 2 (intercambia 5 y 3)
    4 3 2 5 (intercambia 5 y 2 ...)
    ```
  - Ordenación por Selección. Selecciona el elemento más pequeño de la lista y lo coloca en su posición. Después busca el siguiente más pequeño de entre el resto de la lista y lo coloca en su sitio, y así...

    ```
    4 5 3 2 (min = 2)
    2 5 3 4 (intercambio de posiciones)
    ```
  - Ordenación por Inserción. Divide la lista en dos partes: una ordenada y otra desordenada. En cada paso, el primer elemento de la parte desordenada se inerta en la parte ordenada.

    ```
    4 5 3 2 (ordenado hasta el 3)
    3 4 5 2 (3 se inserta en su lugar)
    ```

UNIVERSIDAD
DE MURCIA

TDA Lista

Implementación

TDA Lista: Implementación

```
class List:
    def __init__(self):

    def len(self) -> int:

    def contains(self, elemento: object) -> bool:

    def get_item(self, pos: int) -> object:

    def set_item(self, pos: int, elemento: object):

    def insert_item(self, pos: int, elemento: object):

    def remove_item(self, pos: int):

    def clear(self):

    def is_empty(self) -> bool:

    def first(self) -> object:

    def last(self) -> object:
```

UNIVERSIDAD
DE MURCIA

TDA Pila

Especificación

TDA Pila: Especificación

- Una pila es una lista con el criterio Last In First Out (LIFO).
- Operaciones
  - Stack() : Stack. Crea una nueva pila, inicialmente vacía.
  - len() : int. Retorna el número de elementos de la pila.
  - isEmpty() : Bool. Indica si la pila está vacía o no.
  - push(value) : None. Inserta un nuevo nodo en el tope de la lista.
    Para una pila $< a_0, a_1, a_2, \ldots, >$ y un valor *value* la pila se modifica para conseguir la pila $< value, a_0, a_1, a_2, \ldots, >$.
  - peek() : value. Retorna el valor del tope. También se suele usar la signatura top() : value.
    Para una pila $< a_0, a_1, \ldots, >$ retornará el valor a_0.
  - pop() : value. Retorna el valor del tope y además borra el primer nodo de la pila.
    Para una pila $< a_0, a_1, a_2, \ldots, >$ retornará el valor a_0 y la nueva pila es $< a_1, a_2, \ldots, >$.
  - clear() : None. Limpia la pila y la deja sin elementos.
- MUY IMPORTANTE: **No se puede acceder directamente a ningún elemento central**.

UNIVERSIDAD
DE MURCIA

TDA Pila

Representación e Implementación

TDA Pila: Representación e Implementación

- **REPRESENTACIÓN**:
  a Estructura contigua.
  b Estructura enlazada.
- **IMPLEMENTACIÓN**: Programar los métodos con la estructura de datos seleccionada, atendiendo a las particularidades de las operaciones de TDA Pila. En este caso, **no hay operaciones que usen una posición pues siempre se trabaja con la misma.**

```
class Stack:

    # creación
    def __init__(self):

    # operaciones
    def len(self) -> int:

    def isEmpty(self) -> bool:

    def push(self, elemento: object):

    def peek(self) -> object:

    def pop(self) -> object:

    def clear():
```

UNIVERSIDAD
DE MURCIA

TDA Cola

Especificación

TDA Cola: Especificación

- Una cola es una lista con el criterio First In First Out (FIFO).
- Operaciones
  - Queue() : Queue. Crea una nueva cola, inicialmente vacía.
  - len() : int. Retorna el número de elementos de la cola.
  - isEmpty() : Bool. Indica si la cola está vacía o no.
  - enqueue(elemento) : None. Añade un nuevo elemento al final de la cola
    Para la cola $< a_0, a_1, \ldots, a_{n-1} >$ se modificará a la lista
    $< a_0, a_1, \ldots, a_{n-1}, value >$
  - peek() : value. Retorna el valor del primer elemento de la lista, pero no lo borra. También es usual esta signatura front() : value.
  - dequeue() : value. Retorna el primer elemento de la cola borrandolo de la cola. También es usual esta signatura top() : value.
    Para la cola $< a_0, a_1, \ldots, >$ retornará el valor a_0. Se genera un error si la cola está vacía.
  - clear() : None. Limpia la cola y la deja sin elementos.
- MUY IMPORTANTE: **No se puede acceder directamente a ningún elemento central.**

UNIVERSIDAD
DE MURCIA

TDA Cola

Representación e Implementación

TDA Cola: Representación e Implementación

- **REPRESENTACIÓN**:
  a Estructura contigua.
  b Estructura enlazada.
- **IMPLEMENTACIÓN**: Programar los métodos con la estructura de datos seleccionada, atendiendo a las particularidades de las operaciones de TDA Cola. En este caso, **no hay operaciones que usen una posición pues siempre se trabaja con la primera o última.**

```
class Queue:

    # creación
    def __init__(self):

    # operaciones
    def len(self) -> int:

    def is_empty(self) -> bool:

    def enqueue(self, elemento: object):

    def peek(self) -> object:

    def dequeue(self) -> object:

    def clear():
```

UNIVERSIDAD
DE MURCIA

Tema 10. Tipos de Datos Abstractos Sin Orden Lineal

Tecnología de la Programación

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

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
9 de diciembre de 2024

Índice de Contenidos

UNIVERSIDAD
DE MURCIA

TDAs sin Orden Lineal

Qué son los TDAs sin relación de orden lineal

- Los TDA con ordenación lineal son aquellos que contienen a un conjunto de objetos pero donde cada objeto tiene exactamente un predecesor inmediato o un sucesor inmediato.
- Los TDA sin ordenación lineal son aquellos que contienen a un conjunto de objetos pero donde cada objeto puede tener de 0 a varios sucesores:
  - Los elementos tienen nombre en un orden jerárquico:
    - El primer elemento de la ordenación recibe el nombre de **raíz**, y por tanto cumple que solo tiene descendientes y no tiene padre.
    - Los elementos que no tienen hijos se llaman **hojas**.
    - Los que no son ni hojas ni raíz se llaman **intermedios**.
    - Si *a* es **padre** de *b*, *b* es **hijo** de *a*.
    - Si *a* es **ascendiente** o ancestro de *b*, entonces *b* es **descendiente** o sucesor de *a*.
- **Ejemplos**: Sistema de archivos (carpetas y ficheros), árbol genealógico, jerarquías organizacionales/empresariales, ...

UNIVERSIDAD
DE MURCIA

TDA Árbol

Especificación

TDA Árbol: Definición Recursiva

- Un árbol responde a la siguiente definición recursiva que consta de los siguientes casos:
  - **Caso base:** El conjunto vacío es un árbol. El árbol vacío.
  - **Caso recursivo:** Un árbol es una colección de datos que está formada por un dato, r, y una lista de 0 o más árboles, $A_1, A_2, \ldots A_n$.
- Se deberá cumplir una ordenación jerárquica donde:
  - Cada dato r será el elemento raíz del árbol que se esté definiendo, y será padre de las raíces de los árboles A_i.
  - Los árboles A_i se llaman subárboles del árbol cuya raíz es r.

TDA Árbol: Especificación - I

- Operaciones básicas. Se trabaja con los nodos (pos) del árbol:
  - Tree(): Tree. Crea un nuevo árbol
  - append(value, node=pos): nuevo_nodo. Añade un nuevo nodo hoja al árbol con el valor value. Si se especifica el segundo parámetro, el nuevo nodo será hijo del nodo pos. Sino, se puede recorrer en anchura e insertar en el primer hijo vacío. Devuelve el nodo creado.
  - value(pos): type_value. Retorna el contenido del nodo (posición) pos.
  - replace(pos, value): value. Cambia el valor del nodo de la posición pos por un nuevo valor (el de entrada) y retorna el antiguo valor.
  - remove(pos): None. Elimina el nodo de la posición pos y sus hijos.
  - parent(pos): pos. Retorna la posición del padre para la posición pos. Será None si la posición de entrada se corresponde con la raíz.
  - children(pos): container. Retorna un contenedor iterable con todos los hijos inmediatos de pos.
  - positions(): container. Retorna un contenedor iterable con todos los nodos del árbol.

TDA Árbol: Especificación - II

- elements(): container. Retorna un contenedor iterable con todos los valores del árbol.
- num_children(pos): int. Indica el número de hijos que tiene el nodo pos.
- len(): int. Retorna el número de elementos que tiene el árbol.
- depth(pos): int. Retorna la profundidad (número de predecesores desde la raíz) del nodo pos.
- height(pos): int. Retorna la altura (camino más largo hasta una de sus hojas) del nodo pos.
- root(): pos. Retorna la posición del nodo raíz del árbol.
- isRoot(pos): Bool. Retorna *True* si la posición pos es el nodo raíz del árbol. Retorna *False* en otro caso.
- isInternal(pos): Bool. Retorna *True* si la posición pos es el de un nodo interno. Retorna *False* en otro caso.
- isLeaf(pos): Bool. Retorna *True* si pos es un nodo hoja del árbol. Retorna *False* en otro caso.
- isEmpty(): Bool. Indica si el árbol está vacío o no.

TDA Árbol: Especificación - III

- Se pueden ampliar con una gran cantidad de métodos como, por ejemplo:
  - borrar(): None. Borrar los nodos empezando por el nivel más profundo.
  - contar(criterio): int. Contar el número de elementos que cumplan cierto criterio.
  - buscar(valor): bool. Buscar un elemento en el árbol.
  - altura(): int. Calcula la altura del árbol.
  - hojas(): int. Calcula el número de hojas del árbol.

UNIVERSIDAD
DE MURCIA

TDA Árbol

Representación

TDA Árbol: Representación

- Estructura:

```
struct Nodo
   TDato dato
   Secuencia nodosHijos

struct Arbol
   Nodo raiz
```

- Un árbol vacío es aquel que cumpla arbol.raiz = None.
- Notar que al ser un Árbol una composición de nodos se debe aplicar delegación.
  Por ejemplo: dos árboles son iguales si lo son cada uno de sus nodos (el dato y la secuencia de hijos).
- Nota: En un árbol **N-ario** cada nodo puede tener un número fijo de hijos.

UNIVERSIDAD
DE MURCIA

TDA Árbol Binario

Especificación

TDA Árbol Binario: Especificación

- **Definición:** Un **árbol binario** es un árbol que **siempre** tiene dos subárboles que reciben el nombre de hijo (o subárbol) izquierdo e hijo (o subárbol) derecho. En un árbol binario los subárboles pueden ser vacíos.
- **Operaciones**:
  - append(value, node=pos): nuevo_nodo. Añade un nuevo nodo hoja al árbol con el valor value. Si se especifica el segundo parámetro, el nuevo nodo será hijo del nodo pos. Sino, se puede recorrer en anchura e insertar en el primer hijo vacío. Devuelve el nodo creado.
  - remove(pos): None. Elimina el nodo pos y sus hijos.
  - mostrar(arbol): Mostrar los elementos de un árbol como si fuera una estructura de directorio.
  - contar(arbol): int: Contar el número de elementos que cumplan cierto criterio.
  - buscar(arbol, valor): bool: Buscar un elemento en el árbol.
  - altura(arbol): int: Calcular la altura de un árbol.
  - hojas(arbol): int: Calcular el número de hojas.
  - Y hay más ....

UNIVERSIDAD
DE MURCIA

TDA Árbol Binario

Representación

TDA Árbol Binario: Representación

- Estructura:

```
struct Nodo
   TDato dato
   Nodo hijoIzquierdo
   Nodo hijoDerecho

struct ArbolBinario
   Nodo raiz
```

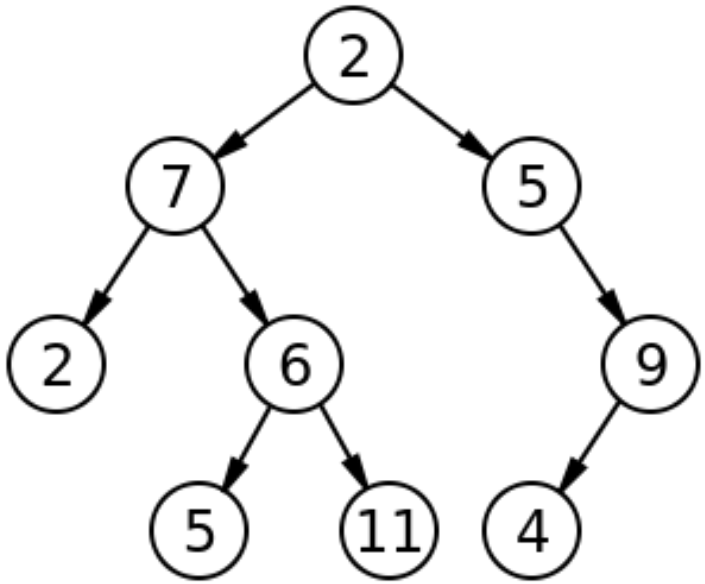


Figura: Árbol binario

UNIVERSIDAD
DE MURCIA

TDA Árbol Binario

Implementación

TDA Árbol Binario: Algunas indicaciones

El **recorrido en profundidad** de los nodos (DFS, Depth-First Search) se puede implementar de tres formas recursivas según el orden de visita:

- Preorden (Raíz, Izquierdo, Derecho)

```
DFS_Preorden(nodo):
    si nodo != null:
        accion(nodo.dato)                  // Procesar la raíz
        DFS_Preorden(nodo.hijoIzquierdo) // Recorrer subárbol izq.
        DFS_Preorden(nodo.hijoDerecho)   // Recorrer subárbol der.
```

- Inorden (Izquierdo, Raíz, Derecho)

```
DFS_Inorden(nodo):
    si nodo != null:
        DFS_Inorden(nodo.hijoIzquierdo) // Recorrer subárbol izq.
        accion(nodo.dato)                // Procesar la raíz
        DFS_Inorden(nodo.hijoDerecho)   // Recorrer subárbol der.
```

- Postorden (Izquierdo, Derecho, Raíz)

```
DFS_Postorden(nodo):
    si nodo != null:
        DFS_Postorden(nodo.hijoIzquierdo) // Recorrer subárbol izq.
        DFS_Postorden(nodo.hijoDerecho)   // Recorrer subárbol der.
        accion(nodo.dato)                  // Procesar la raíz
```

TDA Árbol Binario: Algunas indicaciones

El **recorrido en anchura** de todos los nodos (BFS, Breadth-First Search) visita los nodos nivel por nivel, utilizando una cola para manejar los nodos pendientes.

```
BFS(raiz):
    si raiz == None:
        devolver
    cola = Cola()
    cola.encolar(raiz)                  // Inicia con la raíz
    mientras cola no esté vacía:
        nodo = cola.desencolar()        // Procesar la cabeza de la cola
        accion(nodo.dato)               // Procesar e
        si nodo.hijoIzquierdo != None:
            cola.encolar(nodo.hijoIzquierdo) // Hijo izq. a la cola
        si nodo.hijoDerecho != None:
            cola.encolar(nodo.hijoDerecho)   // Hijo der. a la cola
```

También se puede hacer un **recorrido en profundidad** *(iterativo) usando una pila.*

TDA Árbol Binario: Implementación

```
class NodoArbol:
    def __init__(self):

class ArbolBinario:
    def __init__(self):

    def append(self, valor: object, posicion: NodoArbol = None) -> NodoArbol:

    def remove_recursivo(self, posicion: NodoArbol, nodo_actual: NodoArbol=None

    def remove_iterativo(self, posicion: NodoArbol):

    def count_recursivo(self, value, nodo_actual=None):

    def count_iterativo(self, value):

    ...
```