

# Lenguaje para la clase:



¿Por qué Python?

- **Eficiencia:** Python permite lograr más con menos líneas de código en comparación con otros lenguajes.
- **Sintaxis clara:** La sintaxis de Python fomenta la escritura de código limpio, fácil de leer, depurar, extender y mejorar.
- **Versatilidad:** Python se utiliza en diversos campos como desarrollo de juegos, aplicaciones web, resolución de problemas empresariales, herramientas internas y trabajo científico/académico.
- **Comunidad:** La comunidad Python es diversa y acogedora, lo cual es fundamental para los programadores, especialmente para quienes están aprendiendo o vienen de otros lenguajes. La comunidad ofrece apoyo para resolver problemas.
- **Ideal para aprender:** Python es un excelente lenguaje para aprender a programar.

# Lenguaje para la clase:

Python es un **lenguaje interpretado**, lo que significa que su código no se compila previamente a un lenguaje máquina, sino que se ejecuta línea por línea a través de un **intérprete**.



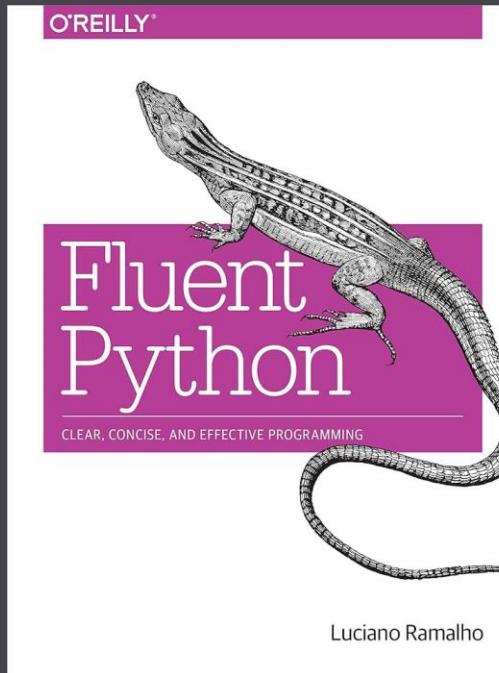
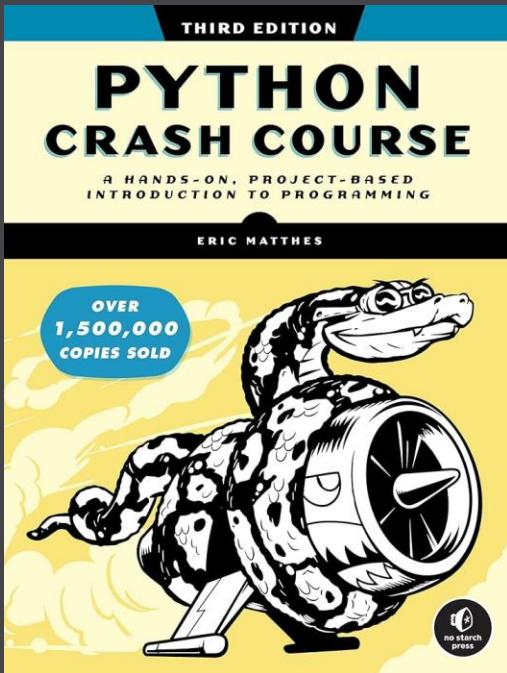
## ¿Qué es un intérprete en Python?

Un **intérprete** es un programa que traduce y ejecuta el código fuente de Python en tiempo real. En lugar de compilar todo el código antes de ejecutarlo (como en lenguajes como C o Java), el intérprete lee cada línea del código, la convierte en instrucciones comprensibles para la computadora y la ejecuta inmediatamente.

Interprete → `#!/usr/bin/env python3  
# -*- coding: utf-8 -*-`

← Porque latam

# Bibliografía



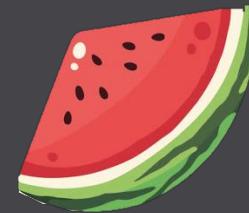
[Documentación de python](#)

# Importancia de la Programación Orientada a Objetos (POO) en el desarrollo de software.

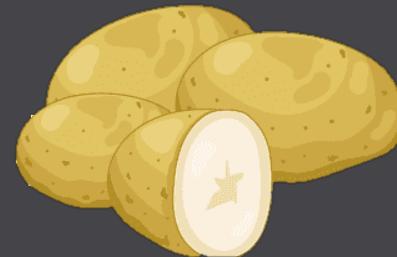
La programación orientada a objetos (POO) es uno de los los enfoques más eficaces para escribir software.

En la programación orientada a objetos se escriben **clases** que representan **cosas y situaciones** del mundo, y se crean objetos basados en estas clases. Cuando se escribe una clase, se define el comportamiento general que puede tener toda una categoría de objetos.

# Importancia de la Programación Orientada a Objetos (POO) en el desarrollo de software.



# Importancia de la Programación Orientada a Objetos (POO) en el desarrollo de software.



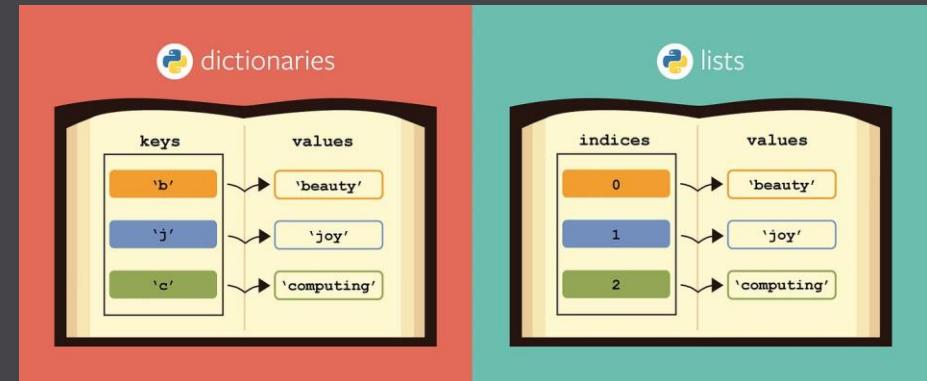
# Diccionario

Un diccionario en Python es una **colección de pares clave-valor**. Cada clave está conectada a un valor, y puedes usar una clave para acceder al valor asociado a esa clave. El valor de una clave puede ser un número, una cadena, una lista o incluso otro diccionario

```
Fruta = { 'color': 'verde', 'precio': 5 }
```

Clave

Valor



# Diccionarios

A veces queremos almacenar varios diccionarios en una lista, o una lista de elementos como valor en un diccionario. Esto se denomina **anidamiento**. Puede anidar diccionarios dentro de una lista, una lista de elementos dentro de un diccionario, o incluso un diccionario dentro de otro diccionario.

```
#Lista de diccionarios
piña = {'color': 'amarillo', 'peso':200}
mandarina = {'color': 'naranja', 'peso':50}
manzana = {'color': 'rojo', 'peso':25}

frutas=[piña, mandarina,manzana]
print(frutas)
```

```
#A Listas en un diccionario
#Guardar informacion sobre una pizza siendo ordenada
pizza = {
'orilla': 'rellena',
'ingredientes': ['queso', 'champiñones'],
}
```

# Diccionarios

Ejercicio:

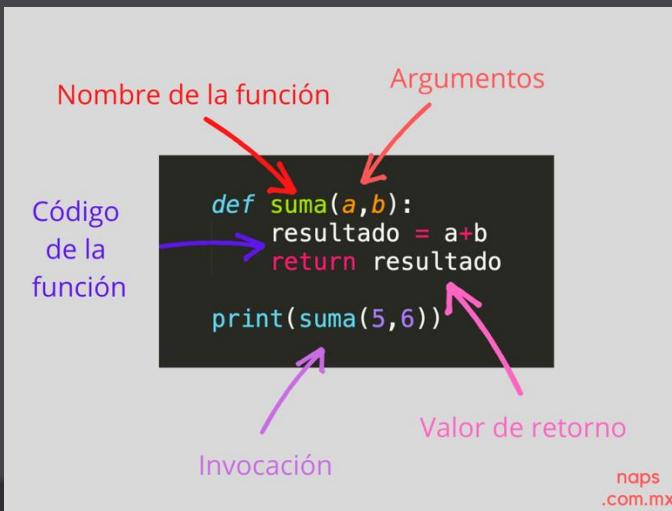
- Haz un diccionario llamado ciudades.
- Utiliza los nombres de tres ciudades como claves del diccionario.
- Crea un diccionario con información sobre cada ciudad eciudad, su población aproximada y un dato sobre sobre esa ciudad.
- Las claves del diccionario de cada ciudad deben ser algo así como *país*, *población* y *dato*.
- Imprime el nombre de cada ciudad y toda la información que hayas almacenado sobre ella.

Al inicio del programa desplegar los paises al usuario y preguntarle que informacion quiere saber de ella

Imprimir la informacion que pide.

# Funciones

Cuando quieras realizar una tarea concreta que hayas definido en una función, llama a la función responsable de ella. Si necesitas realizar esa tarea varias veces a lo largo de tu programa, no necesitas escribir todo el código para la misma tarea una y otra vez;



# Funciones

Ejercicio:

## **Calculadora de Frutas:**

- Crea una función que reciba el nombre de una fruta y la cantidad deseada.
- La función debe devolver el precio total de la compra (puedes asignar precios fijos a cada fruta).
- Ejemplo: calcular\_precio\_fruta("manzana", 5) debería devolver el precio de 5 manzanas.

## **Inventario de Frutas:**

- Crea un diccionario donde las claves sean nombres de frutas y los valores sean la cantidad en stock.
- Crea una función que reciba el nombre de una fruta y la cantidad a agregar al inventario.
- Crea otra función que reciba el nombre de una fruta y la cantidad a restar del inventario (si hay suficiente stock).

## **Frutas por Temporada:**

- Crea un diccionario donde las claves sean nombres de frutas y los valores sean la temporada en la que se cultivan (ej: "manzana": "otoño").
- Crea una función que reciba una temporada y devuelva una lista de frutas que se cultivan en esa temporada.

# Importancia de la Programación Orientada a Objetos (POO) en el desarrollo de software.

La programación orientada a objetos (POO) es uno de los los enfoques más eficaces para escribir software.

En la programación orientada a objetos se escriben **clases** que representan **cosas y situaciones** del mundo, y se crean objetos basados en estas clases. Cuando se escribe una clase, se define el comportamiento general que puede tener toda una categoría de objetos.

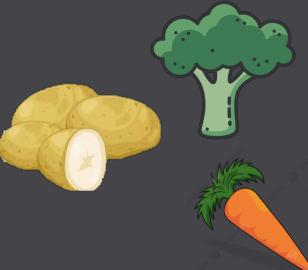
# Importancia de la Programación Orientada a Objetos (POO) en el desarrollo de software.

Hortaliza

Verduras

S

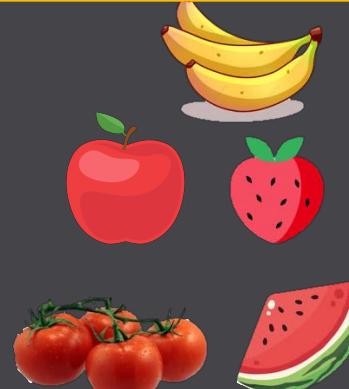
Frutas



Color

Precio

sabor



# Método: \_\_init\_\_()

```
dog.py ❶ class Dog:  
    """A simple attempt to model a dog."""  
  
❷     def __init__(self, name, age):  
        """Initialize name and age attributes."""  
❸         self.name = name  
        self.age = age  
                Valores de instancia  
  
❹     def sit(self):  
        """Simulate a dog sitting in response to a command."""  
        print(f"{self.name} is now sitting.")  
  
    def roll_over(self):  
        """Simulate rolling over in response to a command."""  
        print(f"{self.name} rolled over!")
```

- Debe definirse con tres parámetros:

- self: referencia a la propia **instancia**.
  - name y age: atributos específicos de la instancia.

- self se pasa automáticamente en las llamadas a métodos.

Los atributos pueden ser accedidos desde cualquier método de la clase o desde una instancia.

## Métodos en una Clase

- Una función dentro de una clase se llama **método**.
- Los métodos funcionan como funciones normales, pero se llaman de manera diferente.

## Método Especial \_\_init\_\_()

- Se ejecuta automáticamente al crear una instancia de la clase.
- Tiene doble guion bajo antes y después (`__init__`) para evitar conflictos con nombres reservados de Python.

# Clases

tarea:

- Restaurante: Crea una clase llamada Restaurante. El método `__init__()` de `Restaurant` debe almacenar dos atributos: un `nombre_restaurante` y un `tipo_de_cocina`.  
Crea un método llamado `describe_restaurant()` que imprima esta información la información del `self`, y un método llamado `open_restaurant()` que imprima si el restaurante está cerrado o está abierto.  
Crea una instancia llamada `restaurante` a partir de tu clase. Imprime los dos atributos individualmente, y luego llame a ambos métodos.
- Tres restaurantes: Empiece con su clase del Ejercicio anterior. Cree tres instancias diferentes de la clase y llame a `describe_restaurant()` para cada instancia.
- Usuarios: Crea una clase llamada `Usuario`. Crea dos atributos llamados `first_name` y `last_name`, y añadir otros atributos que se almacenan típicamente en un perfil de usuario.  
Crea un método llamado `describe_user()` que imprima un resumen de la información del usuario.  
Crea otro método llamado `greet_user()` que imprima un saludo personalizado al usuario.  
Crea varias instancias que representen diferentes usuarios, y llama a ambos métodos para cada usuario para cada usuario

Las **variables de instancia** pertenecen a un objeto específico y se definen con **self.atributo**, todas las variables dentro de clase pueden ser llamadas o modificadas

```
class Car:

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
   ❶    self.odometer_reading = 0

    def get_descriptive_name(self):
        --snip--

   ❷    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

my_new_car = Car('audi', 'a4', 2024)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

Las **variables de default** no definidas en el `__init__` pero si como un **self.atributo**, pueden usarse dentro de los métodos y asignarse después.



```
2024 Audi A4
This car has 0 miles on it.
```

Tambien podemos **modificar el valor** de un atributo mediante un **método**

```
class Car:  
    --snip--  
  
    def update_odometer(self, mileage):  
        """Set the odometer reading to the given value."""  
        self.odometer_reading = mileage  
  
my_new_car = Car('audi', 'a4', 2024)  
print(my_new_car.get_descriptive_name())  
  
❶ my_new_car.update_odometer(23)  
my_new_car.read_odometer()
```

Tambien podemos **aumentar el valor** de un atributo mediante un **método**

```
class Car:  
    --snip--  
  
    def update_odometer(self, mileage):  
        --snip--  
  
    def increment_odometer(self, miles):  
        """Add the given amount to the odometer reading."""  
        self.odometer_reading += miles  
  
❶ my_used_car = Car('subaru', 'outback', 2019)  
print(my_used_car.get_descriptive_name())  
  
❷ my_used_car.update_odometer(23_500)  
my_used_car.read_odometer()  
  
my_used_car.increment_odometer(100)  
my_used_car.read_odometer()
```

El valor se ingrementa, a partir del valor anterior

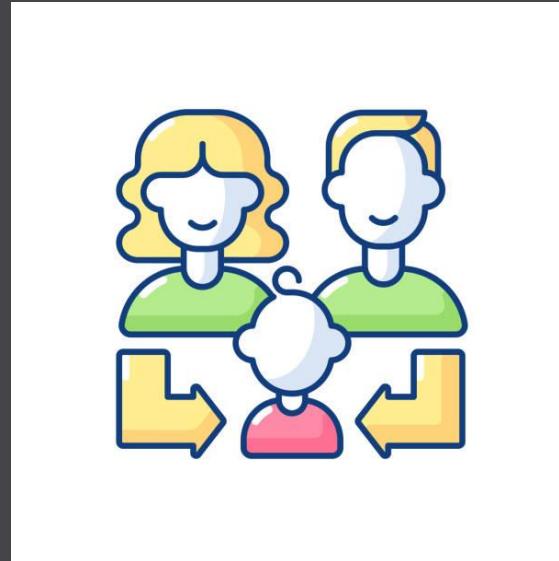
# Clases

tarea:

- Restaurante-2:
- Añade un atributo llamado número\_servido con un valor por defecto de 0.
- Crea una instancia llamada restaurante a partir de esta clase. Imprime el número de clientes que el restaurante ha servido, y luego cambie este valor e imprímalo de nuevo.
- Añade un método llamado set\_number\_served() que te permita establecer el número de clientes que han sido servidos.
- Llame a este método con un nuevo número e imprima el valor de nuevo.
- Añade un método llamado increment\_number\_served() que te permite incrementar el número de clientes que han sido atendidos.
- Llame a este método con cualquier número que represente el número de clientes atendidos en, digamos, un día de negocio.

# Herencias

- La **herencia** permite que una clase hija herede atributos y métodos de una clase padre.
- Evita la duplicación de código y facilita la reutilización.
- Se define pasando la clase padre como argumento en la declaración de la clase hija.



# Herencias

```
❷ class ElectricCar(Car): ← Nombre del padre
    """Represent aspects of a car, specific to electric vehicles."""

❸     def __init__(self, make, model, year):
        El método __init__() recoge la información
        necesaria para crear una instancia de Car
        """Initialize attributes of the parent class."""
❹     super().__init__(make, model, year)

❺ my_leaf = ElectricCar('nissan', 'leaf', 2024)
print(my_leaf.get_descriptive_name())
```

(4) **super()** es una función especial que permite llamar a un método desde la clase padre. Esta línea le dice a Python que llame al método `__init__()` de `Car`, que da a una instancia de `ElectricCar` todos los atributos definidos en ese método

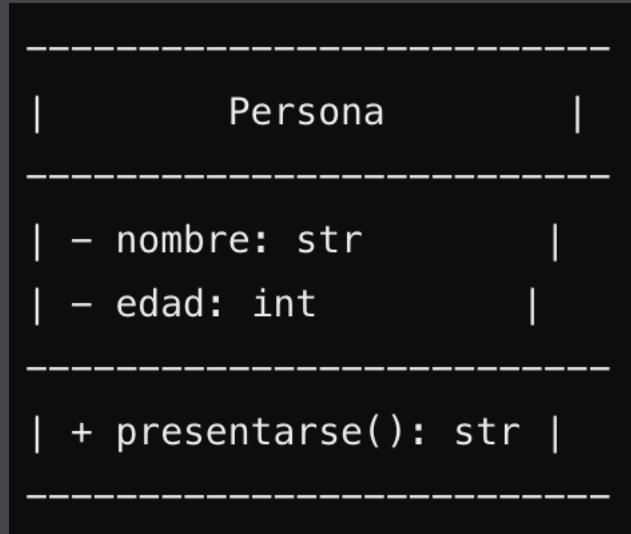
# UML

- Diagramas:

UML (**Unified Modeling Language**) es un lenguaje gráfico para modelar sistemas de software.

Un **diagrama de clases UML** representa **clases, atributos, métodos y relaciones** entre ellas.

- Es estandarizado, constituye un método versátil, flexible y fácil de utilizar para visualizar el diseño de un sistema. Los objetos del sistema de software pueden especificarse, visualizarse, construirse y documentarse con el uso de UML.

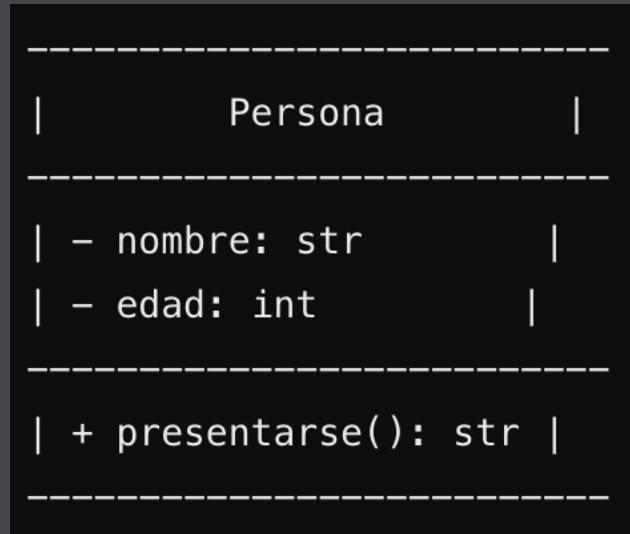


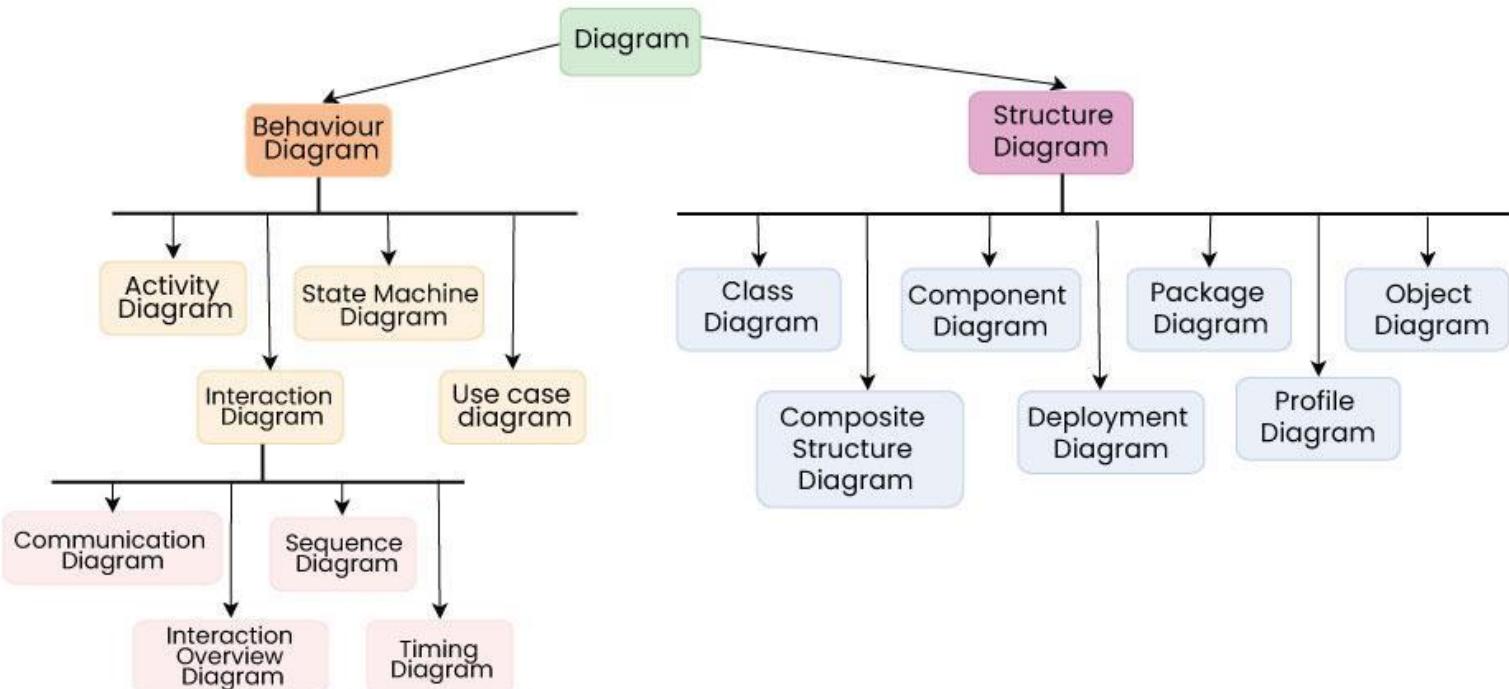
Diagramas:

# UML

- Persona es la clase, nombre y edad los atributos de instancia.

- indica atributo privado
- + indica método público.
- (()) presentarse() es un método.



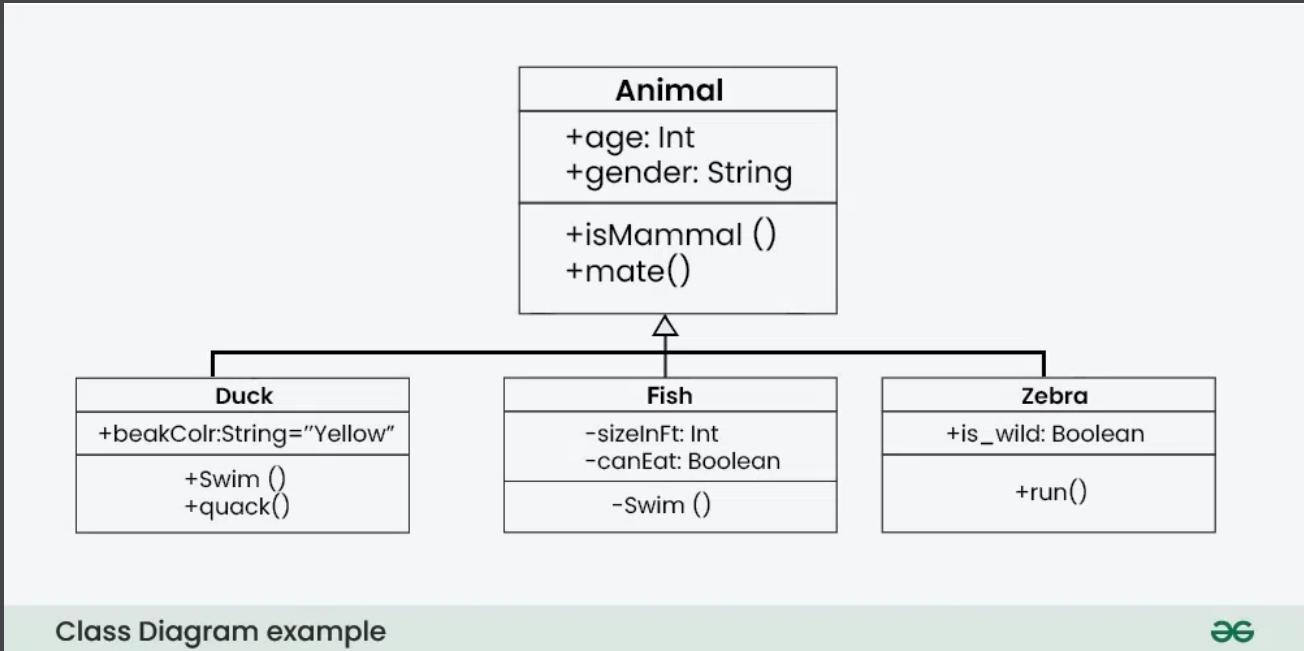


## UML Diagrams

# Diagramas UML estructurales

- Los diagramas UML estructurales son representaciones visuales que representan los aspectos estáticos de un sistema, incluidas sus clases, objetos, componentes y sus relaciones, proporcionando una visión clara de la arquitectura del sistema.

Class Diagram

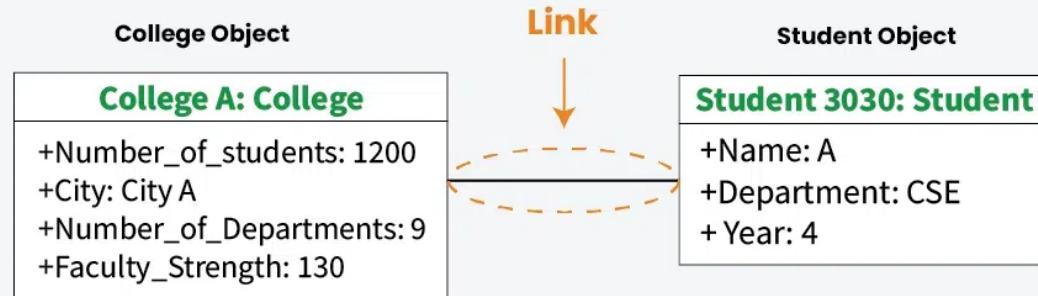


Un diagrama --> captura de pantalla de las instancias de un sistema y de la relación que existe entre ellas.

Dado que los diagramas de objetos muestran el comportamiento de los objetos cuando han sido instanciados --> estudiar el comportamiento del sistema en un instante determinado.

- Un diagrama de objetos es similar a un diagrama de clases,--> instancias de las clases del sistema.
- Mediante los diagramas de clases representamos las clases reales y sus relaciones.
- Por otro lado, un diagrama de objetos representa instancias específicas de clases y las relaciones entre ellas en un momento dado.

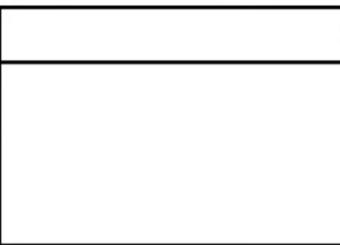
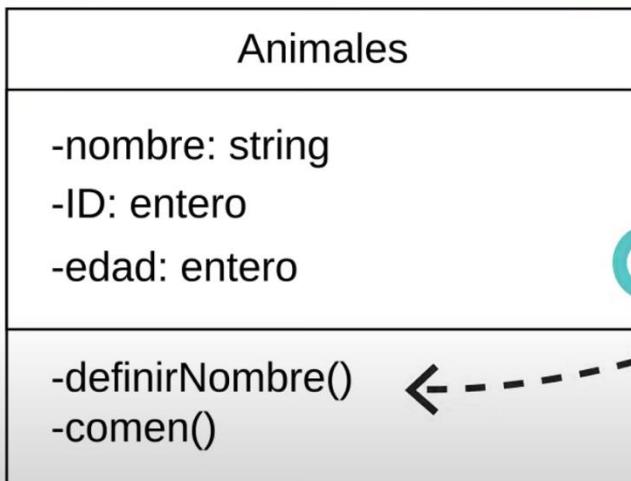
### An object diagram using a link and 2 objects



An object of class Student is linked to an object of class College.

# SISTEMA ZOOLÓGICO

## Paquete



## VISIBILIDAD

- privado
- + público
- # protegido
- ~ paquete/defecto

# RELACIONES

- herencia →

Animales
-nombre: string
-ID: entero
-edad: entero
-definirNombre()
-comen()



# RELACIONES

- herencia →

Animales
-nombre: string
-ID: entero
-edad: entero
-definirNombre()
-comen()

tortugas

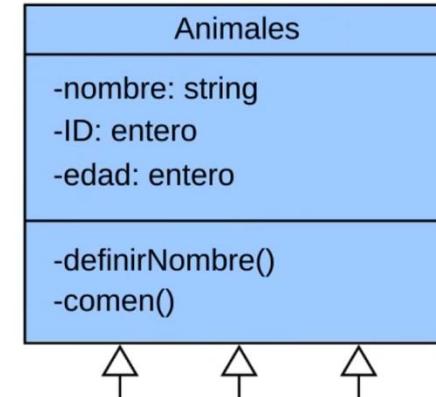
nutrias

loris

# RELACIONES

- herencia →

clase derivada  
subclases



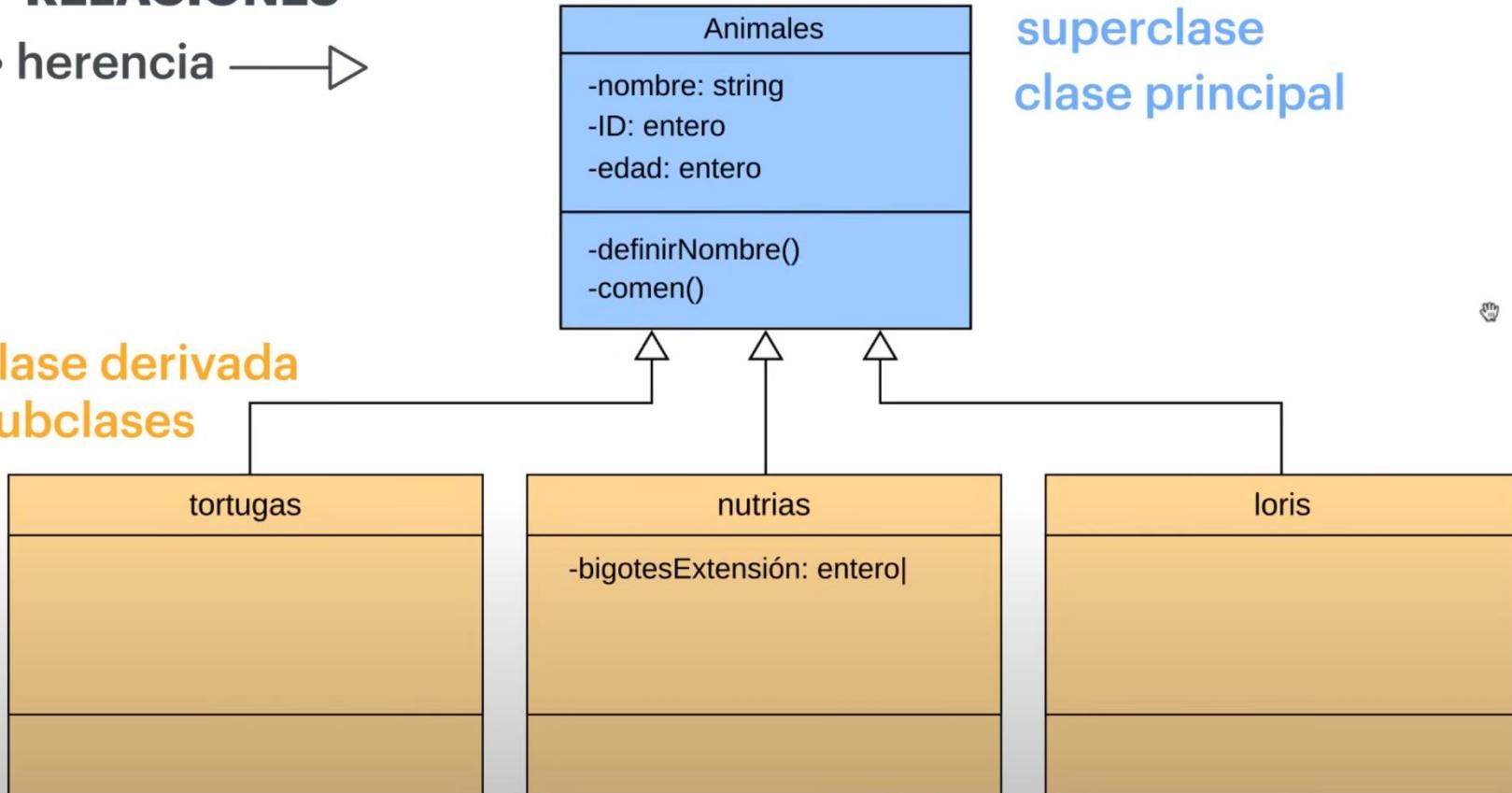
superclase  
clase principal



# RELACIONES

- herencia →

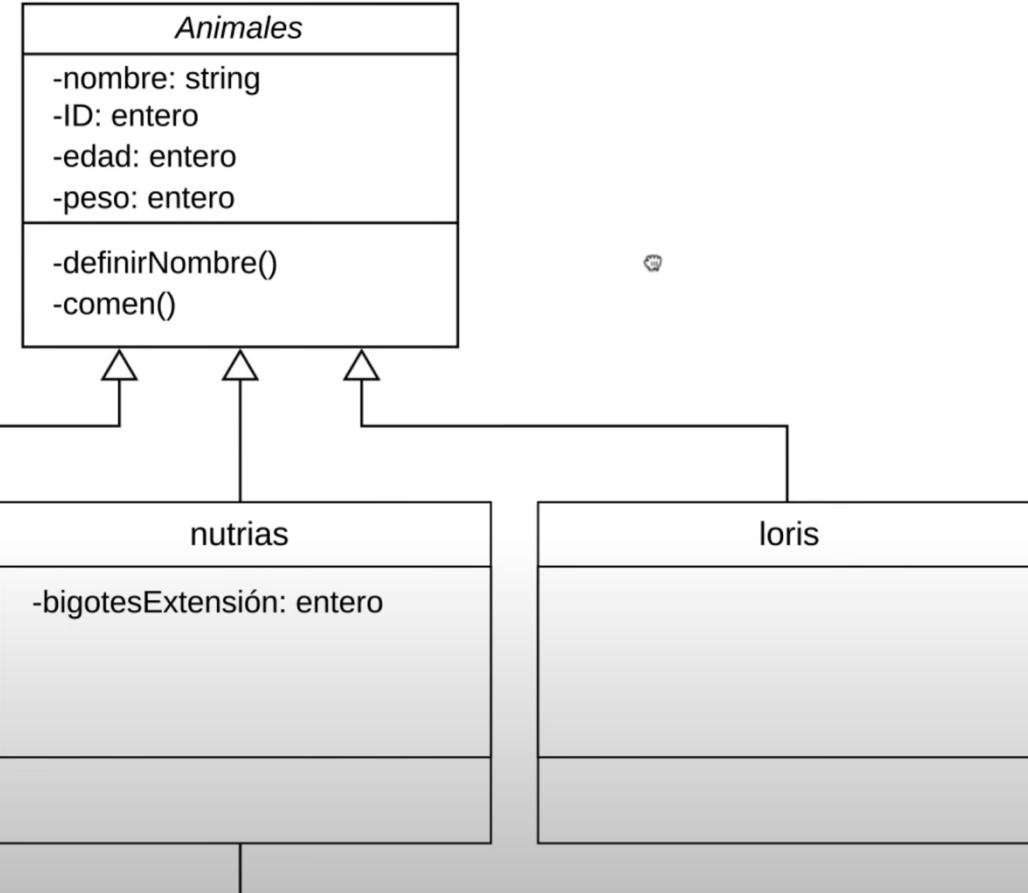
clase derivada  
subclases



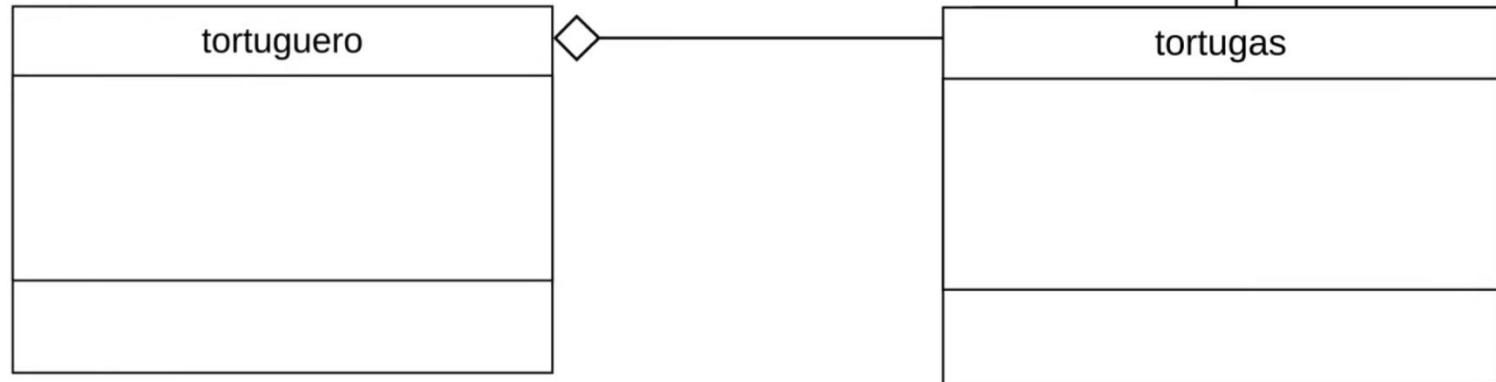
superclase  
clase principal

# RELACIONES

- herencia →
- asociación —
- agregación —◇
- composición —◆



## Agregación: un objeto puede existir dentro o fuera de la clase



# Composición: un objeto puede no existir sin el otro

Centros de Visitantes



Recepción



Baño



## Composición: un objeto puede no existir sin el otro

Centros de Visitantes



Recepción



Baño



## MULTIPLICIDAD

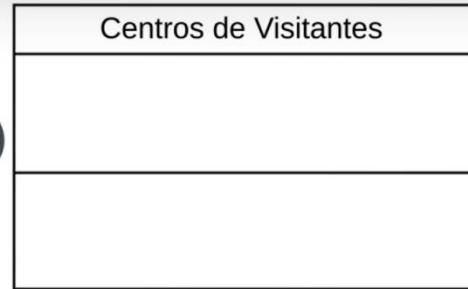
0..1 cero a uno (opcional)

n (cantidad específica)

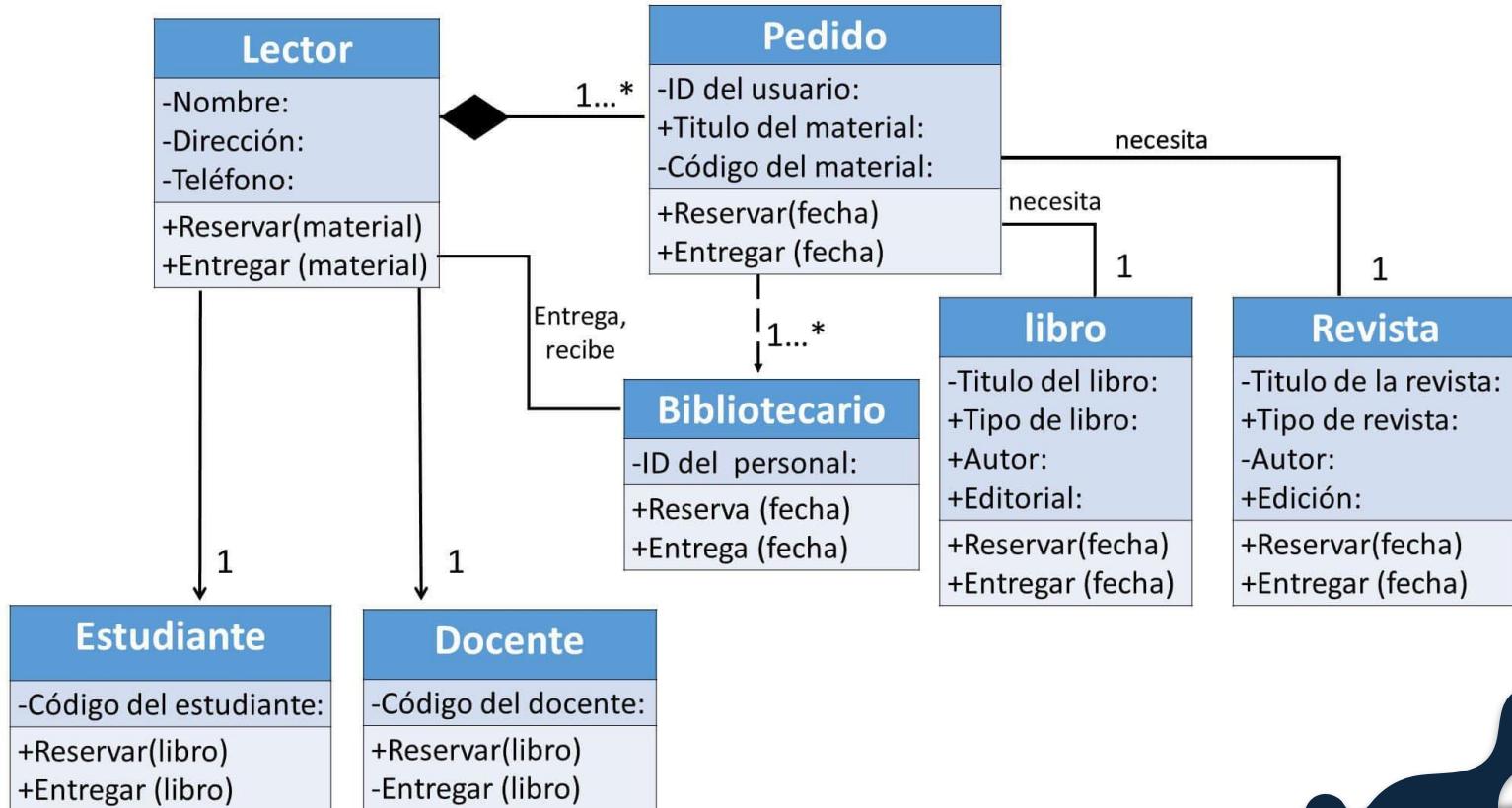
0..\* cero a muchos

1..\* uno a muchos

m..n rango específico



# Diagrama de clases de un sistema de servicios bibliotecarios



# Descriptores de Acceso (@property)

## Getters

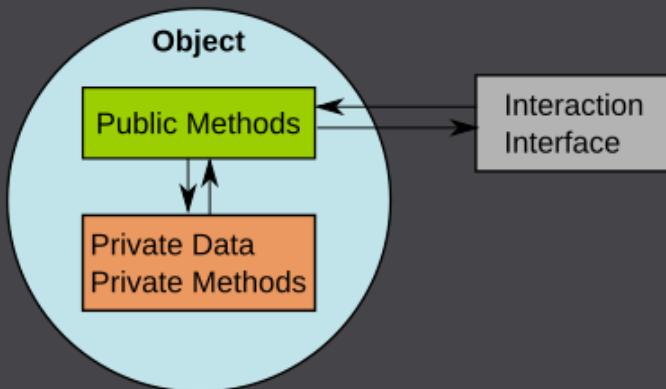
## Setters

Los “Getters” y “Setters” se utilizan en POO para garantizar el principio de la encapsulación de datos.

La encapsulación se refiere a la agrupación de datos --> la restricción del acceso directo a algunos de los componentes de un objeto.

### Objetivo

- Ocultar valores o estado de un objeto de datos en una clase



### Beneficios

- Evitar acceso directo por parte de clientes
- Proteger detalles de implementación
- Mantener invariancia de estado por métodos

# Inicialización de Objetos con Constructores (dunders)

- Existen métodos especiales, en [Python](#) llamados "dunder methods"
- Rodeados de dobles guiones bajos (`__`)
- Definen [comportamiento de objetos](#)
- Usados en operaciones comunes  
sumas, comparaciones, conversión a cadenas, iteración y más

- Un constructor `(__init__)` es un método especial que se ejecuta automáticamente al crear un objeto.
  - Se usa para inicializar atributos con valores iniciales.
  - Puede recibir parámetros opcionales con valores por defecto.

\* La documentación de Python se refiere a ellos como *métodos especiales* y señala el sinónimo «método mágico», pero muy raramente utiliza el término «método dunder». Sin embargo, «dunder method» es un coloquialismo bastante común en Python.

# Descriptores de Acceso (@property)

## Getters

Los “Getters” y “Setters” se utilizan en POO para garantizar el principio de la encapsulación de datos.

```
class Clase:  
    def __init__(self, mi_atributo):  
        self.__mi_atributo = mi_atributo  
  
    @property  
    def mi_atributo(self):  
        return self.__mi_atributo  
  
    @mi_atributo.setter  
    def mi_atributo(self, valor):  
        if valor != "":  
            print("Modificando el valor")  
            self.__mi_atributo = valor  
        else:  
            print("Error está vacío")
```

## Setters

En Python, usamos `@property` para definir **propiedades controladas** sin necesidad de métodos como `get_atributo()` y `set_atributo()`

- `_atributo` es una convención, no una restricción real
- `@property` nos permite usar getters y setters sin cambiar la sintaxis habitual.
- Los setters nos ayudan a validar datos antes de asignarlos.

En **Python**, no es realmente privado, solo es una **convención**. El acceso sigue siendo posible.

# Inicialización de Objetos con Constructores (dunders)

## The 3 essential dunder methods

There are 3 dunder methods that *most* classes should have: `__init__`, `__repr__`, and `__eq__`.

Operation	Dunder Method Call	Returns
<code>T(a, b=3)</code>	<code>T.__init__(x, a, b=3)</code>	<code>None</code>
<code>repr(x)</code>	<code>x.__repr__()</code>	<code>str</code>
<code>x == y</code>	<code>x.__eq__(y)</code>	Typically <code>bool</code>

El método `__init__` es el inicializador (no confundir con el constructor),

El método `__repr__` personaliza la representación en cadena de un objeto,

El método `__eq__` personaliza lo que significa que los objetos sean iguales entre sí.

# Ejercicio: Sistema de Gestión de Productos

## Instrucciones:

Crea la clase Producto con los siguientes atributos privados:

- `_nombre` (str)
- `_precio` (float)
- `_stock` (int)

Usa `@property` y `@setter` para:

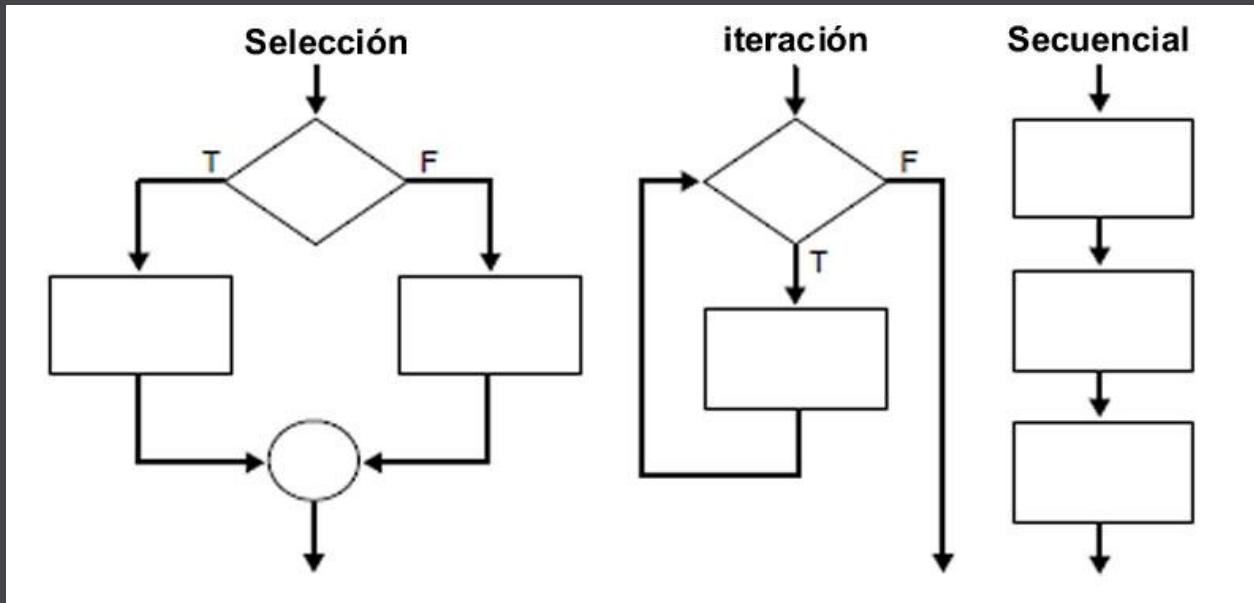
- Asegurar que el precio sea positivo.
- Asegurar que el stock sea un número entero positivo.

Implementa los siguientes dunder methods:

- `__str__`: Devuelve una representación legible del producto.
- `__repr__`: Devuelve una representación más técnica del objeto.
- `__eq__`: Permite comparar productos por nombre y precio.
- `__add__`: Permite sumar el stock de dos productos si tienen el mismo nombre y precio.

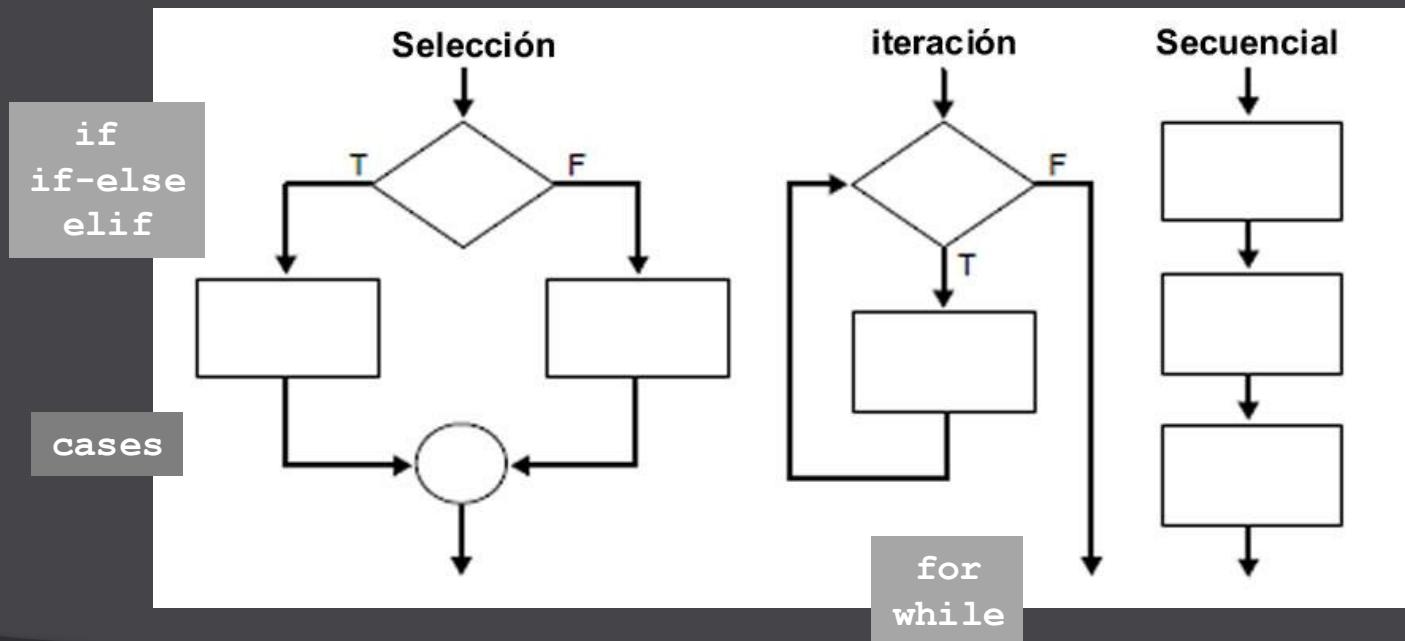
# Estructura de control

Las estructuras de control, denominadas también sentencias de control, permiten tomar decisiones y realizar un proceso **repetidas veces**. Se trata de estructuras muy importantes, ya que son las encargadas de controlar el flujo de un programa.



# Estructura de control

Las estructuras de control, denominadas también sentencias de control, permiten tomar decisiones y realizar un proceso **repetidas veces**. Se trata de estructuras muy importantes, ya que son las encargadas de controlar el flujo de un programa.



# Parsers

Parsear es un proceso que implica que un programa **analice una cadena de texto**, divida sus elementos y **extraiga información de utilidad** de la misma. Es una técnica muy utilizada en campos como la programación, aunque no se limita solo a esta área. Hablar de parsear implica convertir datos en un formato estructurado, que puede ser entendido y procesado por un sistema o programa.

```
with open('foto.jpg', 'r+') as f:
```

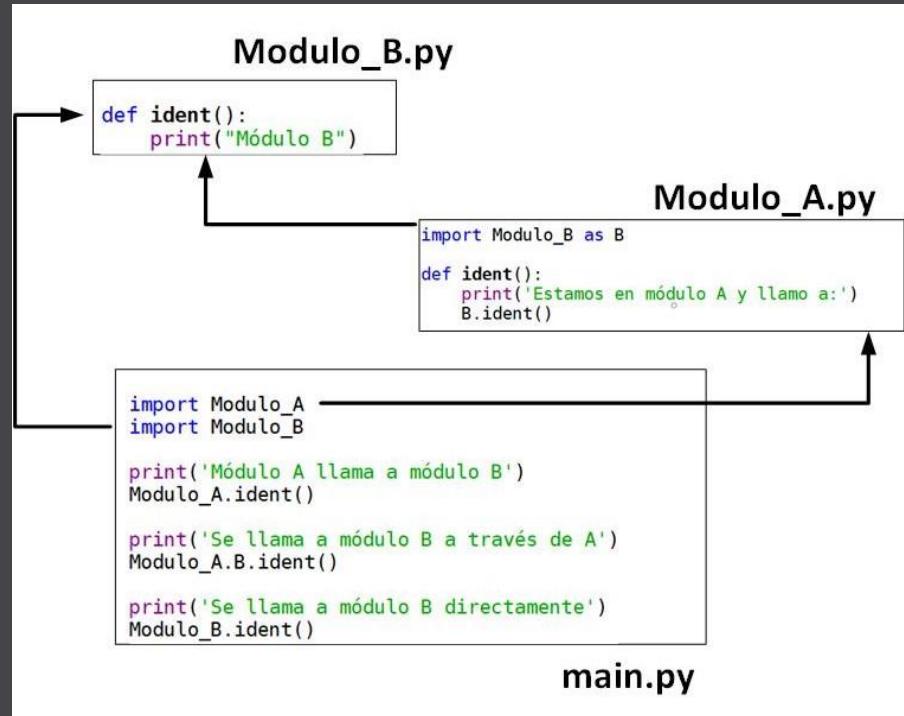
- **r** : Abre el fichero en modo lectura.
- **r+** : Si quieras leer y escribir en el fichero.
- **w** : Para sobreescribir el contenido.
- **a** :Para añadir al final del fichero en el caso de que ya exista.

# Almacenamiento e Importación de Modulos Python

A medida que agregas más funciones a tus clases, tus archivos pueden volverse largos.

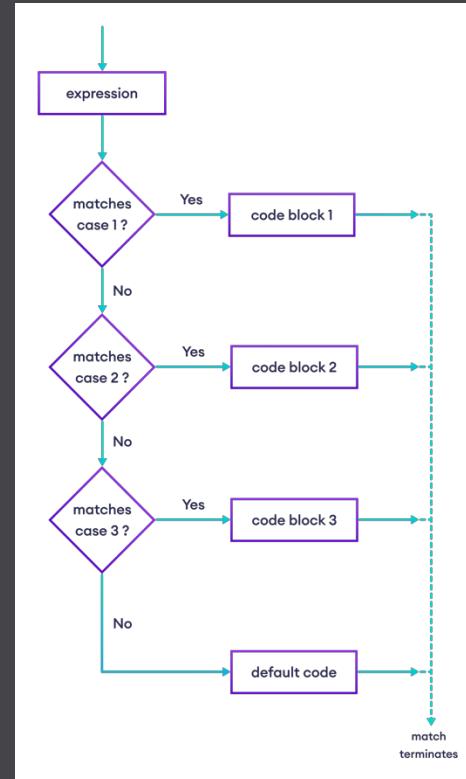
Python permite almacenar clases en módulos e importarlas.

Un **módulo** en Python es un archivo .py que contiene funciones, clases o variables reutilizables



# Tipos de métodos: La clase Match.

- Concordancia de patrones
- Proporciona forma concisa de probar expresiones
- Permite operaciones utilizando enfoques tradicionales
- Simplifica operaciones de largas sentencias if-else a bloque más compacto y legible



\*Match...case se introdujo  
en Python 3.10 y no  
funciona en versiones  
anteriores.

# Tipos de métodos: La clase Match.

```
match expression_to_match:  
    case pattern1:  
        # Code block for pattern1  
    case pattern2:  
        # Code block for pattern2  
    ...  
    case patternN:  
        # Code block for patternN  
    case _:  
        # Catch-all, default code block (optional)
```

# Tipos de métodos: La clase Match.

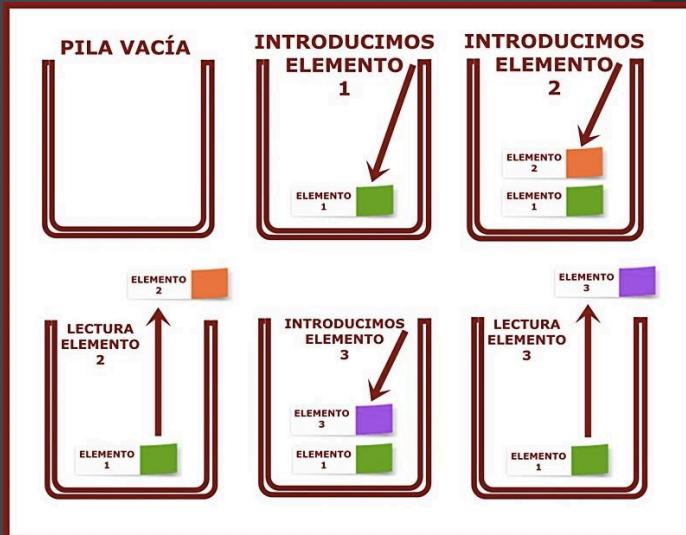
Una Expresión Regular o RegEx es una secuencia especial de caracteres que utiliza un patrón de búsqueda para encontrar una cadena o conjunto de cadenas.

Puede detectar la presencia o ausencia de un texto haciéndolo coincidir con un patrón particular y también puede dividir un patrón en uno o más sub-patrones.

Function	Description
re.findall()	finds and returns all matching occurrences in a list
re.compile()	Regular expressions are compiled into pattern objects
re.split()	Split string by the occurrences of a character or a pattern.
re.sub()	Replaces all occurrences of a character or patter with a replacement string.
re.escape()	Escapes special character
re.search()	Searches for first occurrence of character or pattern

# Pila de llamadas (call stack)

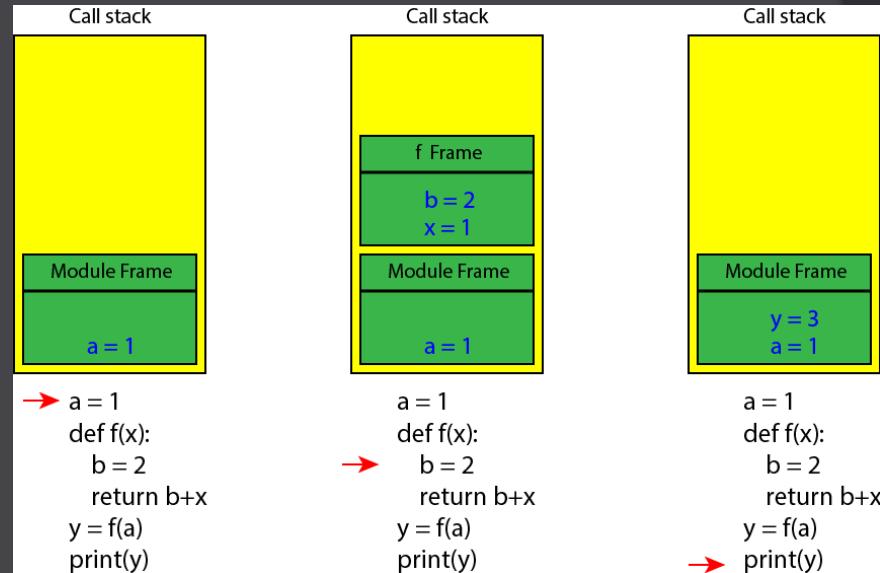
Mientras se ejecuta una función, Python necesita **almacenar** alguna **información** sobre las variables locales de esa función. Los valores de las **variables locales** se almacenan en un trozo de memoria llamado *frame*. Cuando la función regresa, esas variables locales ya no están en el ámbito, y pueden ser destruidas; el marco asociado con la función es destruido. Estos marcos se almacenan en una estructura llamada **pila de llamadas**. La función que se está ejecutando actualmente tiene un marco accesible, en la **parte superior** de la pila. Debajo en la pila está el marco de la función suspendida que llamó a la función que se está ejecutando actualmente.



# Pila de llamadas (call stack)

Es una estructura dinámica de datos LIFO(Last In, First Out ), que almacena la información sobre las subrutinas activas de un programa.

El mantenimiento de la pila de llamadas es importante para el correcto funcionamiento de la mayoría de programas, los detalles por lo general están ocultos y de forma automática en los lenguajes de alto nivel. Muchos conjuntos de instrucciones de computadora proveen instrucciones especiales para manipular pilas.



### Python 3.6

```
→ 1 def functionA():
 2     print("A called.")
 3     x = 5
 4     functionC()
 5     print(x)
 6
 7 def functionB():
 8     print("B called.")
 9     y = 10
10
11
12 def functionC():
13     print("C called.")
14     x = 2
15     k = 6
16     functionB()
17     print(x)
18
```

Print output (drag lower right corner to resize)

#### Frames

Global frame

functionA

functionB

functionC

functionA

#### Objects

function  
functionA()

function  
functionB()

function  
functionC()

### Python 3.6

```
1 def functionA():
2     print("A called.")
3     x = 5
4     functionC()
5     print(x)
6
7 def functionB():
8     print("B called.")
9     y = 10
10
11
12 def functionC():
13     print("C called.")
14     x = 2
15     k = 6
16     functionB()
17     print(x)
18
```

Print output (drag lower right corner to resize)

A called.

### Frames

Global frame

functionA

functionB

functionC

### Objects

function  
functionA()

function  
functionB()

function  
functionC()

functionA

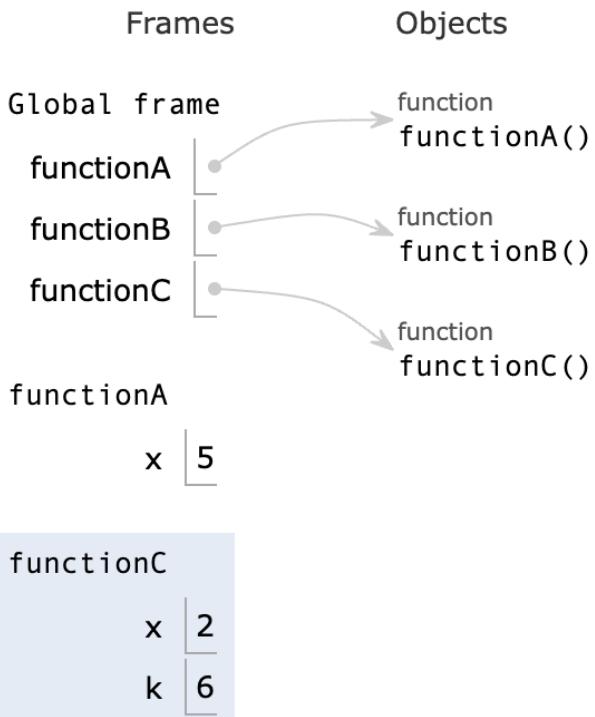
x | 5

### Python 3.6

```
1 def functionA():
2     print("A called.")
3     x = 5
4     functionC()
5     print(x)
6
7 def functionB():
8     print("B called.")
9     y = 10
10
11
12 def functionC():
13     print("C called.")
14     x = 2
15     k = 6
16     functionB()
17     print(x)
18
```

Print output (drag lower right corner to resize)

A called.  
C called.



## Python 3.6

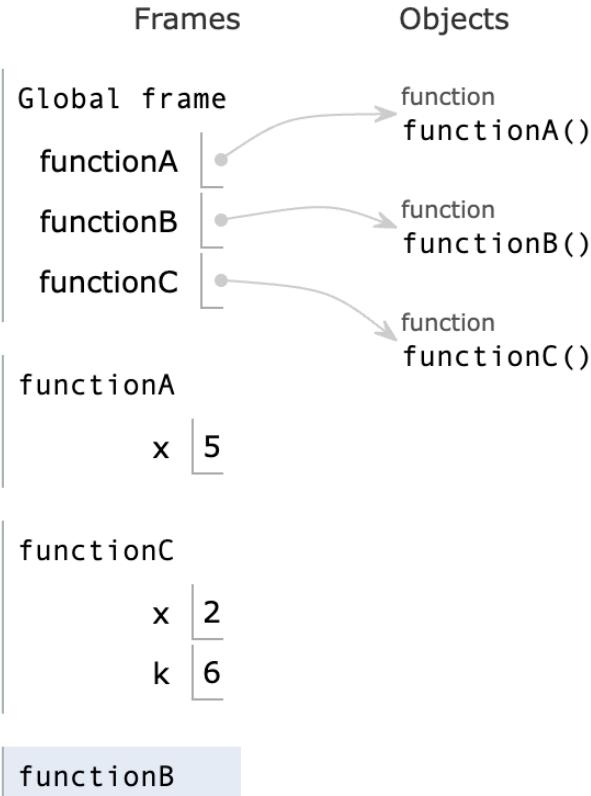
```
1 def functionA():
2     print("A called.")
3     x = 5
4     functionC()
5     print(x)
6
7 def functionB():
8     print("B called.")
9     y = 10
10
11
12 def functionC():
13     print("C called.")
14     x = 2
15     k = 6
16     functionB()
17     print(x)
18
```

[Edit Code & Get AI Help](#)

line that just executed  
next line to execute

Print output (drag lower right corner to resize)

A called.  
C called.



### Python 3.6

```
1 def functionA():
2     print("A called.")
3     x = 5
4     functionC()
5     print(x)
6
7 def functionB():
8     print("B called.")
9     y = 10
10
11
12 def functionC():
13     print("C called.")
14     x = 2
15     k = 6
16     functionB()
17     print(x)
18
```

Print output (drag lower right corner to resize)

```
A called.  
C called.
```

### Frames

Global frame

functionA

functionB

functionC

### Objects

function  
functionA()

function  
functionB()

function  
functionC()

functionA

x 5

functionC

x 2

k 6

### Python 3.6

```
1 def functionA():
2     print("A called.")
3     x = 5
4     functionC()
5     print(x)
6
7 def functionB():
8     print("B called.")
9     y = 10
10
11
12 def functionC():
13     print("C called.")
14     x = 2
15     k = 6
16     functionB()
17     print(x)
18
```

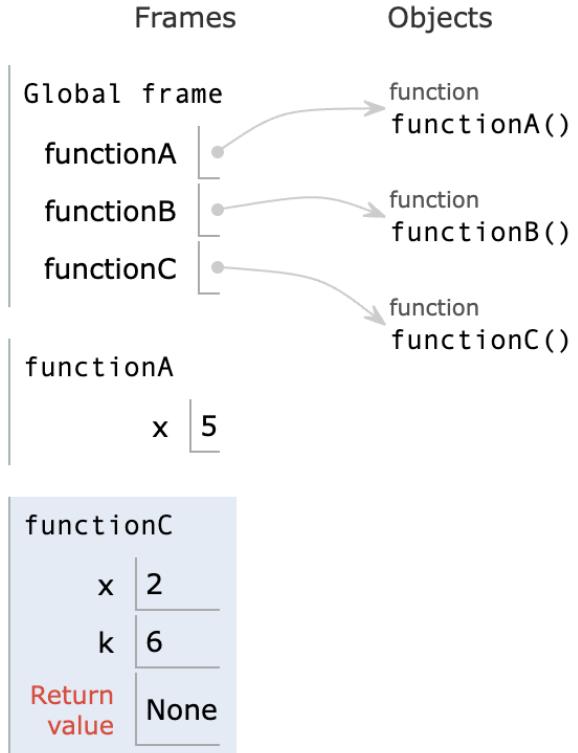
[Edit Code & Get AI Help](#)

→ line that just executed

→ next line to execute

### Print output (drag lower right corner to resize)

```
C called.  
B called.  
2
```



## Python 3.6

```
1 def functionA():
2     print("A called.")
3     x = 5
4     functionC()
5     print(x)
6
7 def functionB():
8     print("B called.")
9     y = 10
10
11
12 def functionC():
13     print("C called.")
14     x = 2
15     k = 6
16     functionB()
17     print(x)
18
```

[Edit Code & Get AI Help](#)

→ line that just executed

→ next line to execute

Print output (drag lower right corner to resize)

```
C called.  
B called.  
2
```

### Frames

Global frame

functionA

functionB

functionC

### Objects

function

functionA()

function

functionB()

function

functionC()

functionA

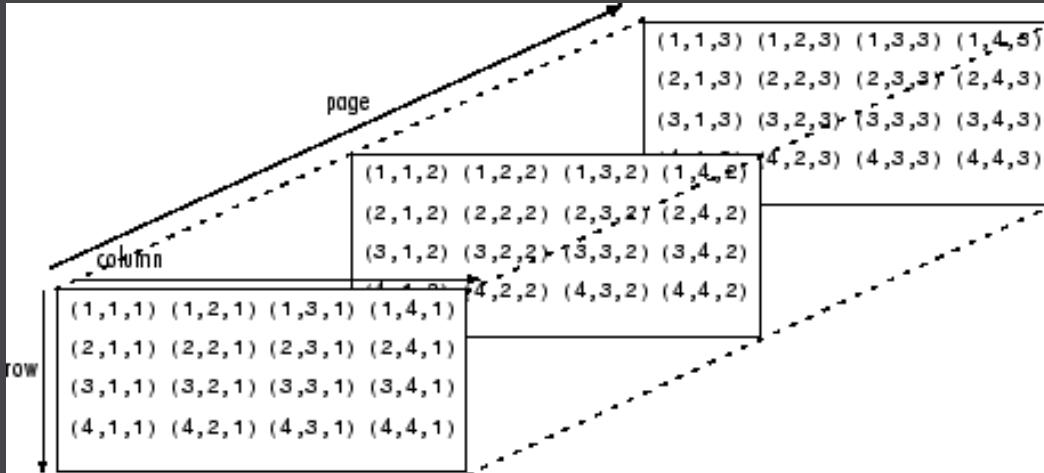
x | 5



# Sobrecarga de funciones

En algunos lenguajes de programación, como Java o C++, es posible definir múltiples funciones con un mismo nombre, en tanto en cuanto las definiciones difieran en la cantidad de argumentos o en el tipo de dato de alguno de ellos. A esta práctica se la denomina *sobrecarga de funciones* (o de métodos, en caso de tratarse de una clase).

# Arreglos multidimensionales



# Arreglos multidimensionales

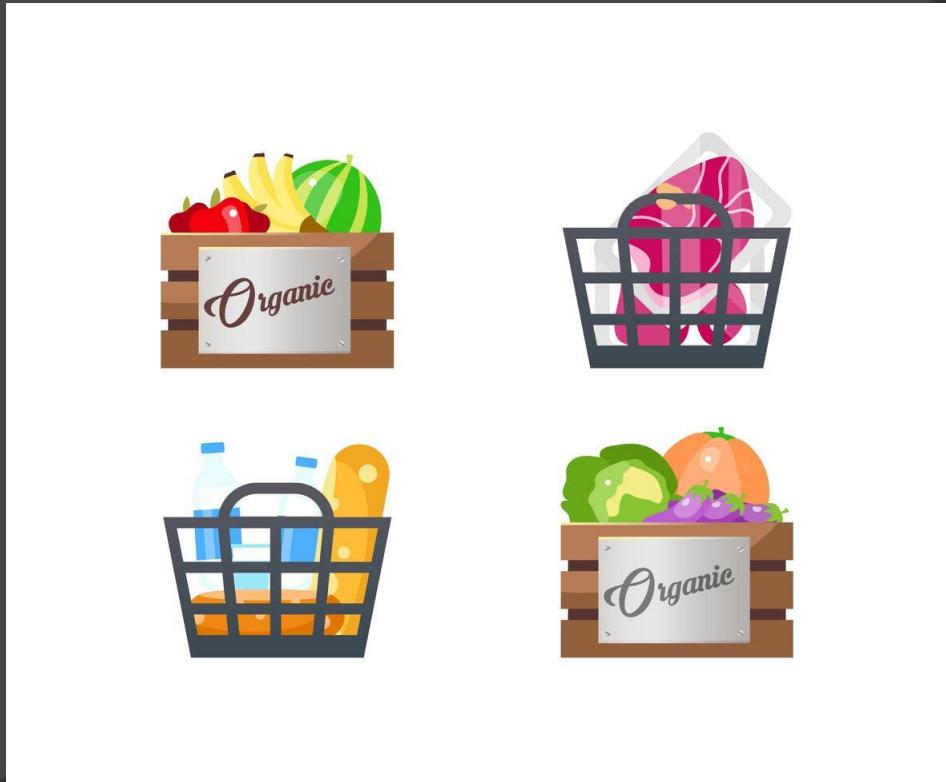
Columnas

Filas

País	Ciudad 1	Ciudad 2	Ciudad 3
Colombia	Bogotá	Cali	Barranquilla
Argentina	Buenos Aires	Córdoba	La Plata
Mexico	Ciudad Juárez	Monterrey	Guadalajara

# Clases anidadas

- Una clase anidada es una clase que se define dentro de otra clase.
- Las clases anidadas permiten organizar el código, encapsular la funcionalidad y mantener una estructura clara.
- Permiten acceder a los atributos y métodos de la clase que las contien



# Arreglos multidimensionales

## Ejercicio: Consulta de información sobre países y ciudades

Usando una fuente de datos con los siguientes elementos, crea un programa que modele la relación entre países y ciudades.

### Cada país debe tener como atributos:

- Comida típica
- Población total
- Producto Interno Bruto (PIB)
- Superficie (en km<sup>2</sup>)

### Cada ciudad debe tener como atributos:

- Densidad poblacional
- Gentilicio
- Altitud media (en metros)
- Principal actividad económica

### Requisitos del programa:

#### • Permitir al usuario consultar por nombre de país o ciudad.

- Si se consulta un **país**, se debe mostrar toda su información y además listar las ciudades que pertenecen a él (disponibles en la base de datos).

The diagram illustrates a 2D array structure. It features a grid of four columns and three rows. The first column is labeled 'País', the second 'Ciudad 1', the third 'Ciudad 2', and the fourth 'Ciudad 3'. The rows are labeled 'Colombia', 'Bogotá', 'Cali', 'Barranquilla' in the first row; 'Argentina', 'Buenos Aires', 'Córdoba', 'La Plata' in the second row; and 'Mexico', 'Ciudad Juárez', 'Monterrey', 'Guadalajara' in the third row. A green bracket on the left side of the grid is labeled 'Filas' (Rows), and a green bracket at the top of the grid is labeled 'Columnas' (Columns).

País	Ciudad 1	Ciudad 2	Ciudad 3
Colombia	Bogotá	Cali	Barranquilla
Argentina	Buenos Aires	Córdoba	La Plata
Mexico	Ciudad Juárez	Monterrey	Guadalajara