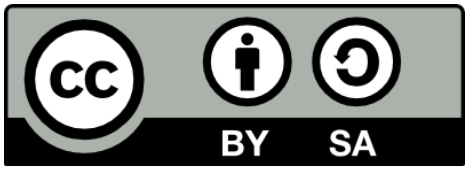
Informática 2

- Presentaciones -

Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

Curso 2024-2025

Universidad Rey Juan Carlos



Índice de contenidos

Índice de contenidos

Informática 2

Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

Bloque I.
Repaso a la programación

Universidad Rey Juan Carlos

Tema 1: Introducción a la programación

Bloque I. Repaso de programación

¿Qué es un programa?

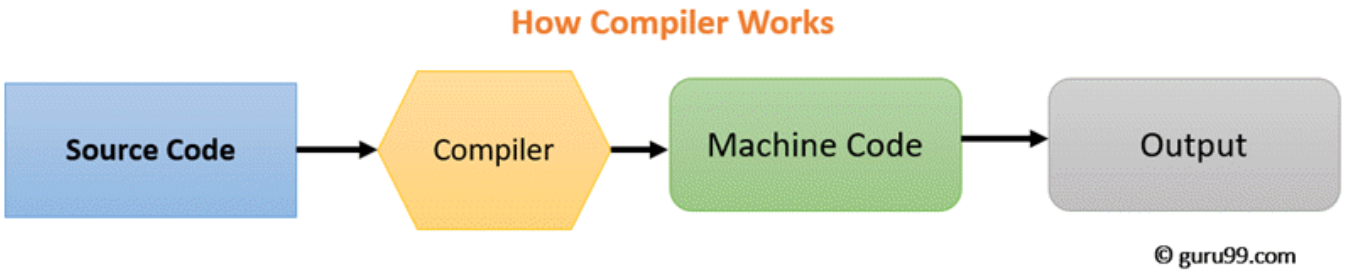
Un **programa de ordenador** es una colección de instrucciones que realiza una tarea específica cuando es ejecutada por un ordenador

```
# Python 3: Simple output (with Unicode)
>>> print("Hello, I'm Python!")
Hello, I'm Python!

# Input, assignment
>>> name = input('What is your name?\n')
>>> print('Hi, %s.' % name)
What is your name?
Python
Hi, Python.
```

¿Qué es Python?

Python es un lenguaje de programación de alto nivel y su filosofía de diseño se basa en la legibilidad del código y en una sintaxis que permita a los programadores expresar conceptos en unas pocas líneas de código.

Lenguaje interpretado vs. compilado



Ventajas de Python

- **Legible**: Sintaxis intuitiva y estricta
- **Productivo**: ahorra mucho código.
- **Portátil**: Para todos los sistemas operativos.
- **Completo**: Viene con muchas librerías

Para qué se usa Python

Desarrollo web

- Frameworks como *Django, Flask*

Análisis de datos

- Librerías como NumPy y Pandas
- Librerías de visualización de datos como Matplotlib y Seaborn

Internet-of-things

- Raspberry Pi + Python

Web scraping

- Scrapy

Visión por computador

- Librería OpenCV

Machine Learning

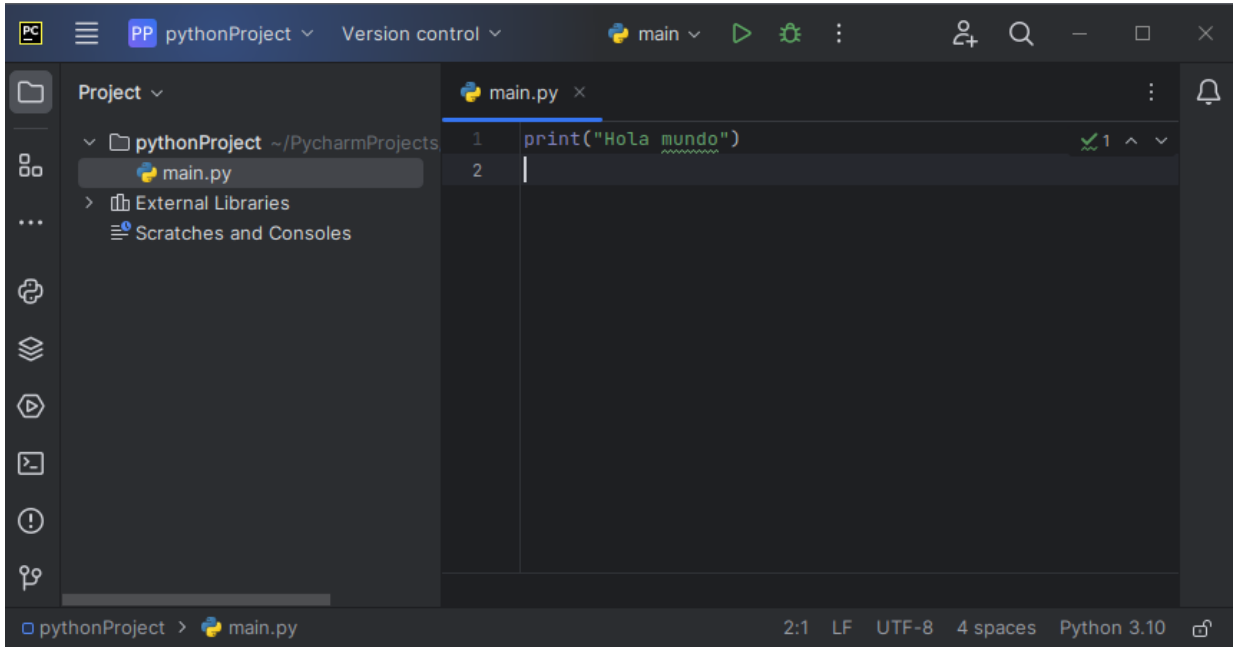
- Librerías como Scikit-learn, Pytorch

Desarrollo de juegos

- PyGame

Entorno de programación

- PyCharm



- Terminal: lanzar programas con `python3`

Intérprete de Python

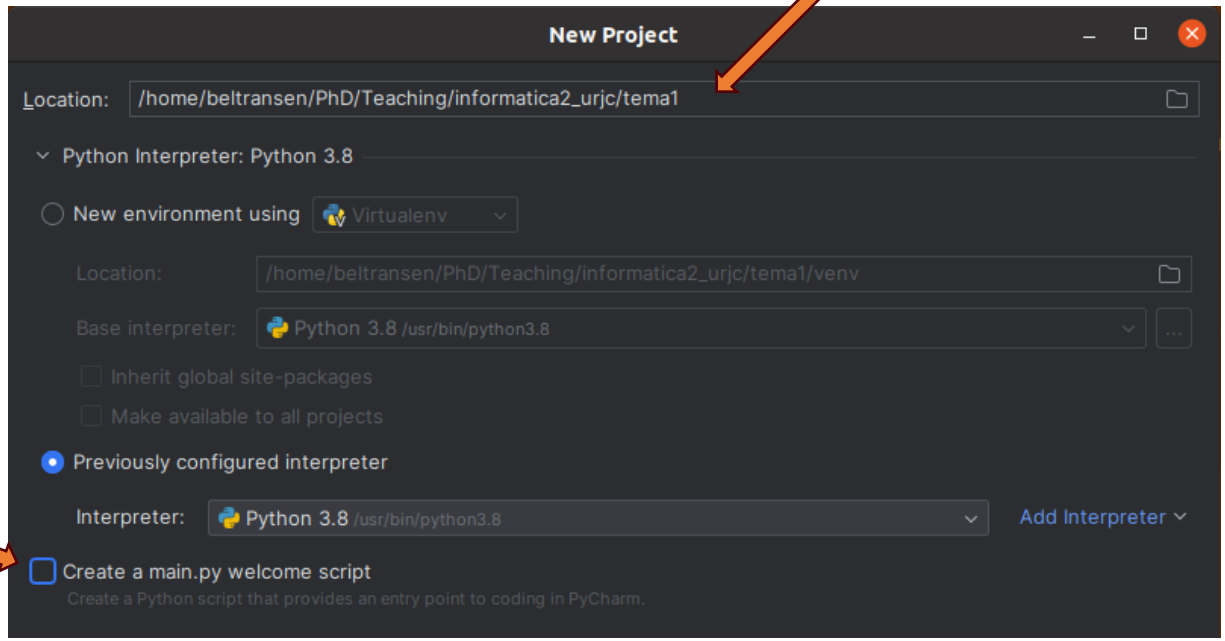
- Se ejecuta en la terminal mediante el comando ***python3***
- Es una consola interactiva, donde poner instrucciones y ver el resultado al instante
- Se suele emplear para pequeñas pruebas
- Ctrl+D para salir


```
python3
[redacted] ~ python3
Python 3.8.10 (default, May 26 2023, 14:05:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hola mundo')
Hola mundo
>>> print('Bienvenidos a la asignatura')
Bienvenidos a la asignatura
>>>
```

Trabajando en PyCharm

- Crear un nuevo proyecto por sesión
- Programar cada ejercicio en un fichero Python distinto
- Criterio de nombres:
  - ejercicioX.py (ejercicio1.py, ejercicio 2.py)

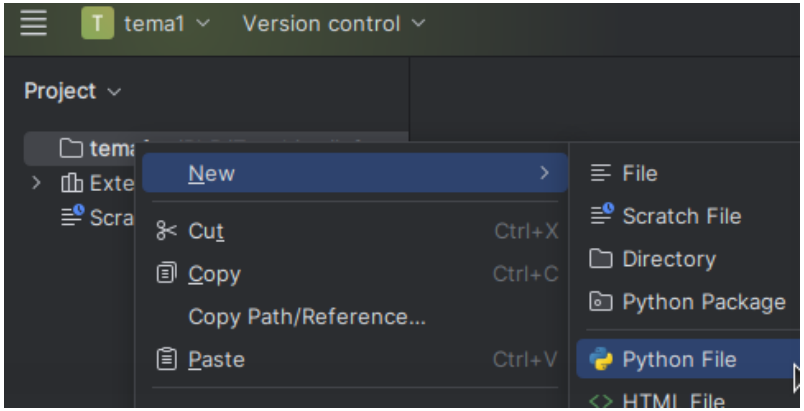
Ruta y nombre del proyecto

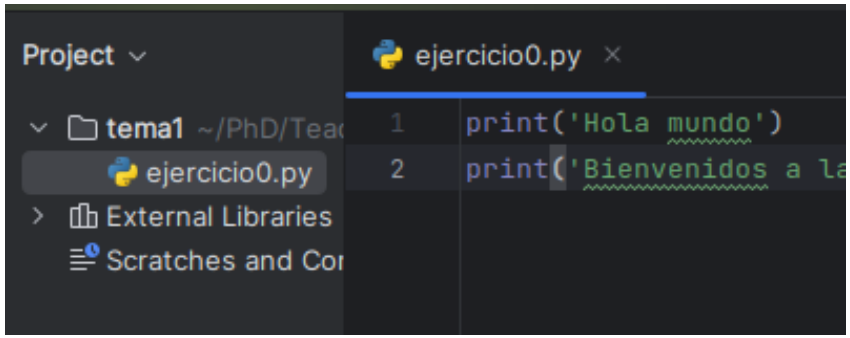


Desmarcar para no crear un archivo por defecto

Creando Python scripts

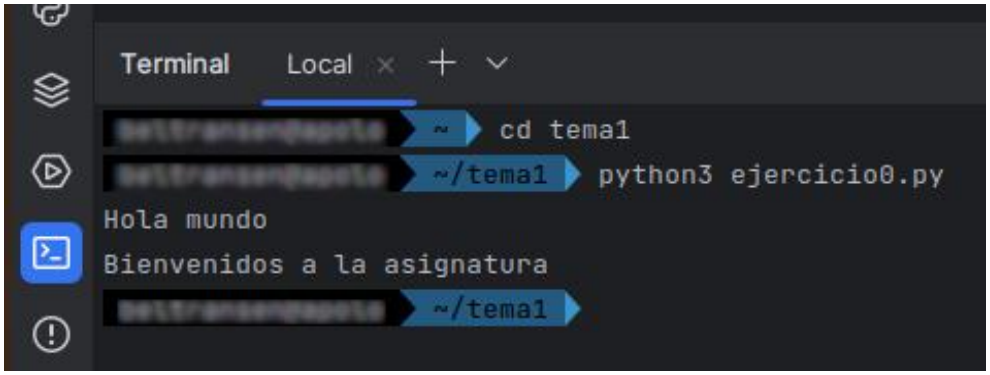
1. Añadir fichero en el proyecto





2. Crear nuestro programa

3. Ejecutar en la terminal



Tema 2:
Python básico I

Bloque I. Repaso de programación

Tipos de datos

- `Integer <int>`
- `Float <float>`
- `String <str>`
- `Boolean <bool>`
- `Etc.`

El comando `type` muestra el tipo.

```
>>> type(5)
<type 'int'>

>>> type(4.6)
<type 'float'>

>>> type('cadena')
<type 'str'>

>>> type(''cadena'')
<type 'str'>

>>> type(True)
<type bool'>
```

Variables

Las variables se emplean para guardar información que se referencia y manipula en un programa

```
a = 5.0
b = 'Hola'
c = 5+6
```

- Elegir nombres que tengan sentido
- Si queremos que tengan dos palabras, usar _
- *None*: variable nula

Nombres

- No pueden usarse palabras reservadas como nombres de variables, funciones, etc.

`False`	`await`	`else`	`import`	`pass`
`None`	`break`	`except`	`in`	`raise`
`True`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

- Se distinguen mayúsculas y minúsculas
- Tienen que empezar con una letra (o un símbolo)
- No se pueden usar símbolos especiales: !, @, #, $,…

Precedencia de operadores

`()`
`**`
`+x`, `-x`, `~x`
`*`, `/`, `//`, `%`
`+`, `-`
`<<`, `>>`
`&`
`^`
`|`
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`
`not`
`and`
`or`

Operadores

Operadores aritméticos

- + suma
- - resta
- / división
- * multiplicación
- % resto
- ** potencia
- // división de parte entera

Operadores lógicos

- and
- or
- not

Operadores de comparación

- < menor que
- > mayor que
- <= menor o igual que
- >= mayor o igual que
- == igual que
- != distinto que

Operadores de asignación

- =
- +=, -=, *=, etc.

Extra: is, is not (comprueban si dos valores corresponden a la misma ubicación en memoria)

Comentarios

Comentarios de una sola línea

```
# Esto es un comentario
```

Comentarios que abarcan múltiples líneas

```
"""
a = input('Inserte un número: ')
a
print(a)
print('Hola')
"""

print('Esto no está comentado')
```

- Usar los comentarios para explicar por qué se hace algo y no cómo se hace (eso ya se puede ver en el código)
- Pueden servir para depurar (comentando línea a línea)

Funciones de entrada/salida

Los operadores de entrada/salida (I/O) se usan para tomar salidas y mostrar salidas

```
>>> a = input('Inserte un número: ')
Inserte un número: 3
>>> a
3
>>> print(a)
3
>>> print('Hola', a)
Hola 3
```

La cadena de salida se puede formatear para hacerla bonita

```
x = 5
y = 10

print('El valor de x es {} y el de y, {}'.format(x,y))
```

Conversión de tipos

- Automática en operaciones con tipos compatibles

```
integer_number = 123
float_number = 1.23

new_number = integer_number + float_number

# Muestra el nuevo valor y el tipo de dato resultante
print("Value:",new_number)
print("Data Type:",type(new_number))
```

- Puede ser explícita: funciones int(), float(), str()

```
num_string = '12'
num_integer = 23

# Conversión de tipos explícita
num_string = int(num_string)
```

Ejercicio 1

Escribir un script que:

- Le pida dos números al usuario
- Sume los dos números y guarde el resultado en una variable
- Imprima el resultado

Nota: Escribe el script en PyCharm, guárdalo, abre la terminal y ejecútalo con el comando ‘python3’ en la terminal

Concatenación de cadenas

El operador + se usa para unir dos cadenas

```
>>> print('Estoy concatenando ' + 'dos cadenas')
Estoy concatenando dos cadenas
```

Ejercicio 2

Crear un algoritmo para calcular el perímetro y el área de un círculo:

1. Crear una variable con el valor de π
2. Pedir el valor del radio al usuario
3. Calcular el perímetro y almacenarlo en una variable
4. Calcular el área y almacenarlo en una variable
5. Mostrar en pantalla el valor de las variables de perímetro y área

Ejercicio 3

Crea un algoritmo para convertir de grados Fahrenheit a Celsius:

1. Pide los grados Fahrenheit al usuario
2. Haz la conversión
3. Muestra el resultado por pantalla

Nota: Para convertir Fahrenheit a Celsius hay que restar 32 y dividir por 1,8

If/else

La sentencia **if/else** ejecuta un bloque de código si se cumple una **condición especificada**. Si la condición no se cumple, puede ejecutarse otro bloque de código.

```
var = 10

if var >= 5:
    print('var mayor o igual que 5')
elif var < 0:
    print('var es negativo')
elif var == 0:
    print('var es cero')
else:
    print('var es menor que 5')
```

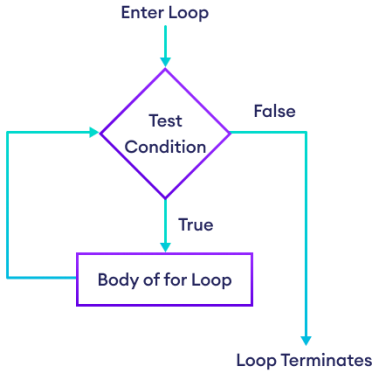
If/else

La indentación determina la jerarquía

```
var = 10

if var >= 5:
    if var >= 7:
        print('var mayor o igual que 7')
    else:
        print('var menor que 7')
    print ('var mayor o igual que 5')
```

Bucles for

Recorre un bloque de código varias veces



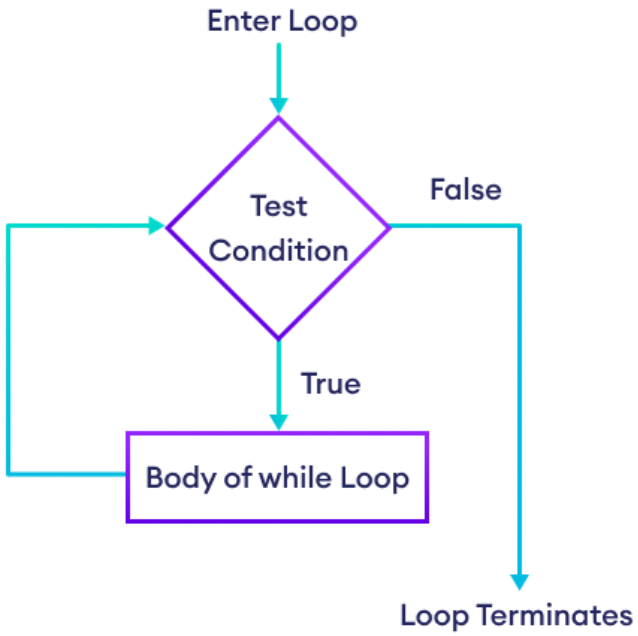

```
for i in range(10):
    print(i)
```

- La función `range(fin)` devuelve una secuencia de números entre 0 y fin-1
- Acepta más parámetros: `range(inicio, fin, paso)`
  - `range(2, 10, 3)` devuelve `2, 5, 8`

```
numbers = range(2, 10, 3)
```

Bucles while

Recorre en bucle un bloque de código mientras se cumple una condición especificada




```
var = 10

while(var <= 20):
   print('var es menor o igual a 20')
   var+=2
```

Funciones

Una **función** es un bloque de código diseñado para realizar una tarea concreta.

```
def suma_dos_numeros(a, b):
    suma = a + b
    return suma

suma_dos_numeros(4, 4)
```

Nota: en los ejercicios vamos a usar esta función, consérvala

- Cuidado: las modificaciones dentro de una función de ciertos tipos de datos (mutables) se reflejan fuera.

Ejercicio 4

Usando la función ***suma_dos_numeros,*** crea una función ***suma_tres_numeros*** que sume tres números de entrada y devuelva el resultado

```
# Prueba
print(suma_tres_numeros(2, 4, 12))
18
```

Ejercicio 5: Fizzbuzz

1. Escribe un programa que imprima los números del 1 al 100. Pero para los **múltiplos de tres imprime "Fizz"** en lugar del número y para los **múltiplos de cinco imprime "Buzz"**. Para los números **que son múltiplos de tres y de cinco imprime "FizzBuzz".**
2. Crea una función para comprobar si un número es múltiplo de otro y úsala en el programa anterior.

Ejercicio 6

Crear una función que determine si un número entero positivo es un número primo o no.

Nota: Recuerda que un número primo es un número natural mayor que 1 que no tiene divisores positivos distintos de 1 y él mismo.

```
>> is_prime(5)
'Primo'

>> is_prime(4)
'No es primo'

>> is_prime(7)
'Primo'

>> is_prime(10)
'No es primo'

>> is_prime(147)
'No es primo'

>> is_prime(149)
'Primo'
```

Tema 3:
Python básico II

Bloque I. Repaso de programación

Tuplas

Una tupla es una lista ordenada (**inmutable**) de elementos.

```
# Crea una tupla
t = (5, 6)

# Crea una tupla
tuplex = ( 'tuple', False, 3.2, 1)
```

- Admite distintos tipos de elementos y duplicados
- Se puede crear sin paréntesis (pero no es recomendable)
- Se puede crear con un solo elemento:

```
var2 = ("hello",)
```

Acceder a elementos de la tupla

1. Indexado

```
letters = ("p", "r", "o", "g", "r", "a", "m")

print(letters[0])   # imprime "p"
```

2. Indexado negativo

```
print(letters[-1])   # imprime " m"
print(letters[-3])   # imprime “r"
```

3. Rebanado (slicing)

```
print(letters[1:4])  #  imprime ('r', 'o', 'g’)
print(letters[:2]) #  imprime (‘p', ‘r')
print(letters[:-5]) # imprime ('p', 'r')
print(letters[5:]) # imprime ('a', 'm')
```

Palabra clave **in**

Iterar sobre una tupla

`language` es tipo ‘str’

```
languages = ('Python', 'Swift', 'C++')

# iterating through the tuple
for language in languages:
    print(language)
```

(Nota: También se puede iterar sobre cadenas)

```
for i in 'hola':
    print(i)
```

Comprobar si un elemento existe

```
print('C' in languages)    # False
print('Python' in languages)    # True
print('C' not in languages)    # True
```

Métodos de las tuplas

Están disponibles los dos métodos siguientes.

```
my_tuple = ('a','p','p','l','e',)

# Count
# Salida: 2
print(my_tuple.count('p'))

# Index
# Salida: 3
print(my_tuple.index('l'))
```

- `Count` cuenta las veces que se repite un elemento
- `Index` muestra el índice de un elemento en la tupla

Ejercicio 7

Crea una tupla con los nombres de los días de la semana, recórrelos e imprímelos letra a letra. Después, muestra el índice de cada día.

```
# Caso de prueba
l
u
n
e
s
Índice  0
m
a
r
t
e
s
Índice  1
```

Listas (lists)

Una lista es el equivalente en Python de un array, pero es **redimensionable** y **puede contener elementos de distintos tipos**

```
xs = [3, 1, 'manzana']    # Crea una lista
```

- Similar a la tupla, pero modificable
- Se pueden crear listas vacías

```
lista_vacia = []
```

Rebanado de listas (list slicing)

Al igual que en el caso de las tuplas, Python proporciona una sintaxis concisa para acceder a sublistas dentro de listas

```
xs = [3, 1, 3, 4, 'manzana']

print(xs)     # Imprime "[3, 1, 3, 4, 'manzana']"
print(xs[2:4])

# Obtiene una porción del índice 2 al 4 (no incluido);
# imprime "[3, 4]"
```

Iterar sobre listas

```
animals = ['cat', 'dog', 'monkey']

"""
Bucles: Puedes hacer un bucle sobre los
elementos de una lista de la siguiente manera
"""
for animal in animals:
    print(animal)

"""
Si desea acceder al índice de cada elemento
dentro del cuerpo de un bucle, utiliza la función enumerar
"""
for idx, animal in enumerate(animals):
    print(str(idx)+') ' +animal)

"""
Mala alternativa cogida de otros lenguajes,
mejor usar enumerar
"""
for i in range(0,len(animals))
    print(animal[i])
```

Añadir elementos

1. append(): añade un elemento al final

```
numbers = [21, 34, 54, 12]
numbers.append(32)
```

2. extend(): mete todos los elementos de un iterable al final

```
numbers = [1, 3, 5]

even_numbers = [4, 6, 8]

# añadir elementos de even_numbers a la lista de números
numbers.extend(even_numbers))
```

3. insert(): añade un elemento en un índice específico

```
numbers.insert(1, 20)
```

Quitar elementos

1. del: elimina el elemento indicado

```
languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']

# borra el segundo elemento
del languages[1]
print(languages) # ['Python', 'C++', 'C', 'Java', 'Rust', 'R']
```

2. remove(): elimina elementos con un valor concreto

```
languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']

# borra 'Python' de la lista
languages.remove('Python')

print(languages) # ['Swift', 'C++', 'C', 'Java', 'Rust', 'R']
```

Tuplas vs. listas

- Las tuplas son **inmutables** y suelen contener una secuencia heterogénea de elementos.
  - "más rápidas" que las listas
  - "protegen" los datos
- Las listas son **mutables**, y sus elementos *suelen ser homogéneos*
- **No se pueden** añadir elementos a una tupla. Las tuplas no tienen métodos append o extend.
- **No se pueden** eliminar elementos de una tupla. Las tuplas no tienen métodos remove o pop.
- **Se puede** saber si un elemento existe en una tupla y en una lista.

Ejercicio 8

- Crea una función que reciba dos listas de números, las concatene y devuelva un lista con los *N//2* números centrales en orden inverso, siendo N el número de elementos de las dos listas concatenadas.

```
# Casos de prueba
>>> imprimir_numeros_centrales_inversos([1, 2, 3], [4, 5, 6, 7])
[5, 4, 3]
>>> imprimir_numeros_centrales_inversos([9, 8, 7, 6], [5, 4, 3, 2])
[4, 5, 6, 7]
```

Ejercicio 9

- Escribe la función **select_unique_values** para obtener valores únicos de una lista.
- **Nota:** el resultado debe estar en una lista.

```
# Casos de prueba
>>> select_unique_values([2,2,2,2,3,3,1,2,6,7,8,9,9,10])
[1, 2, 3, 6, 7, 8, 9, 10]

>>>
select_unique_values(['one','two','one','two','three','one'])
['three', 'two', 'one']
```

Conjuntos (sets)

Un conjunto es una colección **desordenada sin elementos duplicados.**

```
animals = {'cat', 'dog', 'cat', 'dog', 'turtle'}
print(animals)
# Salida: {'dog', 'turtle', 'cat'}
```

Métodos

- add(): añade elementos
- update(): añade todos los elementos de otro iterable
- discard(): elimina el elemento especificado

Operaciones: unión |, intersección &, diferencia -

Diccionarios (dicts)

Un diccionario **almacena pares (clave [key], valor [value])**

```
# Crea un diccionario con datos
d = {'gato': 'miau', 'perro': 'guau'}
```

- Acceso a los elementos:

```
print(d['gato'])
```

- Para añadir nuevos elementos, simplemente se asigna un valor a una nueva clave:

```
d['vaca'] = 'mu'
```

- Eliminar elementos:

```
del d['perro']
```

Conversiones

Se puede convertir entre tipos usando las funciones `list()`, `tuple()`, `set()`

```
# Dict a list
d = {'a': 'Jose', 'b': 'Miguel'}
d_keys = list(d.keys())          #Devuelve la lista de claves del diccionario
d_values = list(d.values())      #Devuelve la lista de valores del diccionario

# Tuple a list
words = ('Blah', 'Hello')
words_list = list(words)

# List a set
fruit = ['banana', 'orange', 'pineapple']
fruit_set = set(fruit)
```

Ejercicio 10

- Escribe de nuevo la función **select_unique_values** para obtener valores únicos de una lista, esta vez utilizando únicamente conversiones entre tipos *list*, *tuple*, *set*, o *dict* (sin recorrerla)
- **Nota:** el resultado debe estar en una lista.

```
# Casos de prueba
>>> select_unique_values([2,2,2,2,3,3,1,2,6,7,8,9,9,10])
[1, 2, 3, 6, 7, 8, 9, 10]

>>>
select_unique_values(['one','two','one','two','three','one'])
['three', 'two', 'one']
```

Ejercicio 11

- Escribe la función **suma_valores** para sumar todos los elementos de un diccionario.

```
# Casos de prueba
d = {'a': 3, 'b': 4,'c':5}

>>> sum_values(d)
12
```

Método items()

```
>>> likes = {"color": "blue", "fruit": "apple", "pet": "dog"}
>>> for item in likes.items():
...     print(item)
...     print(type(item))
...
('color', 'blue')
<class 'tuple'>
('fruit', 'apple')
<class 'tuple'>
('pet', 'dog')
<class 'tuple'>
```

Ejercicio 12

- Escribe la función **get_max_min** para obtener los valores máximo y mínimo de un diccionario y **devolver el resultado en forma de tupla.**
- Escriba la función **get_max_min_dict** para obtener los valores máximo y mínimo de un diccionario y **devolver el resultado en forma de diccionario**.

```
>>> d = {'a': 3, 'b': 4,'c':50,'d':1}
>>> get_max_min(d)
(50, 1)
>>> get_max_min_dict(d)
{'max': 50, 'min': 1}
```

Pista: usar **max()** y **min()**

Manejo de errores (try/except)

- Si **no se detecta** el error, el programa **se detiene**.
- Hay ocasiones en las que queremos evitar que el programa se detenga, ya sea porque sabemos cómo solucionar el error o porque sabemos qué hacer en tal caso.

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")
```

Ejercicio 13

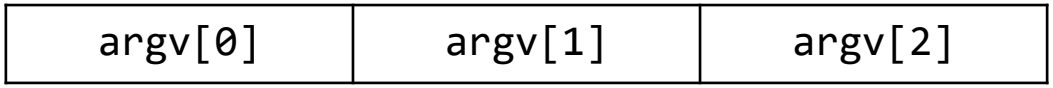
Crea una función que divide dos números enteros

- La función debe manejar el error de división por cero (ZeroDivisionError) y los tipos de operandos no soportados (TypeError)

```
#Test cases

>>> divide(8, 4)
>>> 2

>>> divide(8, 4.0)
>>> 2.0

>>> divide(8, "J")
>>> 'Input not a number, try again..'

>>> divide(8, 0)
>>> 'Division by zero, try again..'
```

Argumentos por línea de comandos

- sys.argv: lista con los comandos que se pasan al programa
- `python3 programa.py argumento1 argumento 2`




```
import sys

add = 0.0

# Obteniendo la longitud de los
# argumentos por linea de comandos
n = len(sys.argv)

for i in range(1, n):
    add += float(sys.argv[i])

print ("the sum is :", add)
```

Ejercicio 14

Crea un programa al que se le pasen parámetros por línea de comandos. El programa debe mostrar por pantalla los que sean números enteros, y mostrar la palabra “NaN” por cada argumento que no sea un número

Leyendo ficheros

```
# Leer texto
f = open('../Dropbox/helloworld.txt','r')
message = f.read()
print(message)
f.close()

# Leer líneas
f = open('../Dropbox/helloworld.txt','r')
message = f.readlines() #List
print(message)
f.close()

# Sin cerrar
with open('../Dropbox/helloworld.txt','r') as f:
    message = f.readlines() #List
```

Escribiendo ficheros

```
# Ejemplo con write
file = open('../Dropbox/testfile.txt','w')
file.write('Linea 1\n')
file.write('Linea 2\n')
file.close()

# reescribir
file = open('../Dropbox/testfile.txt','w')
file.write('Linea 3\n')
file.close()

# append
file = open('../Dropbox/testfile.txt','a')
file.write('Linea 4\n')
file.close()

# with
with open('../Dropbox/testfile.txt','a') as f:
    f.write('Linea 5\n')
```

Ejercicio 15

1. Crea un script que lea el fichero **input.txt**. El fichero contiene un número en cada línea.
2. Suma todos los números y escribe el resultado en **output.txt**
3. Escribe líneas con caracteres diferentes a números en un archivo **errors.txt**

output.txt

```
Total sum = 23369
```

errors.txt

```
'AAA'
'BBB'
'JDG474'
'TFF941'
'281'
'ZZZZ'
```

Informática 2

Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

Bloque II.
Programación orientada a objetos

Photo by Chris Ried on Unsplash

Universidad Rey Juan Carlos

Tema 4:
Clases

Bloque II. Programación orientada a objetos

Universidad
Rey Juan Carlos

Objetos

- Python es un lenguaje de programación **orientado a objetos**
- Un **objeto** es una combinación de variables (también llamadas atributos, variables de instancia o variables de objeto) y comportamientos (es decir, funciones, que son referidas como métodos en el contexto del objeto)
- Para crear un objeto, necesitas crear una **clase** usando la palabra clave `class` como se muestra a continuación:

```
class Estudiante:
    nombre = "Pedro"
```

Esto es un atributo; se pueden tener tantos como hagan falta

Creando objetos a partir de clases

Después de definir una clase, se puede crear un número cualquiera de objetos a partir de ella. Cada objeto de una clase se llama **instancia**.

```
s1 = Estudiante()
print(s1.nombre)
s2 = Estudiante()
print(s2.nombre)
s3 = Estudiante()
print(s3.nombre)
```

Pero todos los estudiantes tienen el mismo nombre…

- ¿Cómo podemos tener estudiantes con distintos nombres?

Ejercicio 1

Vamos a diseñar paso a paso un sistema simple para un comercio electrónico que maneja productos, clientes y órdenes.

- Crea una clase que represente a un producto y contenga su nombre y su precio.
- Instancia un objeto de esa clase e imprime su nombre.

Constructor de una clase

Todas las clases en Python tienen una función llamada `__init__()`, que se ejecuta siempre cuando la clase se instancia (se crea un objeto a partir de ella)

Se puede usar la función `__init__()` para asignar valores a atributos de los objetos – de hecho, se puede añadir cualquier código en la función `__init__()` que resulte necesario para crear objetos a partir de las clases

```
class Estudiante:
    def __init__(self, n):
        self.nombre = n
```

La función __init__() se llama **constructor**

Constructor de una clase

Todas las clases en Python tienen una función llamada `__init__()`, que se ejecuta siempre cuando la clase se inicializa (se crea un objeto a partir de ella)

Se puede usar la función `__init__()` para asignar valores a atributos de los objetos – de hecho, se puede añadir cualquier código en la función `__init__()` que resulte necesario para crear objetos a partir de las clases

```
class Estudiante:
    def __init__(self, n):
        self.nombre = n
```

El constructor siempre tiene la palabra `self` como primer parámetro

Creando objetos a partir de clases

Ahora podemos crear tantos estudiantes como queramos con diferentes nombres:

```
s1 = Estudiante("Alicia")
print(s1.nombre)
s2 = Estudiante("Angela")
print(s2.nombre)
s3 = Estudiante("Alberto")
print(s3.nombre)
```

¿Hay otra forma de asignar nombres diferentes a estudiantes diferentes?

Ejercicio 2

Vamos a diseñar paso a paso un sistema simple para un comercio electrónico que maneja productos, clientes y órdenes.

- Añade un constructor a la clase que creaste anteriormente para que inicialice los valores de nombre y precio al instanciarse.

Métodos

Se pueden asignar diferentes valores a los atributos a través de métodos, que son funciones que definimos dentro de las clases

```
class Estudiante:
    nombre = ""
    def setNombre(self, n):
        self.nombre = n

s1 = Estudiante()
s1.setNombre("Alicia")
print(s1.nombre)
```

Todos los métodos en una clase de Python tienen que tener self *como primer parámetro*

Métodos

Se pueden asignar diferentes valores a los atributos a través de métodos, que son funciones que definimos dentro de las clases

```
class Estudiante:
    nombre = ""
    def setNombre(self, n):
        self.nombre = n

s1 = Estudiante()
s1.setNombre("Alicia")
print(s1.nombre)
```

Todos los atributos en una clase de Python tienen que ir precedidos de self. cuando se accede a ellos

Constructor y métodos

Se puede definir el constructor __init__() junto con cualquier otro método

```
class Estudiante:
    def __init__(self, n):
        self.nombre = n

    def setNombre(self, n):
        self.nombre = n

    def getNombre(self):
        return self.nombre

s1 = Estudiante("Mariano")
print(s1.getNombre())
s1.setNombre("Pedro")
print(s1.getNombre())
```

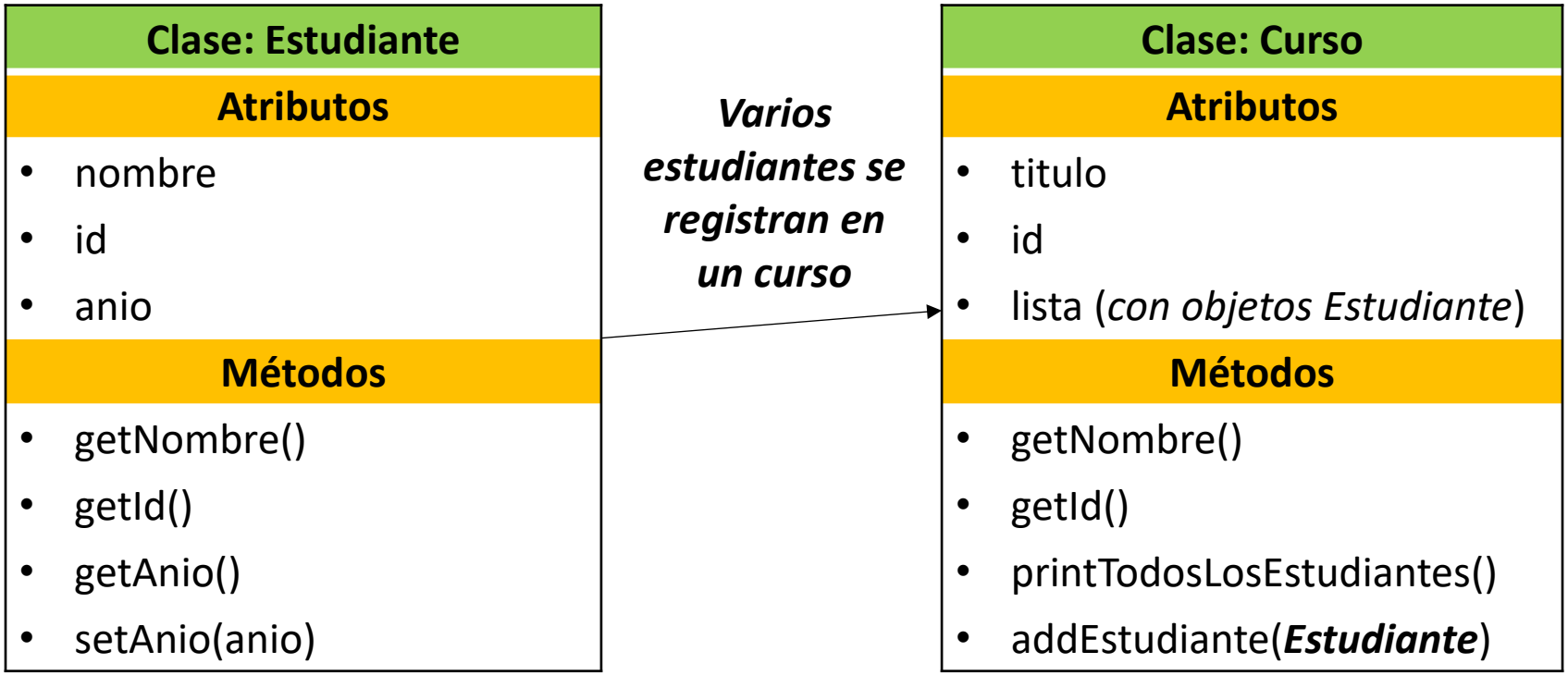
Ejercicio 3

Vamos a diseñar paso a paso un sistema simple para un comercio electrónico que maneja productos, clientes y órdenes.

- Añade un método `obtener_información` que devuelva una cadena que incluya el nombre y el precio de producto formateado correctamente

Ejemplo: estudiantes y cursos

Vamos a escribir un programa basado en objetos para las siguientes dos clases:



Ejemplo: estudiantes y cursos

```
class Estudiante:
    def __init__(self, n, id, a):
        self.nombre = n
        self.id = id
        self.anio = a

    def getNombre(self):
        return self.nombre

    def getId(self):
        return self.id
```

```
def getAnio(self):
    return self.anio

def setAnio(self, a):
    if int(a) > 0 and int(a) < 5:
        self.anio = a
    else:
        print("Año erróneo")
```

Ejemplo: estudiantes y cursos

```
class Curso:
    def __init__(self, t, id, l):
        self.titulo = t
        self.id = id
        self.lista = l

    def getTitulo(self):
        return self.titulo

    def getId(self):
        return self.id
```

Ejemplo: estudiantes y cursos

```
def printTodosLosEstudiantes(self):
    for i in self.lista:
        print(i.getNombre(), i.getId(), i.getAnio())


def addEstudiante(self, st):
    if type(st) is Student:
        for i in self.list:
            if i is st:
                print("Este estudiante ya estaba inscrito")
                return
        self.list.append(st)
    else:
        print("Lo siento, esto no es un estudiante")
```

Ejemplo: estudiantes y cursos

```
c1 = Curso("Redes de computadores", 2360, [])

s1 = Estudiante("Olga", 100, 2)
s2 = Estudiante("Daniel", 101, 2)
s3 = Estudiante("Marta", 102, 3)

c1.addEstudiante(s1)
c1.addEstudiante(s2)
c1.addEstudiante(s3)

c1.printTodosLosEstudiantes()
```

Ejercicio 4

Vamos a diseñar paso a paso un sistema simple para un comercio electrónico que maneja productos, clientes y órdenes.

- Crea otra clase que represente a un cliente y contenga: su nombre, su correo, y su cesta de la compra. Esta clase debe contener también el método `anadir_a_cesta`, que toma como argumento un objeto de la clase utilizada para representar los productos y lo añade a la cesta
- Añade otro método `calcular_total` que devuelva el precio de la cesta de la compra

Tema 5:
Herencia y polimorfismo

Bloque II. Programación orientada a objetos

Universidad
Rey Juan Carlos

Herencia

La herencia permite crear una nueva clase a partir de una clase existente.

```
# definir una superclase
class super_clase:
    # atributos y métodos

# herencia
class sub_clase(super_clase):
    # atributos y métodos de la superclase
    # atributos y métodos de la subclase
```

Herencia

La idea de herencia supone que la nueva clase usa detalles de una clase existente sin modificarla.

La nueva clase formada es una clase derivada (o clase hijo). Del mismo modo, la clase existente es una clase base (o clase padre).

```
class Animal:
    def comer(self):
        print("Puedo comer")
    def dormir(self):
        print("Puedo dormir")

class Perro(Animal):
    def ladrar(self):
        print("Puedo ladrar")

dog1 = Perro()
dog1.comer()
dog1.dormir()
dog1.ladrar()
```

Herencia

```
class Animal:
    def __init__(self, nombre):
        self. nombre = nombre
    def comer(self):
        print(“Voy a comer")
    def dormir(self):
        print(“Voy a dormir")

class Perro(Animal):
    def __init__(self, nombre):
        super().__init__(nombre)
    def ladrar(self):
        print(“Voy a ladrar")
```

Ejercicio 5

Vamos a diseñar paso a paso un sistema simple para un comercio electrónico que maneja productos, clientes y órdenes.

- Crea una clase que se llame ClienteVIP y herede de la clase cliente, con un atributo adicional: descuento, que contiene el % de descuento aplicable a la compra del cliente. Este descuento debería poder fijarse al instanciar la clase.

Polimorfismo

"Más de una forma": la misma entidad (método,…) puede llevar a cabo diferentes operaciones en diferentes escenarios.

```
class Suscriptor:
    def ver_serie(self):
        raise NotImplementedError

class SuscriptorEstandar(Suscriptor):
    def ver_serie(self):
        self.emitir_baja_calidad()

class SuscriptorPremium(Suscriptor):
    def ver_serie(self):
        self.emitir_alta_calidad()

s1 = Suscriptor()
s1.ver_serie()

s2 = Suscriptor()
s2.ver_serie()
```

Ejercicio 6

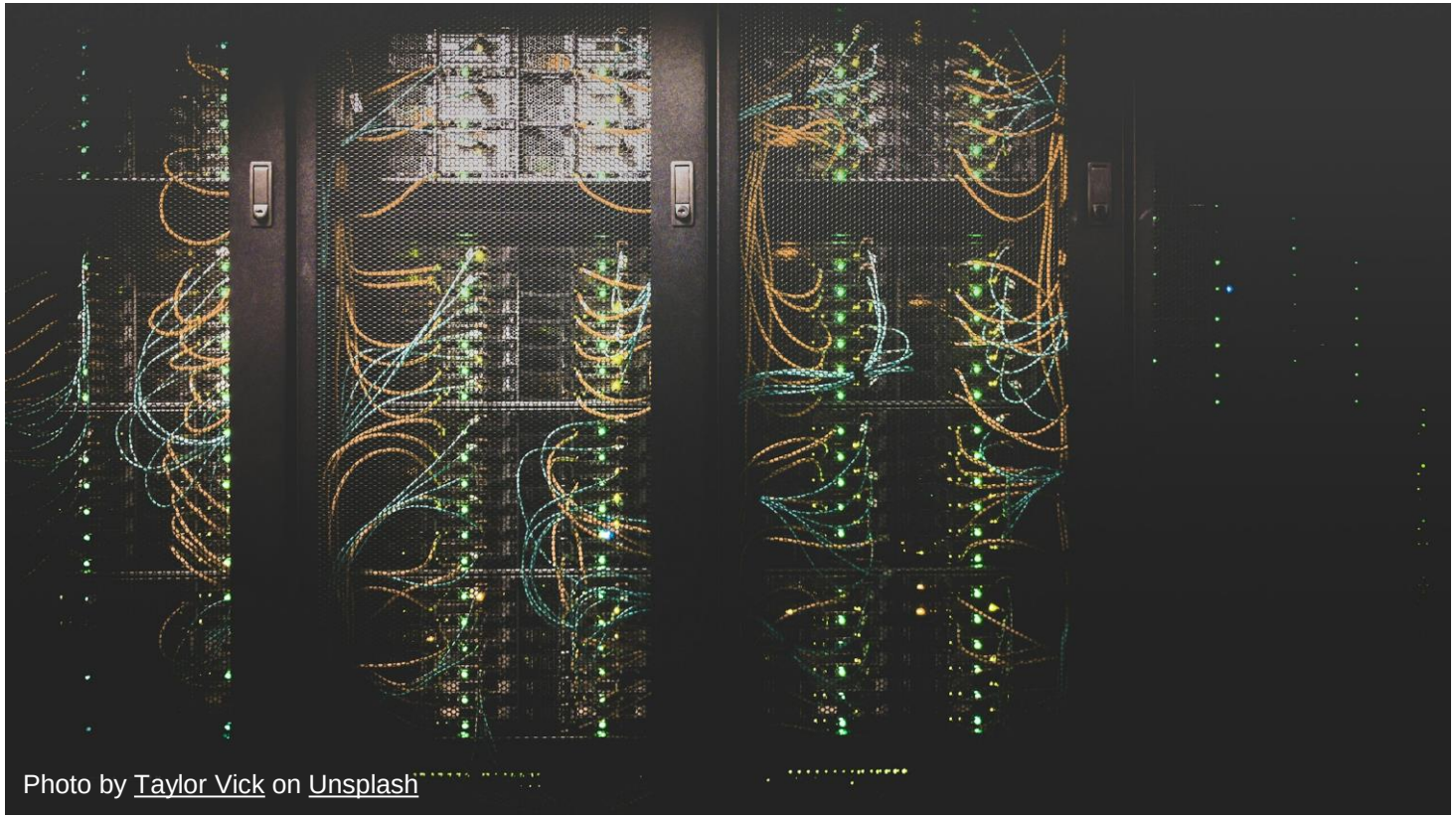
Vamos a diseñar paso a paso un sistema simple para un comercio electrónico que maneja productos, clientes y órdenes.

- Modifica la clase ClienteVIP para que `calcular_total` tenga en cuenta el descuento propio de cada cliente VIP

Informática 2

Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

Bloque III.
Programación de apps telemáticas



Universidad Rey Juan Carlos

Tema 6:
Sockets

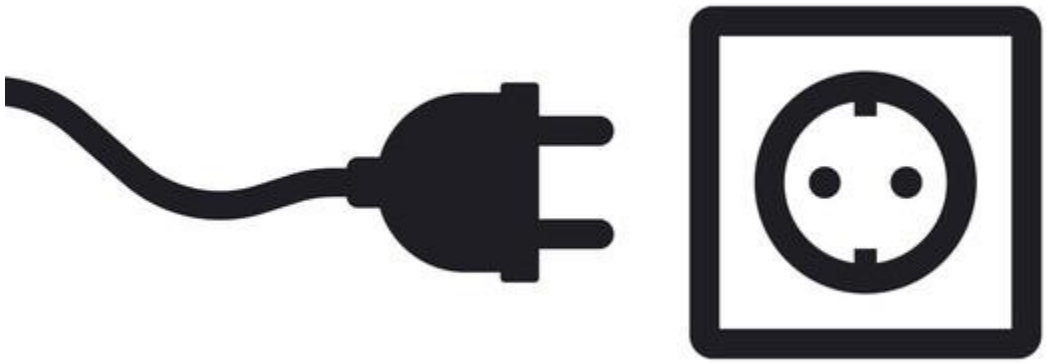
Bloque III. Programación de aplicaciones telemáticas

Comunicando procesos (en red)

Cuando **dos programas quieren comunicarse**, estando en ordenadores distintos o no, pueden hacerlo por medio de **sockets**.

Un socket es una interfaz para la comunicación entre diferentes procesos situados en **la misma o en diferentes máquinas**. En este último caso, hablamos de sockets de red.

Cuando un socket está en estado de escucha o conexión, siempre está vinculado a **una dirección IP más un número de puerto** que identifica el host (máquina) y el proceso.



Localizando a otro programa (IP + puerto)

Para que ocurra el proceso de comunicación entre dos programas, es necesario que **uno de los programas pueda localizar al otro.**

Direcciones IP

- Representan a un equipo (host) en Internet. Pueden no ser únicas (Protocolo NAT)
  - 212.128.240.50 -> www.urjc.es
- Se pueden usar sistemas DNS para traducir los nombres de los equipos a direcciones IP.

Puertos

- Es un punto virtual en el que comienzan y terminan las conexiones (entero de 0 a 65535)
- Permiten dar soporte a varias conexiones a la vez en una sola máquina.
- Un solo puerto puede mantener varias conexiones activas:
  - Cada conexión es una tupla (IP de origen, puerto de origen, IP de destino, puerto de destino)

Tipos de socket

En un **socket TCP/IP** uno de los programas conocido como "cliente", establece una conexión con el programa que está a la escucha en otro ordenador y se conoce como "servidor". Una vez **establecida la conexión**, ambos pueden enviarse datos de uno a otro y el protocolo de red garantiza que dichos datos van a llegar correctamente. Si uno de los dos programas no está o se cae, se rompe la conexión y no es posible el envío de datos.

- Es similar al teléfono. Se establece una conexión entre los teléfonos (dos extremos) y tiene lugar una conversación (transferencia de datos).

En un **socket UDP**, uno de los programas envía al otro paquetes de datos a la dirección y puerto en el que el otro está escuchando. **No se establece conexión previa**, y el programa que envía puede enviar sus datos independientemente de que el otro programa esté corriendo o no, esté escuchando o no o ni siquiera exista. Este protocolo no garantiza que los datos lleguen o que lleguen en el mismo orden que se han enviado. Solo garantiza que si el dato llega, está bien, sin errores.

- Es similar al buzón. Las cartas (datos) depositadas en el buzón se recogen y entregan (transmiten) a un buzón (toma receptora).

Sockets en Python

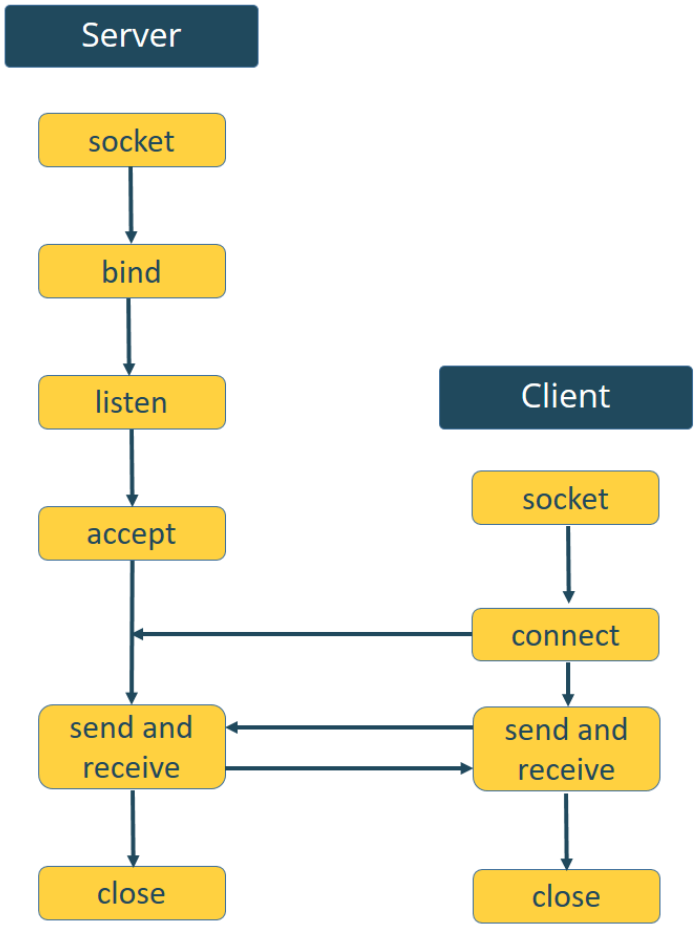
Para conectar dos procesos en Python, ambos tienen que importar la librería **socket**, que proporciona las funciones necesarias para realizar la comunicación. Para gestionar dicha conexión, existen los siguientes métodos:

Método	Descripción	Cliente	Servidor
socket()	Crea un socket	✓	✓
accept()	Acepta una conexión y devuelve: (1) un socket nuevo para comunicarte con el otro endpoint, y (2) su dirección		✓
bind()	Vincula el socket a una dirección (IP, puerto) concreta		✓
close()	Cierra la conexión	✓	✓
connect()	Abre una conexión con una dirección remota	✓	
listen()	Pone el socket en escucha para aceptar conexiones remotas		✓

Cliente-servidor

El tipo más común de aplicaciones de socket son las aplicaciones **cliente-servidor**, en las que una de las partes ofrece un servicio y espera las conexiones de los clientes para usarlo.

- Los servidores escuchan las conexiones entrantes en un número de puerto (normalmente conocido).
  - Ejemplos: 80 para HTTP, 22 para SSH...
- Los clientes suelen elegir un número de puerto arbitrario para abrir la conexión (asignado por el kernel).



Cliente-servidor TCP/IP

Cliente-servidor TCP: estableciendo conexión

server.py

```python
import socket
import time

print("Arrancando servidor...")
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((socket.gethostname(), 5555))
server_socket.listen(1)

client_socket, addr = server_socket.accept()
if client_socket:
    print("Cliente conectado: ", addr)
    time.sleep(3)

server_socket.close()

print("Apagando servidor...")
```

Tipo de socket: TCP/IP

1. Crear socket

2. Vincularlo a una dirección (IP+puerto)

3. Ponerlo en escucha

4. Aceptar conexión de un cliente

5. Comunicación (Todavía no)

6. Cerrar conexión

Cliente-servidor TCP: estableciendo conexión

client.py

Tipo de socket: TCP/IP

```
import socket
import time

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 5555))  # "127.0.0.1"

print("Conectado al servidor")
time.sleep(2)

s.close()

print("Conexión cerrada")
```

1. Crear socket

2. Conectar con el servidor (IP+puerto)

3. Comunicación (Todavía no)

4. Cerrar conexión

Enviando y recibiendo datos por sockets

Cuando la conexión ha sido establecida, el cliente y el servidor pueden intercambiar mensajes. Para ello, se hace uso de estas funciones:

Método	Tipo	Descripción
send()	TCP	Envía datos por el socket en formato ***bytes***. Devuelve el número de bytes enviados (puede que no se pueda enviar todo)
sendall()	TCP	Envía datos por el socket en formato ***bytes***. Al contrario que *send()* este método continúa enviando los datos hasta que termina (devolviendo None) o salta un error (lanzando una excepción)
recv()	TCP	Recibe datos por el socket en forma de **bytes**. Hay que decodificarlos.
sendto()	UDP	Envía datos por el socket (en formato ***bytes***) a la dirección especificada (IP + puerto). Devuelve el número de bytes enviados.
recvfrom()	UDP	Recibe datos por el socket. Devuelve una tupla **(bytes, dirección)** con los datos codificados y la información del emisor (IP + puerto)

Cliente-servidor TCP: envío de datos básicos

server.py

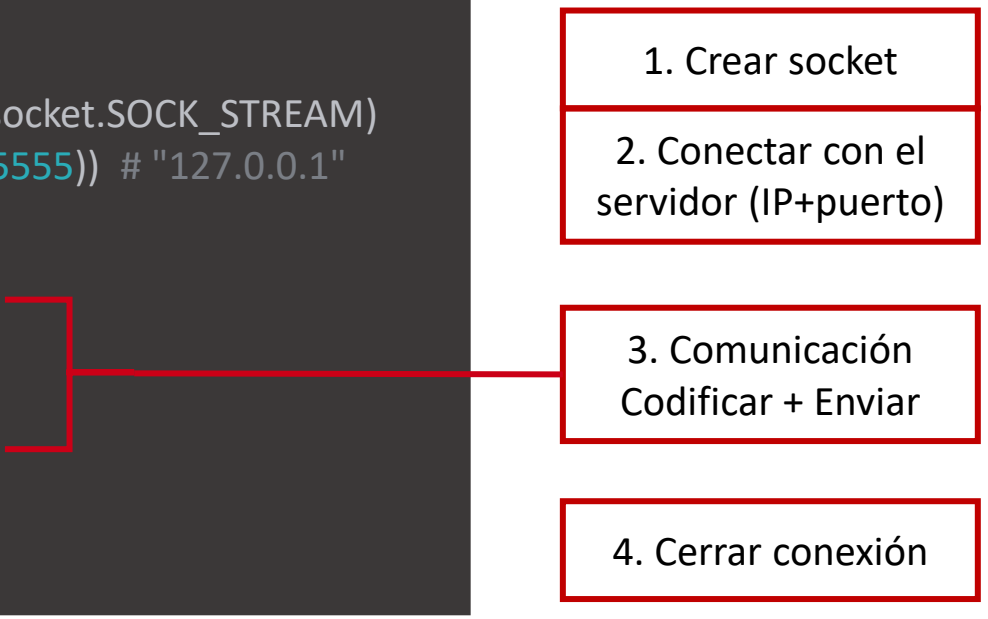
```
import socket

print("Arrancando servidor...")
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((socket.gethostname(), 5555))
server_socket.listen(1)

client_socket, addr = server_socket.accept()
if client_socket:
    print("Cliente conectado: ", addr)
    datos = client_socket.recv(1024)
    cadena_datos = datos.decode()
    print("Mensaje recibido:", cadena_datos)

server_socket.close()

print("Apagando servidor...")
```

1. Crear socket
2. Vincularlo a una dirección (IP+puerto)
3. Ponerlo en escucha
4. Aceptar conexión de un cliente
5. Comunicación: Recibir + Decodificar
6. Cerrar conexión

Cliente-servidor TCP: envío de datos básicos

client.py

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 5555))  # "127.0.0.1"
print("Conectado al servidor")

msg = "Hola amigo"
s.send(msg.encode())
print("Mensaje enviado:", msg)

s.close()
print("Conexión cerrada")
```



Ejercicio 1

Vamos a crear una aplicación cliente-servidor en Python.

- El servidor debe recibir el puerto en el que escucha como argumento por línea de comandos, y al arrancar, mostrar la IP por pantalla. Después, se queda en esperar hasta recibir una conexión de un cliente.
- El cliente debe recibir la IP y el puerto del servidor al que conectarse como argumento por línea de comandos.
- Una vez conectado, el cliente pide al usuario que introduzca por teclado su nombre completo, y enviárselo al servidor. Después, cerrará la conexión.
- El servidor mostrará el mensaje recibido por pantalla y cerrará la conexión.

Nota 1: recuerda que debes lanzar el servidor y el cliente en terminales separadas.

Nota 2: se recomienda no hacer copy-paste del código de las diapos, ya que en el examen no disponemos de materiales de apoyo.

Cliente-servidor TCP: respondiendo a un mensaje

server.py

```python
import socket

print("Arrancando servidor...")
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((socket.gethostname(), 5555))
server_socket.listen(1)

client_socket, addr = server_socket.accept()
if client_socket:
    print("Cliente conectado: ", addr)
    while True:
        datos = client_socket.recv(1024)
        if not datos:  # Cliente cierra socket
            print("Cliente desconectado: ", addr)
            break
        cadena_datos = datos.decode()
        print("Recibido:", cadena_datos)
        client_socket.sendall(cadena_datos.encode())  # Echo

server_socket.close()
print("Apagando servidor...")
```

1. Crear socket
2. Vincularlo a una dirección (IP+puerto)
3. Ponerlo en escucha
4. Aceptar conexión de un cliente
5. Comunicación: Recibir y decodificar + Codificar y enviar
6. Cerrar conexión

Cliente-servidor TCP: respondiendo a un mensaje

client.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 5555))  # "127.0.0.1"
print("Conectado al servidor")
while True:
    msg = input("Dile algo al server: ")
    s.send(msg.encode())
    response = s.recv(1024)
    print("Respuesta: %s" % response.decode())

s.close()
print("Conexión cerrada")
```

1. Crear socket

2. Conectar con el servidor (IP+puerto)

3. Comunicación
Codificar y enviar
+
Recibir y decodificar

4. Cerrar conexión

Ejercicio 2

Partiendo del ejercicio anterior, vamos a modificar el cliente y el servidor para que hablen entre sí.

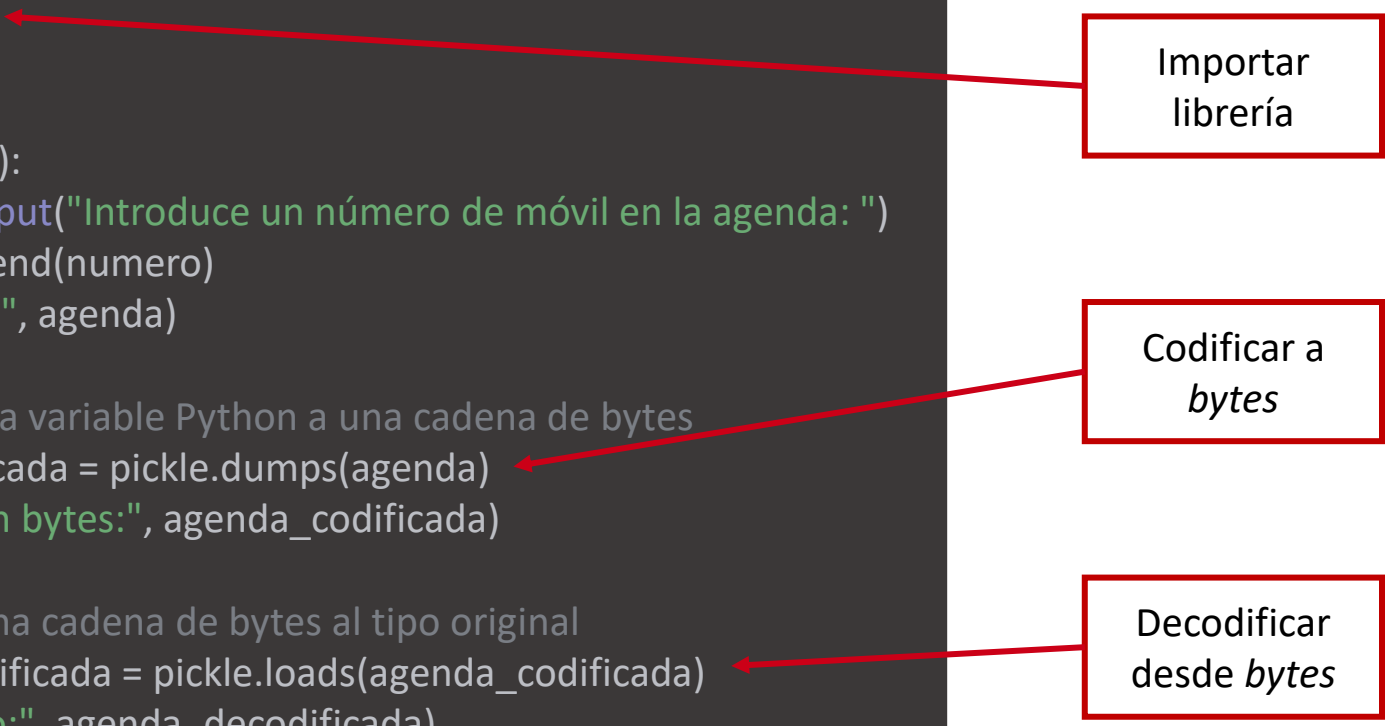
- El cliente debe poder enviar mensajes de forma continua hasta que:
  - Se pulse Ctrl+C y se termine el programa
  - Se introduzca el mensaje "bye bye", que no se enviará y el cliente cerrará la conexión

- El servidor tiene que recibir los mensajes del cliente y contestar con la misma cadena de caracteres recibida pero transformada a mayúsculas.

Serializando datos con pickle

Hemos visto que para comunicar ***strings*** por sockets podemos utilizar las funciones ***encode()*** y ***decode()*** que permiten serializarlas (convertirlas a *bytes*). Sin embargo, para el resto de tipos de datos, no existen. Para ello usaremos la librería **pickle**, y sus funciones ***dumps()*** y ***loads()***

```
import pickle

agenda = []
for i in range(2):
    numero = input("Introduce un número de móvil en la agenda: ")
    agenda.append(numero)
print("Agenda:", agenda)

# Convierte una variable Python a una cadena de bytes
agenda_codificada = pickle.dumps(agenda)
print("Datos en bytes:", agenda_codificada)

# Decodifica una cadena de bytes al tipo original
agenda_decodificada = pickle.loads(agenda_codificada)
print("Recibido:", agenda_decodificada)
```



Ejercicio 3

Partiendo del ejercicio anterior, vamos a modificar el cliente para que pueda enviar información más compleja:

- El cliente pedirá al usuario que introduzca por teclado el nombre de una canción y el artista que la canta.
- El cliente creará un diccionario con las claves "artista" y "cancion" y los valores recibidos, y lo enviará al servidor.
- El servidor tiene que recibir el mensaje del cliente, decodificarlo, guardarlo en una lista, y responder al cliente con la lista completa.

Como antes, el cliente seguirá enviando mensajes de forma continua hasta que el usuario introduzca "salir" como artista. Entonces, no enviará el mensaje y cerrará la conexión.

Cliente-servidor UDP

udp_server.py

Tipo de socket: UDP

```
import socket

print("Arrancando servidor...")
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind((socket.gethostname(), 5555))

try:
    while True:
        data, client_address = server_socket.recvfrom(1024)
        print("Recibido: " + data.decode())
        server_socket.sendto(data, client_address)
finally:
    server_socket.close()
    print("Apagando servidor...")
```

1. Crear socket

2. Vincularlo a una dirección (IP+puerto)

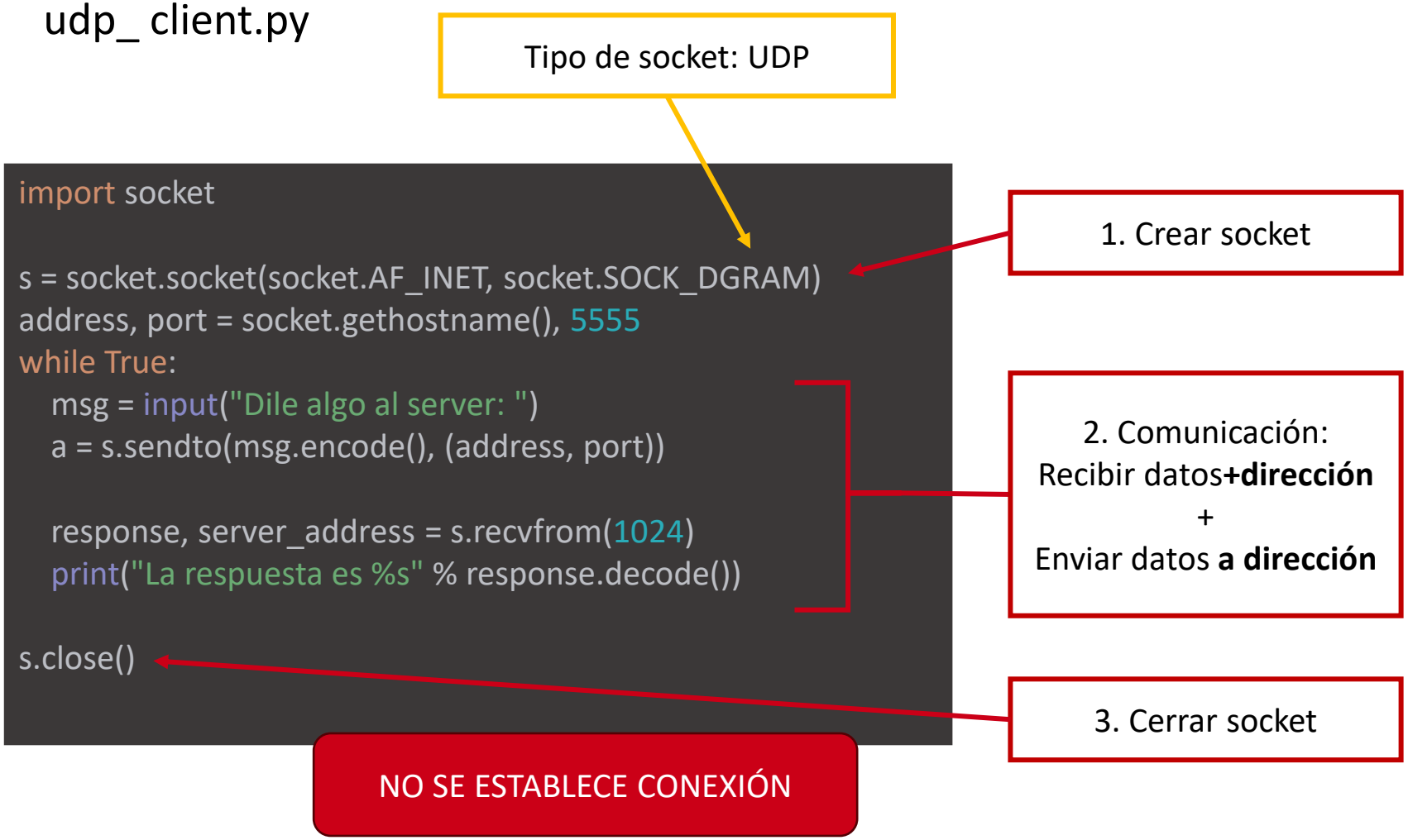
3. Comunicación:
Recibir datos**+dirección**
+
Enviar datos **a dirección**

NO SE ESTABLECE CONEXIÓN

4. Cerrar socket

Cliente-servidor UDP

udp_ client.py



Tema 7: Concurrencia

Bloque III. Programación de aplicaciones telemáticas

Recap: servidor UDP

udp_server.py

Tipo de socket: UDP

```
import socket

print("Arrancando servidor...")
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind((socket.gethostname(), 5555))

try:
    while True:
        data, client_address = server_socket.recvfrom(1024)
        print("Recibido: " + data.decode())
        server_socket.sendto(data, client_address)
finally:
    server_socket.close()
    print("Apagando servidor...")
```

1. Crear socket
2. Vincularlo a una dirección (IP+puerto)
3. Comunicación:
   Recibir datos**+dirección**
   +
   Enviar datos **a dirección**
4. Cerrar socket

NO SE ESTABLECE CONEXIÓN

Recap: cliente UDP

udp_ client.py

Tipo de socket: UDP

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
address, port = socket.gethostname(), 5555
while True:
    msg = input("Dile algo al server: ")
    a = s.sendto(msg.encode(), (address, port))

    response, server_address = s.recvfrom(1024)
    print("La respuesta es %s" % response.decode())

s.close()
```

1. Crear socket

2. Comunicación:
Recibir datos**+dirección**
+
Enviar datos **a dirección**

3. Cerrar socket

NO SE ESTABLECE CONEXIÓN

Ejercicio 4

Utilizando el ejemplo cliente-servidor UDP de las diapos anteriores:

1. Lanza el servidor en una terminal
2. Lanza un cliente en otra terminal, y comprueba que funciona.
3. En una tercera terminal, lanza un segundo cliente.

¿El segundo cliente se conecta? ¿El servidor responde?

Recap: servidor TCP

tcp_server.py

Tipo de socket: TCP/IP

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((socket.gethostname(), 5555))
server_socket.listen(1)

client_socket, addr = server_socket.accept()
if client_socket:
    while True:
        datos = client_socket.recv(1024)
        if not datos:  # Cliente cierra socket
            print("Cliente desconectado: ", addr)
            break
        print("Recibido:", datos.decode())
        client_socket.sendall(datos)  # Echo

server_socket.close()
```

1. Crear socket
2. Vincularlo a una dirección (IP+puerto)
3. Ponerlo en escucha
4. Aceptar conexión de un cliente
5. Bucle de comunicación
6. Cerrar conexión

Recap: cliente TCP

tcp_client.py

Tipo de socket: TCP/IP

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 5555))  # "127.0.0.1"
print("Conectado al servidor")
while True:
    msg = input("Dile algo al server: ")
    s.send(msg.encode())
    response = s.recv(1024)
    print("Respuesta: %s" % response.decode())

s.close()
print("Conexión cerrada")
```

1. Crear socket
2. Conectar con el servidor (IP+puerto)
3. Comunicación
   Codificar y enviar
   +
   Recibir y decodificar
4. Cerrar conexión

Ejercicio 5

Utilizando el ejemplo cliente-servidor TCP de las diapos anteriores:

1. Lanza el servidor en una terminal
2. Lanza un cliente en otra terminal, y comprueba que funciona.
3. En una tercera terminal, lanza un segundo cliente.

¿El segundo cliente se conecta? ¿El servidor responde?

Servidor TCP multicliente secuencial

Conexión única

```
import socket

server_socket = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM
)
server_socket.bind((socket.gethostname(), 5555))
server_socket.listen(1)

# Se espera a un cliente
client_socket, addr = server_socket.accept()

# Se le atiende
if client_socket:
    print("Cliente conectado: ", addr)
    while True:
        # Enviar y recibir datos
        pass

# Se cierra el servidor
server_socket.close()
```

Conexiones secuenciales

```
import socket

server_socket = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM
)
server_socket.bind((socket.gethostname(), 5555))
server_socket.listen(1)
try:
    # Hasta el infinito, atendiendo a clientes
    while True:
        # Se espera a un cliente
        client_socket, addr = server_socket.accept()
        # Se atiende
        if client_socket:
            print("Cliente conectado: ", addr)
            while True:
                # Enviar y recibir datos
                pass

except KeyboardInterrupt:
    server_socket.close()  # Con Ctrl+C
```

Ejercicio 6

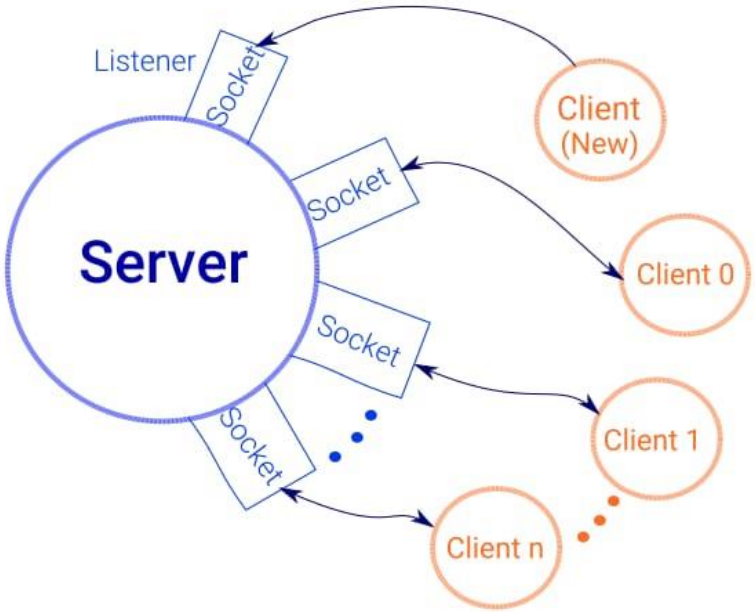
Partiendo del ejemplo de cliente-servidor TCP donde el cliente enviaba un diccionario canción-artista y el servidor almacenaba los datos en una lista:

1. Modifica el servidor para que pueda atender a varios clientes secuencialmente.
2. Prueba a conectar dos o más clientes con el servidor en paralelo desde terminales diferentes. ¿Qué ocurre?

Múltiples conexiones de cliente en paralelo

El servidor multicliente anterior permite que varios clientes interactúen con el servidor. Sin embargo, no es eficiente si cada petición tarda cierto tiempo en procesarse, ya que los clientes no activos estarán en espera hasta que termine.

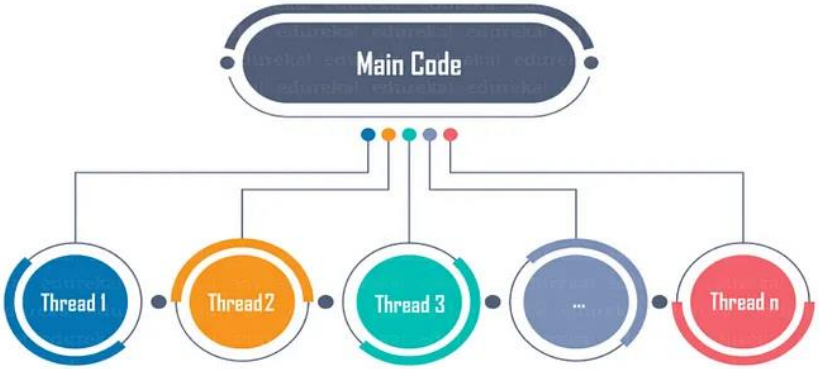
Solución: atender peticiones en paralelo -> Hilos (*threads*)



Hilos o *threads*

Un hilo es **un flujo de ejecución independiente**. Un único proceso puede constar de varios hilos que **funcionan simultáneamente**. Cada hilo de un programa realiza una tarea concreta. Los threads de un proceso comparten el mismo espacio de memoria, por lo que pueden **acceder a variables comunes** y comunicarse fácilmente.

Por ejemplo, cuando estás jugando un videojuego y tu personaje principal está explorando el mundo, otro "hilo" se encarga de reproducir la música de fondo del juego.



Threads en Python

La biblioteca estándar ***threading*** contiene las primitivas para trabajar con hilos en Python. El módulo *Thread*, encapsula los hilos y nos permite trabajar fácilmente con ellos.

Para iniciar un hilo independiente, se **crea una instancia de *Thread* indicándole la función a ejecutar y sus parámetros**. Luego arranca llamando a la función ***start()***:

```
import threading

def mifuncion(a, b, c):
    media = (a + b + c) / 3
    print("La media es:", media)

# Instancia un objeto tipo Thread que ejecute mifuncion
hilo = threading.Thread(target=mifuncion, args=(3, 8, -2))

# Lanzar el hilo
hilo.start()
# Espera a que finalize el hilo
hilo.join()
```

1. Importar librería
2. Definir la función a ejecutar en el hilo
3. Crear el hilo
4. Lanzar el hilo
5. Esperar al fin del hilo

Ejercicio 7

Utilizando la siguiente función de abajo como *target:*

1. Crea 10 hilos que reciban un índice (entero) del 0 al 9.
2. Ejecuta todos los hilos en paralelo y observa el resultado.
3. Repite varias veces la ejecución. ¿Qué pasa?

```
def siesta(indice):
    time.sleep(1)
    print(f"{indice}: se ha despertado de la siesta")
```

Servidor TCP multicliente con hilos

Para convertir un servidor TCP monohilo a un servidor capaz de atender a varios clientes en paralelo:

1. Mover la lógica de gestión de la comunicación (lo que va después del *accept()*) a una función.
2. Poner el socket a la escucha sin limitación: ~~*listen(1)*~~ -> *listen()*
3. Para cada cliente conectado, crear un hilo que ejecute esa función con los parámetros necesarios.
4. Lanzar el hilo llamando a *start()*

Ejercicio 8

Partiendo del ejercicio de cliente-servidor TCP donde el cliente enviaba una frase y el servidor se la devolvía en mayúsculas:

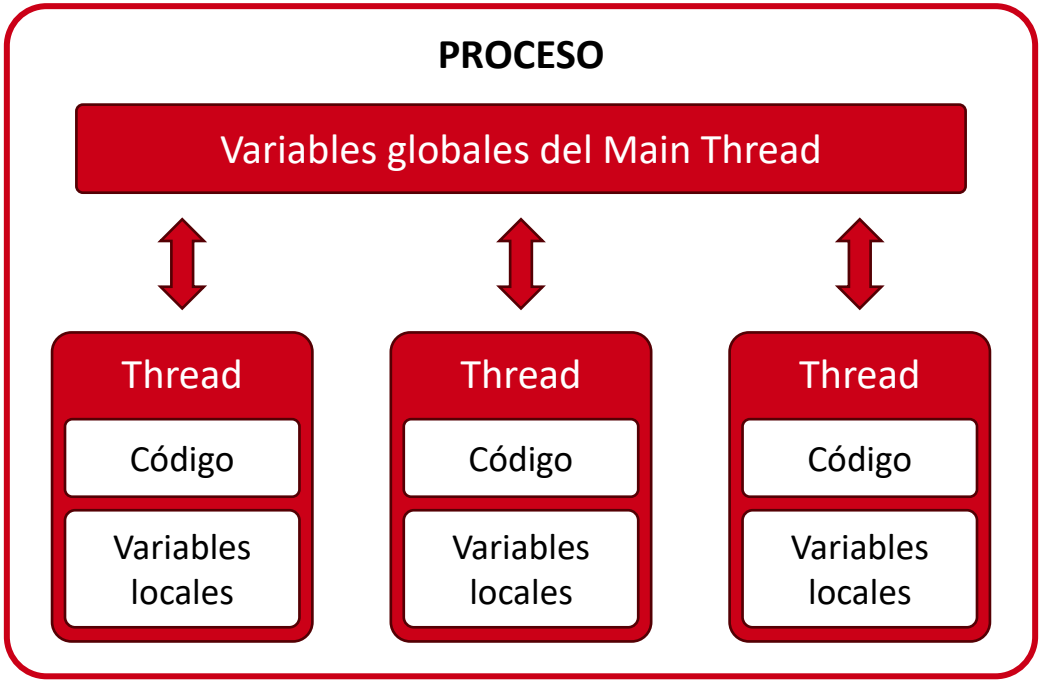
1. Modifica el servidor para que pueda atender la comunicación concurrente de varios clientes con hilos.
2. Prueba a conectar dos o más clientes con el servidor en paralelo desde terminales diferentes. ¿Qué ocurre?

Tema 8:
Exclusión mutua

Bloque III. Programación de aplicaciones telemáticas

Hilos que comparten variables

Al pertenecer a un mismo proceso, los threads **pueden acceder a las variables globales del mismo**. La lectura y escritura de estas variables desde hilos concurrentes debe de gestionarse para **asegurar la consistencia de los datos**.



Condiciones de carrera

Un *mutex lock* es un **mecanismo de sincronización** diseñado para evitar condiciones de carrera: cuando **dos hilos acceden simultáneamente a datos compartidos**, y el resultado final del programa depende de la sincronización relativa de su ejecución.

Las condiciones de carrera provocan comportamientos inesperados y corrupción de datos en un programa. Las **secciones críticas**, partes del código donde se accede a dichas variables compartidas, requieren el uso de ***mutex locks*** para garantizar la **exclusión mutua**.

Locks en Python

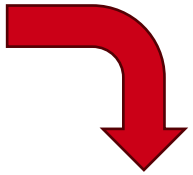
Python proporciona un bloqueo de **exclusión mutua** a través de la clase *threading.Lock*

Para coordinar la exclusión mutua entre varios hilos concurrentes, se **crea una instancia** de Lock, cada hilo la **adquiere** antes de acceder a una sección crítica, y la **libera** al terminar.

```
# Lock para garantizar la exclusión mutua
lock = threading.Lock()

# Adquirir el lock
lock.acquire()

# Sección crítica

# Liberar el lock
lock.release()
```

Mutex lock vs Condición de carrera

```python
contador = 0  # Variable global compartida

def incrementar():  # Incrementa el contador
    global contador  # Acceso a variable global
    for _ in range(100):
        contador_local = contador
        time.sleep(0.0)
        contador_local = contador_local + 1
        contador = contador_local

# Crear dos threads que ejecutan incrementar()
thread1 = threading.Thread(target=incrementar)
thread2 = threading.Thread(target=incrementar)

# Iniciar los threads
thread1.start()
thread2.start()

# Esperar a que ambos threads terminen
thread1.join()
thread2.join()

# Imprimir el resultado final
print("Valor final del contador:", contador)
```



1. Declarar Lock
2. Adquirirlo antes de acceder
3. Liberarlo después del acceso

```python
contador = 0  # Variable compartida
lock = threading.Lock()  # Para exclusión mutua

# Función que incrementa el contador
def incrementar():
    global contador  # Acceso a variable global
    for _ in range(100):
        # Adquirir el lock
        lock.acquire()
        contador_local = contador
        time.sleep(0.0)
        contador_local = contador_local + 1
        try:
            contador = contador_local
        finally:
            # Liberar el lock, incluso si excepción
            lock.release()
```

1

2

3

Ejercicio 9

Vamos a crear un programa en Python que utilice tres threads para escribir elementos en una lista compartida:

1. Declara una variable global lista donde almacenaremos los números.
2. Define una función que meta en la lista compartida cinco números enteros aleatorios (acceso mediante *global*, la lista no se recibe por parámetro).
3. Crea tres hilos que ejecuten dicha función.
4. Asegúrate de que la lista compartida esté correctamente sincronizada para evitar condiciones de carrera.

Ejercicio 10

Partiendo del servidor TCP que almacena la lista de canciones favoritas de sus clientes:

1. Modificarlo para atender a los clientes en threads.
2. Añadir la lógica de exclusión mutua necesaria para evitar condiciones de carrera cuando dos clientes quieran enviar sus canciones en paralelo.

Si quieres practicar más...

Cambia el tipo de variable que guarda la información de las canciones en el servidor para que cada cliente cree (y reciba) su propia lista de canciones privada, en vez de una lista compartida como hasta ahora.

Informática 2

Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

Bloque IV.
Estructuras de datos

Universidad Rey Juan Carlos

Tema 9:
Pilas

Bloque IV. Estructuras de datos

Estructuras de datos

- Las estructuras de datos son formas de organizar y almacenar datos en un ordenador para que puedan ser utilizados de manera eficiente.
- Son fundamentales para diseñar algoritmos eficaces y manejar grandes cantidades de datos.
- Varios tipos:
  - Lineales: Datos organizados secuencialmente. Ej.: Pilas, Colas.
  - No Lineales: Datos organizados de manera jerárquica o en redes. Ej.: Árboles, Grafos.
- Definición formal: una colección de valores de datos, las relaciones entre ellos y las funciones u operaciones que pueden aplicarse a los datos.

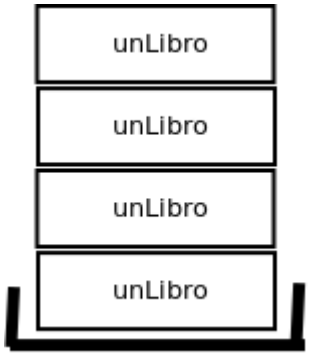
Estructuras de datos

- Cada estructura de datos proporciona una manera particular de organizar los datos para que se pueda acceder a ellos de manera eficiente, dependiendo de su caso de uso.
  - Deshacer acción con CTRL+Z
  - Atención al cliente en un chat
  - Reproductor de música
  - Búsqueda en los registros de una biblioteca
  - …

Pilas

Concepto

- Una pila es una estructura para almacenar información en la misma forma en la que apilamos físicamente objetos; por ejemplo, cajas o papeles. Es decir, uno encima del otro.
- Apilar un elemento es ponerlo en la cima de la pila
- El único elemento disponible para su extracción será la cima
- Debido a la forma en la que se insertan y se extraen los elementos, las pilas también se conocen como estructuras LIFO (*Last In – First Out*)



Usos

Algunos de los usos de las pilas son:

- El historial de visitas del navegador (botón *volver*), va apilando las páginas que visitamos.
- Igual que la lista de acciones de un programa (por ejemplo, Word), para poder hacer *undo* o deshacer
- En los lenguajes de programación, se utiliza para el manejo de las diferentes variables al realizar llamadas a las funciones (conocido como contexto de las funciones).
- Computacionalmente se pueden usar para crear mecanismos de operaciones matemáticas con pilas.

Operaciones esperadas

Esenciales

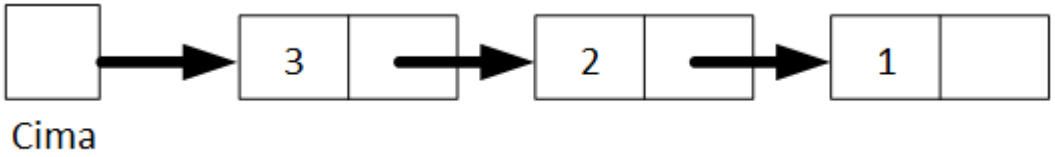
- Crear la pila
- Apilar un elemento

No esenciales

- Desapilar un elemento
- Comprobar si la pila está vacía
- Acceder a la cima

Implementación

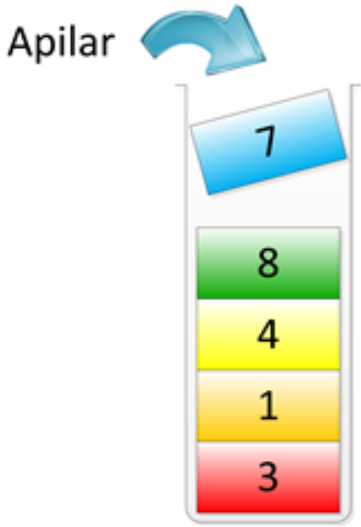
- Nodos que referencian al siguiente



- Normalmente, tiene los siguientes elementos:
  - Una variable apuntando a la cima (salvo que esté vacía)
  - Internamente, cada nodo tiene:
    - El dato en esa posición (la información que realmente se quiere almacenar en la pila)
    - Una variable apuntando al siguiente nodo

Apilar un elemento

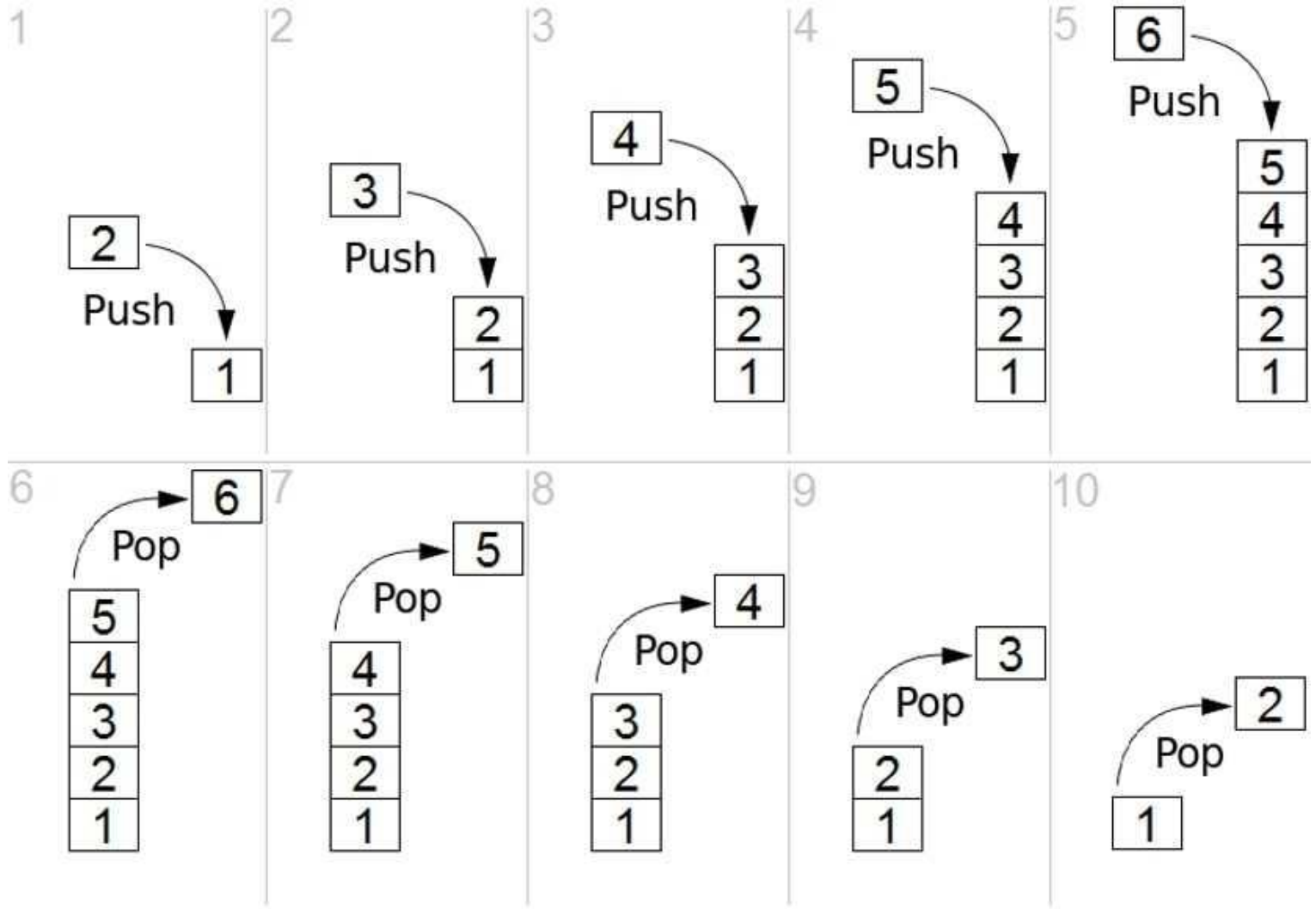
- En una pila, la inserción se hace en la cima, de forma que se ingresa un nuevo elemento apilándolo sobre el último elemento insertado con anterioridad.
- El nombre que se le da a la inserción es “apilar” o *“push”*
- Para insertar uno nuevo, se debe:
  - Reservar espacio para un nuevo nodo
  - Apuntar la referencia al “siguiente” del nuevo nodo al último elemento que había en la pila
  - Actualizar la cima, para que apunte al nuevo como “primero”



Desapilar un elemento

- Al igual que el añadir un elemento, el eliminar un ítem es una operación simple.
- Se libera el último nodo que se insertó en la estructura, el cual está referenciado por la cima, y se actualiza la cima para que apunte al siguiente nodo.
- El nombre que recibe esta operación se conoce como “desapilar” o *“pop”*.

Push/pop


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
Pop
6
5
4
3
2
1
7
Pop
5
4
3
2
1
8
Pop
4
3
2
1
9
Pop
3
2
1
10
Pop
2
1

Implementación en Python

Paso 1

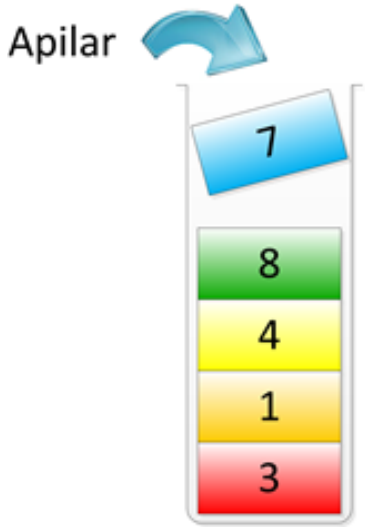
- Crear una clase Nodo que contenga como atributos (que se pasan en el constructor):
  - Dato (cada uno de los ítems que guardaremos en la pila)
  - Referencia al siguiente
- Crear una clase Pila que contenga como atributos:
  - Una referencia a la cima, que empieza vacía (imprescindible).
  - El tamaño de la pila

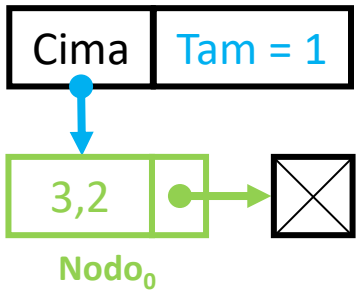
Paso 1. Solución.

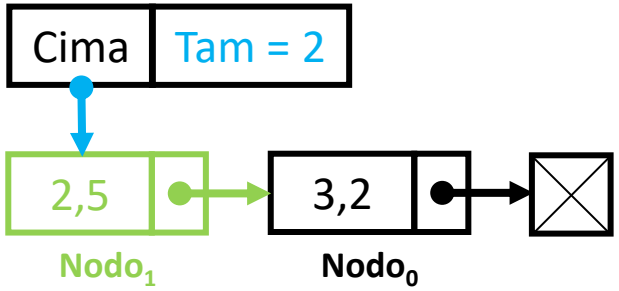
```
class Nodo:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next


class Pila:
    def __init__(self):
        self.cima = None
        self.tamanio = 0
```

Paso 2.a

- Añadir los métodos básicos a la clase Pila:
  1. Apilar un elemento (*push*)
     - Crear el elemento a añadir en la pila a partir del dato
     - Referenciarlo correctamente desde la cima





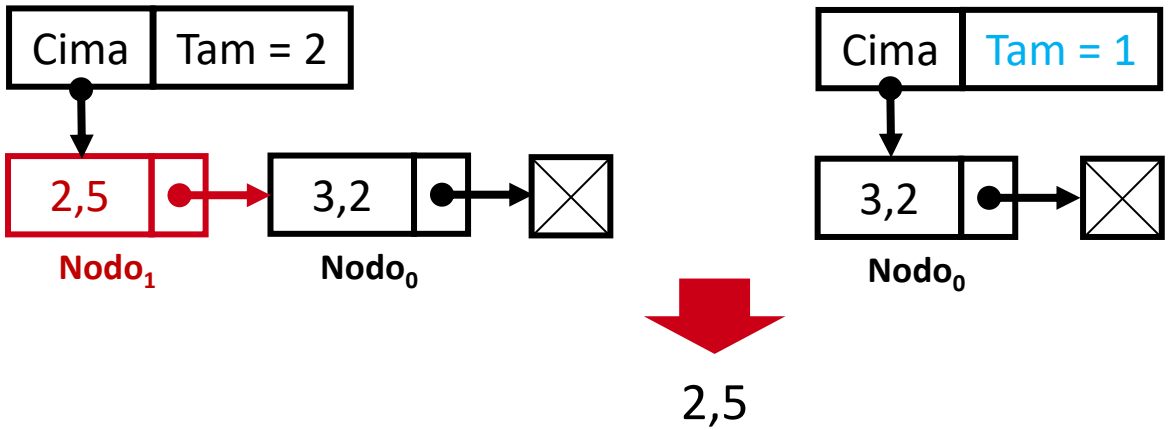


Paso 2.a Solución.

```
def apilar(self, data):
    self.cima = Nodo(data=data,
                     next=self.cima)
    self.tamanio += 1
```

Paso 2.b

- Añadir los métodos básicos a la clase Pila:
  2. Desapilar un elemento (*pull*)
     - Desapilamos cuando queremos acceder al elemento en la cima
     - Aparte de acceder al dato, elimina el elemento de la pila

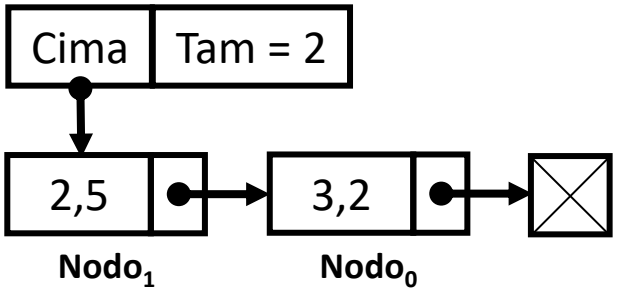


Paso 2.b Solución.

```
def desapilar(self):
    curr = self.cima
    self.cima = curr.next
    self.tamanio -= 1
    return curr.data
```

Paso 3

- Añadir otros métodos que implementen funcionalidades adicionales a la clase Pila:
  - Verificar si la pila está vacía
    - ¿Qué pasa cuando la pila está vacía?
  - Devolver el dato en la cima sin eliminarlo (*peek*)
  - Comprobar la existencia de un dato concreto
  - Mostrar la pila



Paso 3. Solución.

```
def vacio(self):
    return not self.cima
    # O bien, return self.tamanio == 0

def cima(self):
    return self.cima
```

```
def buscar(self, key):
    curr = self.cima
    while curr and curr.data != key:
        curr = curr.next
    if curr:
        return True
    else:
        return False

def mostrar(self):
    node = self.cima
    while node != None:
        print(node.data, end=" => ")
        node = node.next
    print()
```

Probando la pila

- Crea una instancia de la Pila que acabamos de definir
- Apila los números 5, 8 y 9 (el dato será un entero, en este caso)
- ¿En qué orden se quedan apilados?
  - Muestra la pila para comprobarlo
- Desapila un elemento. ¿Cómo es ahora la pila?
  - Muéstrala de nuevo para comprobarlo
- Comprueba si contiene el número 5

Probando la pila

```
# Instancia de la clase
p = Pila()

# Agregamos diferentes elementos
p.apilar(5)
p.apilar(8)
p.apilar(9)

# Imprimimos la lista de nodos
p.mostrar()

# Desapilamos el ultimo
p.desapilar()

# Imprimimos la lista de nodos
p.mostrar()

# Buscamos un elemento
print(p.buscar(5))
```

Ejercicio 1

Desarrolla una calculadora de notación postfija (o notación polaca inversa) que siga el paradigma de una pila (LIFO - Last In, First Out). La calculadora debe ser capaz de realizar operaciones básicas como suma, resta, multiplicación y división.

Requisitos:

- Estructura de pila. Implementa una estructura de pila para almacenar los operandos (números). Las operaciones deben realizarse utilizando únicamente esta estructura.
- Entrada. La calculadora recibirá una cadena de texto que representa la expresión matemática a evaluar. Ejemplo: "5 3 + 2 *“. Los números y operadores estarán separados por espacios.
- Operaciones: La calculadora debe soportar al menos las siguientes operaciones: suma (+), resta (-), multiplicación (*) y división (/). Las operaciones deben respetar el orden en que aparecen en la entrada.
- Evaluación: La expresión se evaluará de izquierda a derecha. Al encontrar un operador, se realizará la operación con los dos últimos números ingresados en la pila y el resultado se devolverá a la pila.
- Salida: La calculadora debe devolver el resultado final de la expresión.

Ejercicio 1: ejemplo

Para la entrada "5 3 + 2 *":

- Se añaden 5 y 3 a la pila.
- Se realiza la suma (5 + 3), y el resultado (8) se coloca en la pila.
- Se añade 2 a la pila.
- Se realiza la multiplicación (8 * 2).
- La salida final es 16.

Expresión infija	Expresión postfija
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +
A + B + C + D	A B + C + D +

Ejercicio 2

Escribe un programa que valide la corrección de los paréntesis en una expresión matemática. La tarea consiste en verificar si los paréntesis están balanceados correctamente. Se considera que una expresión está balanceada si cada paréntesis de apertura tiene su correspondiente paréntesis de cierre en el orden correcto.

Ejemplo: Dada la cadena de entrada "((3 + 4) * 5) / (2 - 1)", el programa deberá imprimir "Expresión balanceada".

En contraste, para la cadena "((8 * 2) - 5 / (4 + 1)", el programa deberá imprimir "Expresión no balanceada".

Utiliza una pila para llevar un seguimiento de los paréntesis de apertura y cierre correctamente.

Tema 10:
Colas

Bloque IV. Estructuras de datos

Concepto

- Una cola es una estructura de datos caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo (el posterior o final) y la operación de extracción por el otro (el anterior o frente).
- Las colas también se llaman estructuras FIFO (del inglés *First In – First Out*), debido a que el primer elemento en entrar será también el primero en salir.
- “Hacer cola para comprar unas entradas”

Operaciones esperadas

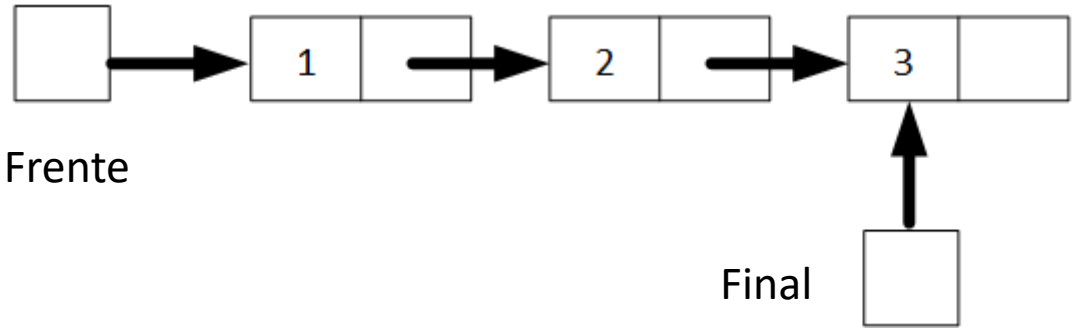
Esenciales

- Crear la cola
- Encolar un elemento (*queue*)
- Desencolar un elemento (*dequeue*)

No esenciales

- Comprobar si la cola está vacía
- Acceder al dato en el frente sin eliminarlo (*peek*)

Implementación

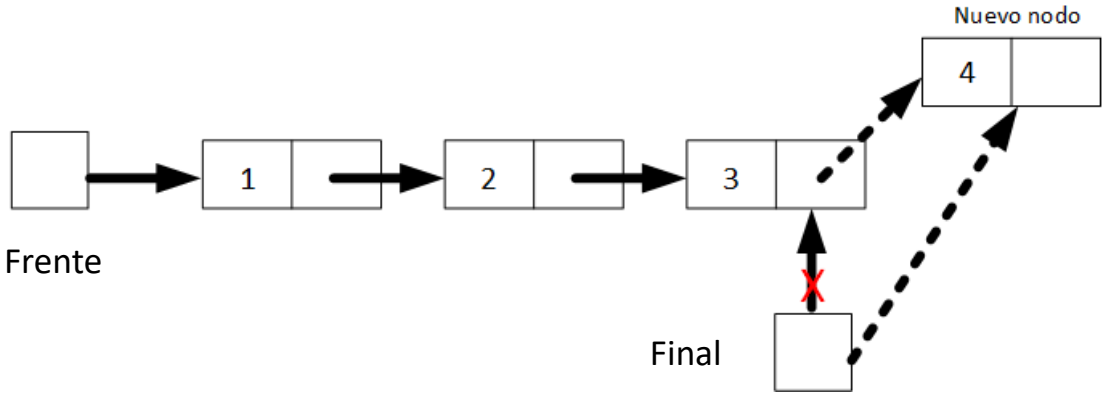


Tiene los siguientes ítems:

- Una referencia al primer nodo que se conoce como frente (salvo que esté vacía)
- Una referencia al último nodo que se conoce como final
- Los nodos tienen la misma implementación que en las pilas (datos + referencia al siguiente)

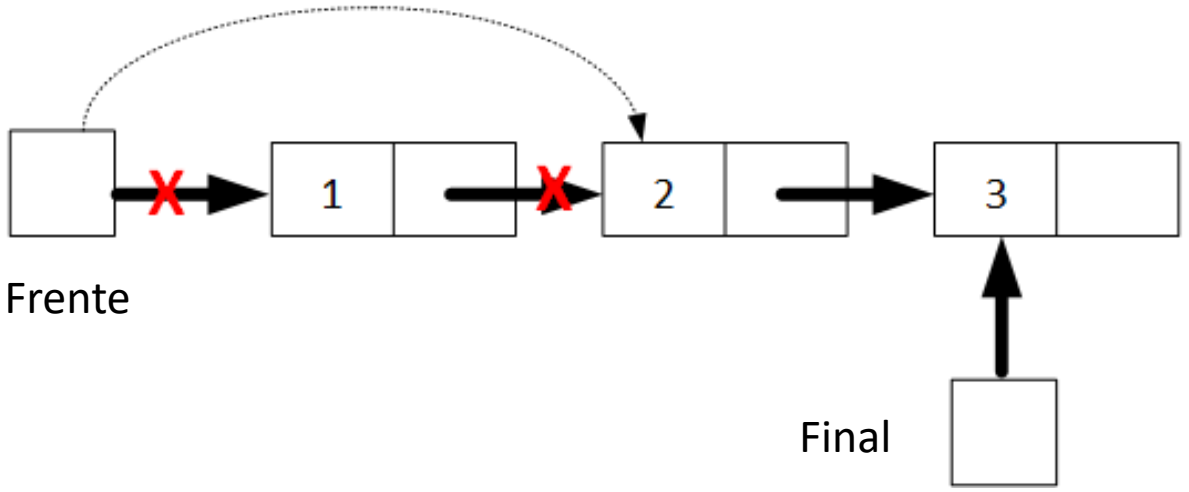
Encolar un elemento

- En una cola, la inserción se hace por el final, de forma que el elemento se inserta por la parte de atrás de la cola.
- Para insertar uno nuevo, se debe reservar espacio para un nuevo nodo, el elemento que se encontraba en el final deberá apuntar a este nuevo nodo y, por último, la referencia al final debe también apuntar al nuevo nodo.
- El nombre que recibe esta operación es la de “encolar”, “insertar” o *“enqueue”*.



Desencolar un elemento

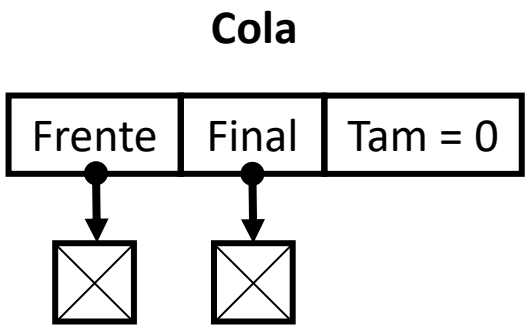
- Para extraer un elemento de la cola, se ha de extraer el primer elemento que se insertó en la estructura, el cual está referenciado por *frente*.
- El nombre que recibe esta operación se conoce como “extraer”, “desencolar” o *“dequeue”*.

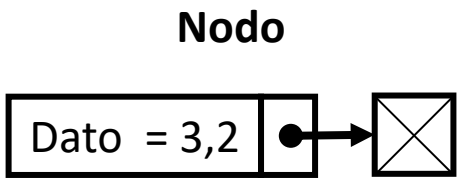


Implementación en Python

Paso 1

- Crear una clase Nodo que contenga como atributos (que se pasan en el constructor):
  - Dato (cada uno de los ítems que guardaremos en la cola)
  - Referencia al siguiente
- Crear una clase Cola que contenga como atributos:
  - Una referencia al frente, que empieza vacía (imprescindible)
  - Una referencia al final, que empieza vacía (imprescindible)
  - El tamaño de la cola



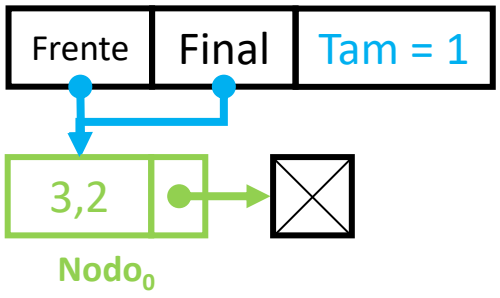


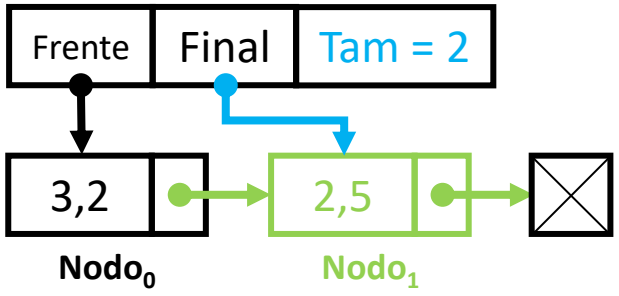
Paso 1. Solución.

```
class Nodo:
    def __init__(self, data=None, next=None):
        self.dato = data
        self.next = next

class Cola:
    def __init__(self):
        self.frente = None
        self.final = None
        self.size = 0
```

Paso 2.a

- Añadir los métodos fundamentales a la clase Cola:
  1. Encolar un elemento
     - En este caso, nos vendría bien implementar antes el método para verificar que la cola está vacía, porque las acciones a realizar cambian dependiendo de si hay algún nodo o no.





Paso 2.a Solución.

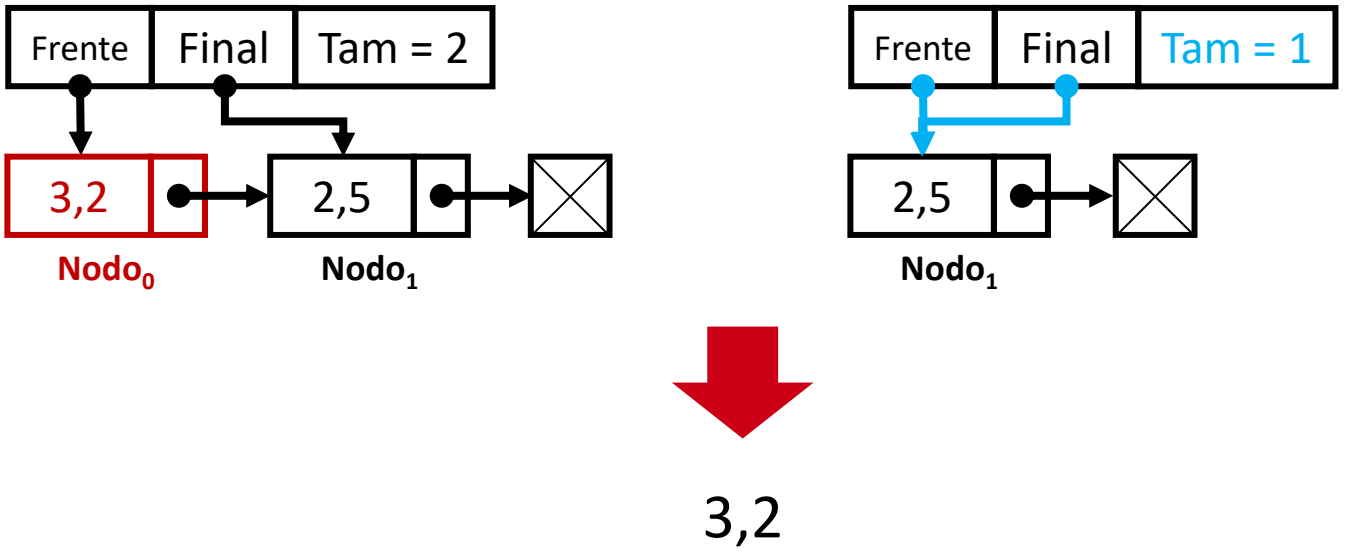
```
def encolar(self, dato):
    nodo = Nodo(dato)

    if self.vacia():  # Frente y final apuntan al nuevo
        self.frente = nodo
        self.final = nodo
    else:
        self.final.next = nodo  # Conecto el último de la fila con el nodo nuevo
        self.final = nodo  # Actualizo la referencia al último de la fila

    self.size += 1
```

Paso 2.b

- Añadir los métodos fundamentales a la clase Cola:

  2. Desencolar un elemento

     - Aquí también conviene distinguir si hay uno o más nodos

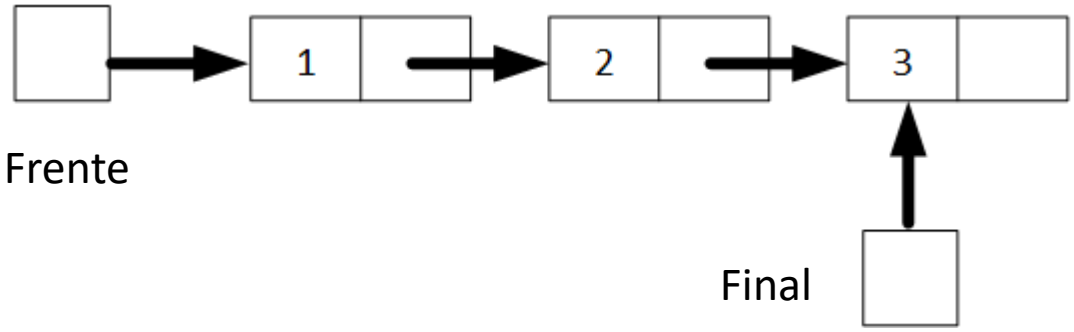


Paso 2.b Solución.

```
def desencolar(self):
    devolver = self.frente
    if self.vacia():
        return devolver  # "devolver" es None
    elif self.size == 1:  # Se queda vacía
        self.frente = None
        self.final = None
    else:  # Frente apunta al segundo (el primero sale de la cola)
        self.frente = self.frente.next

    self.size -= 1  # Disminuyo el tamaño
    return devolver.dato  # Devuelvo el valor
```

Paso 3

- Añadir métodos que implementen funcionalidades adicionales en la clase Cola:
  - Verificar si la cola está vacía
  - Ver el elemento en el frente (sin eliminarlo)
  - Comprobar la existencia de un dato concreto
  - Mostrar la cola



Paso 3. Solución.

```
def vacia(self):
    return self.frente is None and self.final is None
    # return self.size==0

def peek(self):  # Devuelve el primer elemento sin eliminarlo
    # Si no hay elementos, devuelvo None
    if self.vacia():
        return None
    # Si hay algo, devuelvo el valor del primer elemento
    return self.frente.dato
```

Paso 3. Solución cont.

```
def mostrar(self):
    # Uso actual para ir moviéndome entre nodos
    actual = self.frente  # Empiezo en el primer elemento
    while actual is not None:  # Recorro hasta el final
        print(actual.dato, end=" => ")
        actual = actual.next  # paso al siguiente
    print()

def buscar(self, dato):
    # Uso actual para ir moviéndome entre nodos
    actual = self.frente  # Empiezo en la cima
    while actual is not None:  # Recorro hasta el final
        if actual.dato == dato:  # Si encuentro, devuelvo True
            return True
        actual = actual.next
    # Si llego al final sin encontrarlo, devuelvo False
    return False
```

Probando la cola

- Crea una instancia de la Cola que acabamos de definir
- Encola los números 5, 7 y 9 (el dato será un entero, en este caso)
- ¿En qué orden se quedan encolados?
  - Muestra la cola para comprobarlo
- Extrae un elemento.
  - ¿Qué elemento esperas extraer?
- ¿Cómo es ahora la cola?
  - Muéstrala de nuevo para comprobarlo
- Comprueba si contiene el número 5

Probando la cola

```
# Instancia de la clase
c = Cola()
# Inserccion de nodos
c.encolar(5)
c.encolar(7)
c.encolar(9)
# Imprimimos la lista de nodos
c.mostrar()
# Extraemos un elemento
extraido = c.desencolar()
print(extraido)
c.mostrar()
encontrado = c.buscar(5)
if encontrado:
    print("Encontrado")
else:
    print("Nope")
```

Ejercicio 3

Crea un programa para simular un feed de noticias que contiene noticias de diferentes categorías, como deportes, política, entretenimiento, etc. Cada noticia está definida por un titular, una categoría y un autor.

Las noticias se agregan al feed y se muestran a los usuarios en el orden en que se agregaron. Sin embargo, los usuarios pueden elegir ver solo las noticias de una categoría específica.

Implementa el feed de noticias mediante una cola cuya función mostrar permita a los usuarios filtrar las noticias de una categoría.

En el main, instancia un feed e inserta cuatro noticias: una en la categoría de deportes, dos de ciencia, y otra de política. Luego, utilizando la función implementada, muestra las noticias de la categoría ciencia.

Ejercicio 4

Crea una aplicación de cliente-servidor para simular una impresora compartida utilizando una estructura de datos de cola. Los clientes pueden agregar trabajos de impresión a la cola y la impresora (el servidor) los procesará en el orden en que se agregaron.

Cuando un cliente se conecte verá un menú con tres opciones:

- **Agregar trabajo**: el usuario introducirá texto por terminal, que será enviado al servidor.
- **Mostrar pendientes**: el cliente solicita al servidor para que le envíe la cola de impresión en forma de lista.
- **Salir**: se cerrará la conexión

El servidor deberá gestionar a múltiples clientes en paralelo, recibiendo la opción del menú elegida y actuando en consecuencia. Por el contrario, tendrá un único *thread* de impresión, que revisará cada segundo si hay trabajos pendientes de imprimir. Para hacer el funcionamiento más realista, el servidor tardará 5 segundos antes de imprimir un trabajo (mostrarlo por pantalla).

Importante: para añadir y quitar elementos de la cola desde los diferentes threads, se debe garantizar la exclusión mutua.

Tema 11:
Listas enlazadas

Bloque IV. Estructuras de datos

Concepto

La lista enlazada es una estructura de datos que representa una secuencia de elementos ordenados, donde cada nodo contiene un valor y una referencia (o enlace) al siguiente nodo en la secuencia.

- La cantidad de nodos no es fija y puede crecer o decrecer dinámicamente.
- Puede contener elementos repetidos.
- No permite el acceso directo a los elementos individuales.
- Si se desea acceder a un elemento en particular, se debe comenzar por la *cabeza* y seguir las referencias hasta que llegue al elemento.

Son una “generalización” de las estructuras que hemos visto en los temas de pilas y colas.

Operaciones esperadas

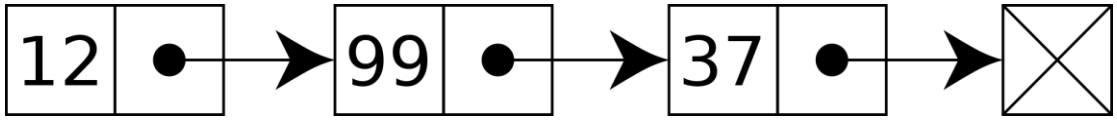
Esenciales

- Construir la lista
- Verificar si está vacía
- Insertar un elemento: al principio, al final....
- Extraer un elemento: del principio, del final...

No esenciales

- Obtener el tamaño
- Insertar un dato en una posición concreta
- Obtener el dato en una posición dada
- Eliminar el elemento en una posición dada
- Mostrar la lista, buscar un dato
- ...

Implementación

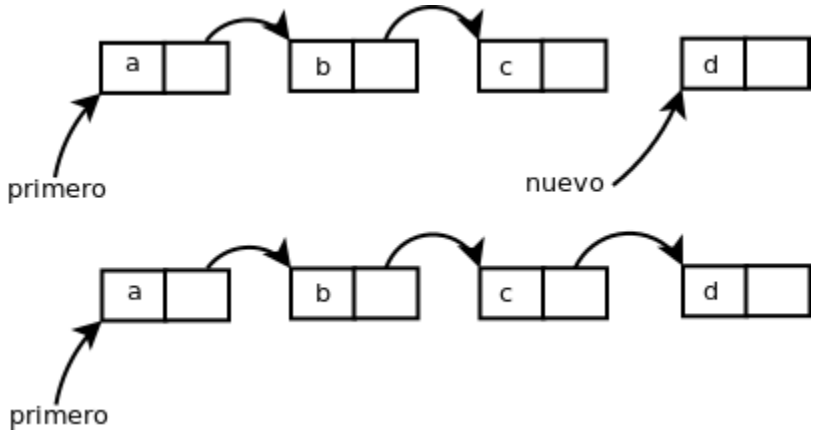


La implementación normalmente tiene los siguientes ítems:

- Una referencia al primer nodo, conocida como *cabeza* (salvo que esté vacía)
- Internamente cada Nodo tiene:
  - El dato en esa posición
  - Una referencia al siguiente nodo

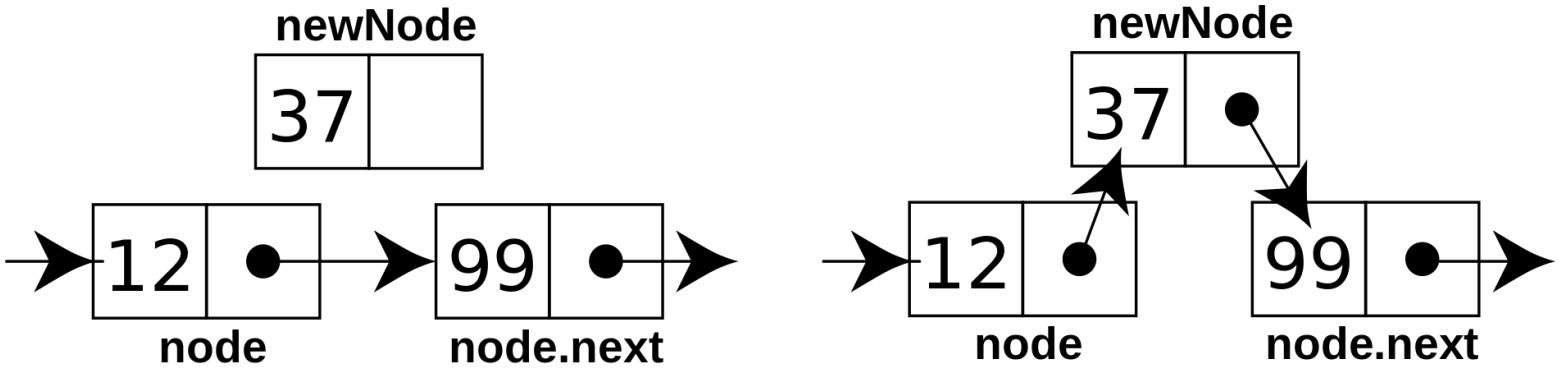
Agregar un elemento al final

- En la lista enlazada, agregar un elemento es fácil, o poco costoso en términos de uso de memoria.
- Para insertar un elemento nuevo al final, se debe reservar espacio para un nuevo nodo, y actualizar el ultimo nodo, para que apunte al nuevo como "siguiente".
- Como para agregar un elemento es necesario primero ubicar el último nodo, es necesario recorrer todos los nodos hasta llegar al final.



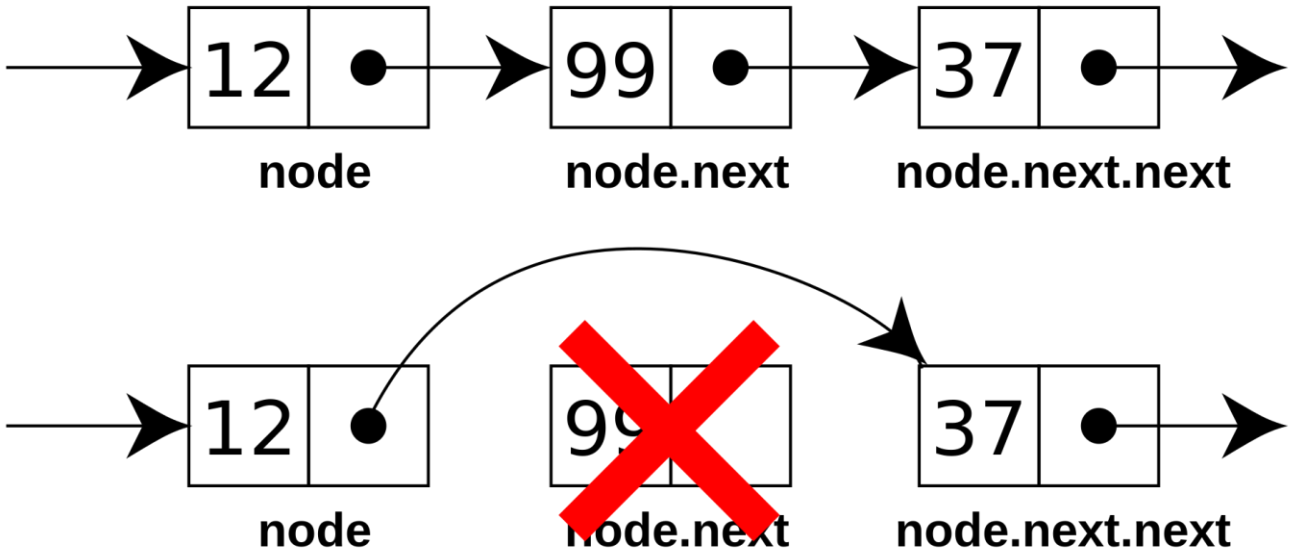
Agregar un elemento en una posición concreta

- Esta función recibe dos parámetros: el dato a insertar, y la posición en la que se desea colocar dicho elemento.
- Para realizar la inserción lo que debemos hacer es posicionarnos en el nodo anterior e introducir el nuevo nodo, actualizando las referencias del anterior al nuevo, y del nuevo al que previamente era el siguiente.



Eliminar un elemento en una posición concreta

- Al igual que el añadir un elemento, el eliminar un ítem es una operación simple. Se libera el nodo, pero primero se “enlazan" entre sí los nodos anterior y siguiente.



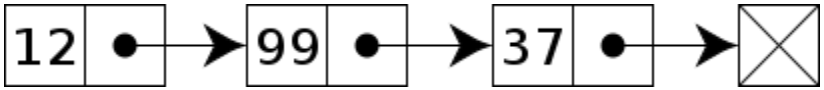
Principales problemas

La lista enlazada tiene las siguientes desventajas:

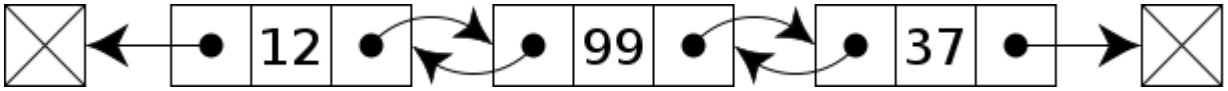
- **Acceso aleatorio:** Para encontrar el elemento en cuestión hay que realizar un recorrido todos los nodos hasta llegar al deseado.
- **Localidad:** la localidad espacial en memoria es "mala", ya que los nodos pueden estar dispersos y "alejados" en la memoria.
- **Recorrido inverso:** debido a la arquitectura de la lista enlazada, el recorrido inverso se hace muy complejo al no existir un enlace entre cada nodo y su anterior.

Tipos de listas enlazadas

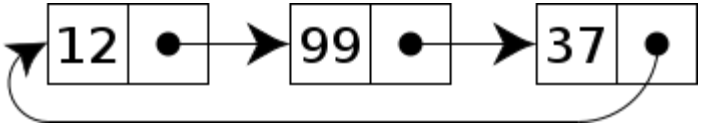
Lista enlazada simple: Cada nodo tiene un enlace que apunta al siguiente nodo en la secuencia.



Lista enlazada doble: Cada nodo tiene dos enlaces, uno que apunta al siguiente nodo y otro que apunta al nodo anterior en la secuencia.



Lista enlazada circular: La referencia del último nodo en la lista apunta al primer nodo, formando un ciclo.



Implementación en Python

Lista enlazada simple en Python

Una lista enlazada puede proveer múltiples operaciones. A continuación, implementaremos una lista enlazada simple con los siguientes métodos:

- Insertar al principio
- Insertar en una posición concreta
- Eliminar por valor
- Verificar si está vacía
- Obtener el tamaño
- Obtener el primer dato de la lista (el que está en la *cabeza*)
- Obtener el último dato de la lista
- Mostrar la lista

Paso 1

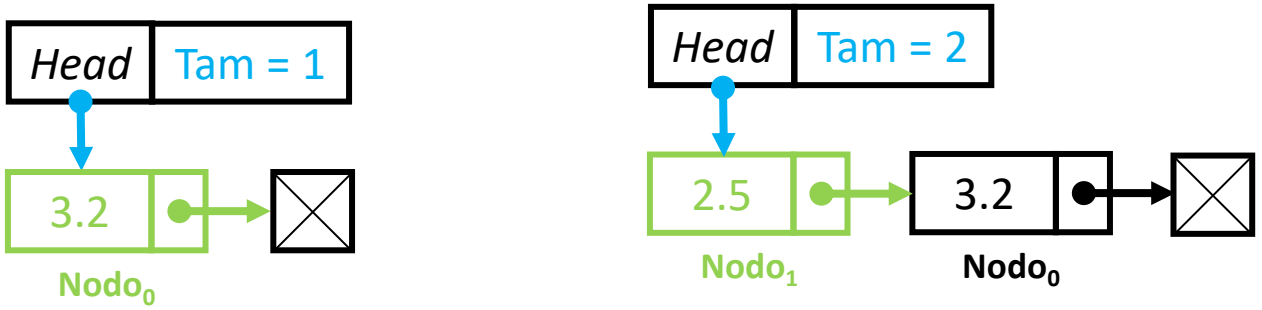
- Crear una clase Nodo que contenga como atributos (que se pasan en el constructor):
  - Dato (cada uno de los ítems que guardaremos en la pila)
  - Referencia al siguiente
- Crear una clase ListaEnlazada que contenga como atributos:
  - Una referencia a la *cabeza*, que empieza vacía (imprescindible).
  - El tamaño de la lista (opcional)

Paso 1 Solución.

```
class Nodo:
    def __init__(self, dato, next=None):
        self.dato = dato
        self.next = next


class ListaSimple:
    def __init__(self):
        self.cabeza = None
        self.size = 0
```

Paso 2.a Insertar un elemento al principio

Añadir un método para insertar un elemento por el principio (la cabeza).

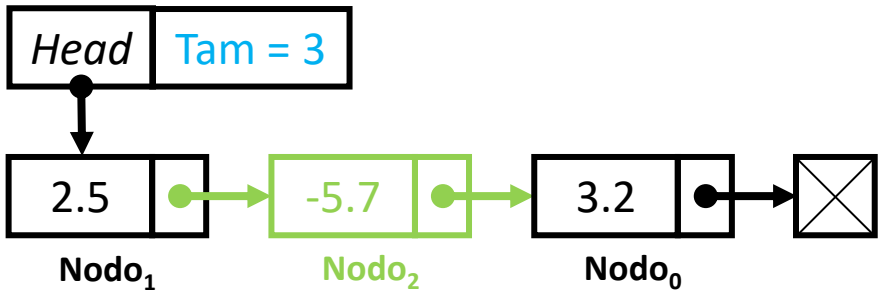


Paso 2.a Solución.

```
def insertar_principio(self, dato):
    # Crear el nodo con el nuevo dato
    nuevo = Nodo(dato)

    # Hacer que el nodo nuevo apunte a la cabeza actual
    nuevo.next = self.cabeza

    # Cambiar la cabeza para que apunte al nuevo
    self.cabeza = nuevo

    # Incrementar el tamaño
    self.size += 1
```

Paso 2.b Insertar un elemento en una posición

Añadir un método para insertar un dato en una posición concreta de la lista. La función recibe el valor a insertar así como el índice indicando la posición donde colocarlo. Si el índice es negativo o supera el tamaño de la lista, la función devolverá *False*. De lo contrario, devolverá *True*

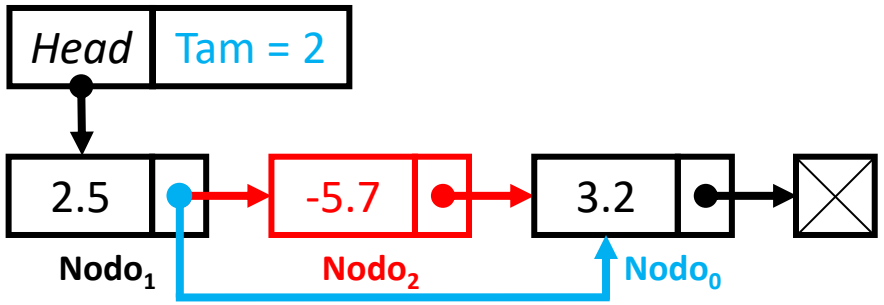


Paso 2.b Solución.

```
def insertar_posicion(self, dato, indice):
    if indice < 0 or indice > self.size:  # Compruebo que el índice es válido
        return False

    nuevo = Nodo(dato)   # Crear el nodo con el nuevo dato
    actual = self.cabeza    # Uso actual para ir moviéndome entre nodos
    anterior = None # Para guardar el nodo previo y actualizar el .next
    posicion = 0  # Para saber en que elemento estoy posicionado

    while actual is not None and posicion != indice: # Hasta posicionarme en el índice deseado
        anterior = actual  # En la siguiente iter, anterior será el nodo actual
        actual = actual.next  # En la siguiente iter, actual es el siguiente
        posicion += 1  # Actualizo posicion

    nuevo.next = actual  # Conectar nodo nuevo con el actual (mover el actual a la derecha)
    if indice == 0:  # Insertar en la primera posicion
        self.cabeza = nuevo
    else:  # Insertar en cualquier otra posicion
        anterior.next = nuevo  # Conectar nodo nuevo con el anterior

    self.size += 1  # Aumento el tamaño
    return True
```

Paso 3 Eliminar un elemento por valor

Añadir un método para eliminar un elemento por valor. Si el valor indicado no se encuentra en la lista enlazada, devolveremos *False*. Si la operación se puede realizar, devolveremos *True*.



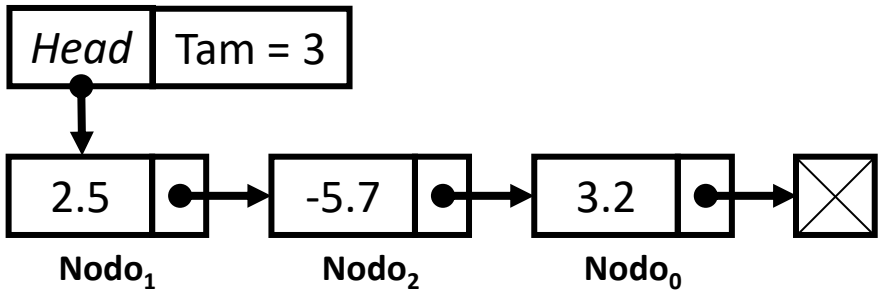
Paso 3 Solución.

```
def eliminar_valor(self, dato):
    actual = self.cabeza    # Uso actual para ir moviéndome entre nodos
    anterior = None # Para guardar el nodo previo y actualizar el .next
    posicion = 0  # Para saber en que elemento estoy posicionado

    # Recorro hasta el final o hasta que encuentro el dato
    while actual is not None and actual.dato != dato:
        anterior = actual  # En la siguiente iter, anterior será el nodo actual
        actual = actual.next  # En la siguiente iter, actual es el siguiente
        posicion += 1  # Actualizo posicion

    if actual is None:  # Lista vacia o no encontrado
        return False
    elif posicion == 0:  # Encontrado en la posicion 0
        self.cabeza = self.cabeza.next
    else:  # Encontrado en cualquier otra posicion
        anterior.next = actual.next

    self.size -= 1  # Disminuyo el tamaño
    return True
```

Paso 4

Añadir métodos para obtener información de la lista:

- Verificar si está vacía
- Obtener el tamaño
- Obtener el primer dato de la lista (el que está en la *cabeza*)
- Obtener el último dato de la lista
- Mostrar la lista



Paso 4 Solución.

```python
def vacia(self):
    return self.cabeza is None

def tamano(self):
    return self.size

def mostrar(self):
    # Uso actual para ir moviéndome entre nodos
    actual = self.cabeza  # Empiezo en la cabeza
    while actual is not None:  # Recorro hasta el final
        print(actual.dato, end=" => ")
        actual = actual.next  # paso al siguiente
    print()
```

```python
def primero(self):
    # Si no hay elementos, devuelvo None
    if self.cabeza is None:
        return None
    # Si hay algo, devuelvo el valor de la cabeza
    return self.cabeza.dato

def ultimo(self):
    # Si no hay elementos, devuelvo None
    if self.size == 0:  # tambien sirve: self.cabeza is None
        return None

    # Sabemos que cabeza no es None porque size>0
    actual = self.cabeza
    while actual.next is not None:
        actual = actual.next

    # Actual está posicionado en el último elemento
    return actual.dato
```

Probando la lista enlazada

- Crea una instancia de la ListaSimple que acabamos de definir
- Mete al principio de la lista el número 3
- Mete al principio de la lista el número 5
- Mete el número 7 en la posición 0
- Mete el número 8 en la posición 2
- Mete el número 10 en la posición 8
- Elimina el valor 5
- Elimina el valor 13
- Muestra la lista tras cada operación para ver su estado
- Imprime el primer y último valor de la lista

Probando la lista enlazada

```
l = ListaSimple()
l.insertar_principio(3)
l.mostrar()
l.insertar_principio(5)
l.mostrar()
# Insertamos el valor 7 en la posición 0
l.insertar_posicion(7,0)
l.mostrar()
# Insertamos el valor 8 en la posición 2
l.insertar_posicion(8,2)
l.mostrar()
# Insertamos el valor 10 en la posición 8
l.insertar_posicion(10,8)
l.mostrar()  # No tendría que haber cambios porque no se puede insertar
l.eliminar_valor(5)
l.mostrar()
l.eliminar_valor(13)
print(l.primero())
print(l.ultimo())
```

Ejercicio 5

Se desea implementar una clase Tren que represente un tren de mercancías formado por una serie de vagones. Cada vagón tiene un **número de identificación** y una **capacidad de carga en toneladas**. La clase Tren debe usar una lista enlazada para almacenar los vagones, ofreciendo los siguientes métodos:

__init__(self): constructor que crea un tren vacío sin vagones.

agregar_vagon(self, id, capacidad): agrega un nuevo vagón al final del tren.

insertar_vagon(self, posicion, id, capacidad): inserta un nuevo vagón en la posición indicada.

quitar_vagon(self, posicion): elimina el vagón ubicado en la posición indicada.

contar_vagones(self): devuelve el número de vagones que tiene el tren.

calcular_carga_total(self): devuelve la suma de las capacidades de carga de todos los vagones.

Ejercicio 6

Vamos a modificar la aplicación de cliente-servidor que simula una impresora compartida del tema anterior para que utilice una estructura de datos de lista enlazada.

De manera general, los clientes pueden agregar trabajos de impresión. Éstos se incorporarán a la lista y la impresora (el servidor) los procesará en el orden en que se agregaron. Sin embargo, si un cliente envía un trabajo nuevo y tiene pendiente algún otro por imprimir, el nuevo trabajo se procesará junto con los demás de ese cliente, saltándose la cola. Para facilitar la implementación, se recomienda añadir dicho trabajo justo antes del primero que tenga en cola el cliente en cuestión.

Cuando un cliente se conecte verá un menú con tres opciones:

- **Agregar trabajo**: el usuario introducirá texto por terminal, que será enviado al servidor.
- **Mostrar pendientes**: el cliente solicita al servidor para que le envíe la cola de impresión en forma de lista.
- **Salir**: se cerrará la conexión

Recuerda que el servidor debe gestionar a múltiples clientes en paralelo, recibiendo la opción del menú elegida y actuando en consecuencia. Por el contrario, tendrá un único *thread* de impresión. Para realizar las pruebas cómodamente, el servidor tardará 20s antes de imprimir un trabajo.

Importante: para añadir y quitar elementos de la cola desde los diferentes *threads*, se debe garantizar la exclusión mutua.

Tema 12:
Árboles binarios

Bloque IV. Estructuras de datos

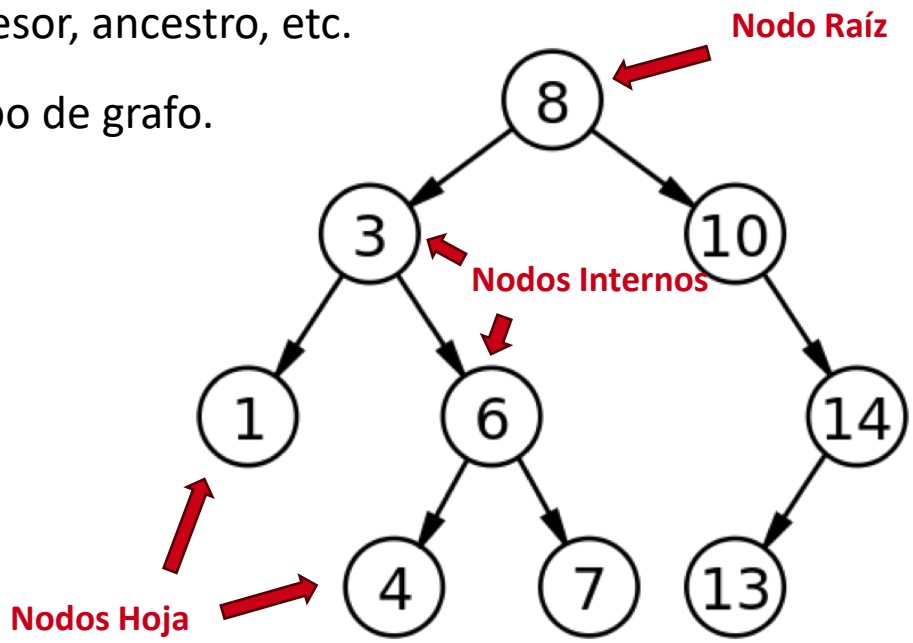
Concepto

Los árboles representan una de las estructuras de datos no lineales y dinámicas más importantes en la computación.

Es una estructura jerárquica aplicada sobre una colección de elementos u objetos llamados nodos; donde encontramos al nodo raíz, nodos internos y nodos hoja.

Entre los nodos existen “relaciones de parentesco” dando lugar a términos como padre, hijo, hermano, antecesor, sucesor, ancestro, etc.

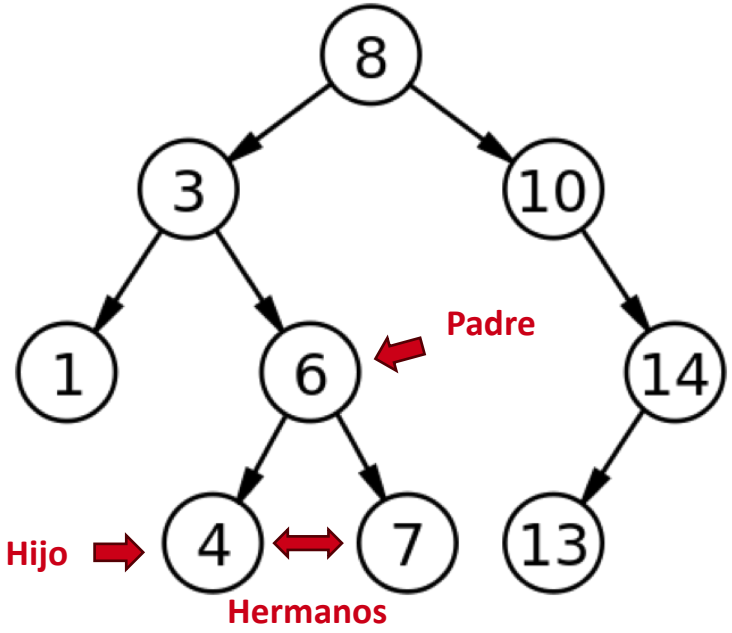
Matemáticamente, un árbol es un tipo de grafo.



Relaciones de parentesco

Al estar compuestos los árboles en forma jerárquica, se generan relaciones entre los diferentes nodos.

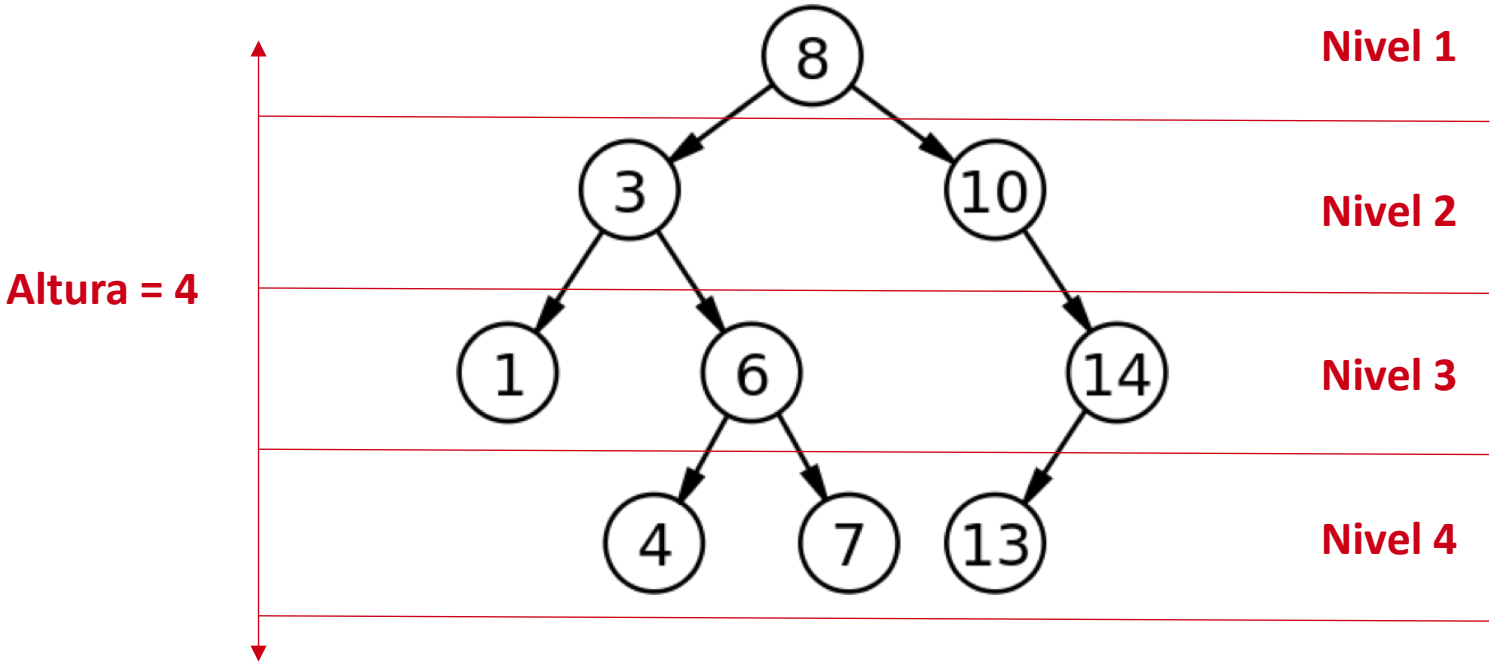
- Nodo Padre: se le llama así al nodo predecesor de un elemento
- Nodo Hijo: es el nodo sucesor de un elemento.
- Hermanos: nodos que tienen el mismo nodo padre.



Altura y nivel

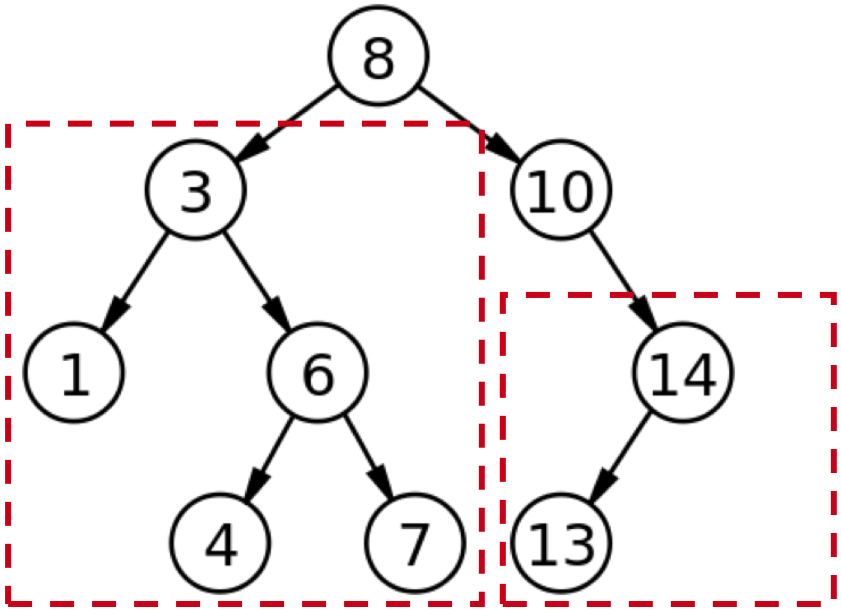
Los nodos de un árbol se organizan en niveles, que corresponde con el número de nodos que hay que recorrer hasta alcanzarlos desde de la raíz.

La altura de un árbol coincide con el nivel más alto de todos los nodos de un árbol. Es decir, con la profundidad del nodo hoja más lejano de la raíz.



Subárbol

Se llama subárbol a un árbol que forma parte de otro árbol más grande. Es un subconjunto formado por un nodo y todos los nodos que se pueden alcanzar desde él siguiendo los enlaces descendientes



Árboles binarios

Son árboles en donde cada uno de los nodos puede tener 0, 1 o 2 subárboles. Es decir, **cada nodo sólo puede tener un máximo de 2 hijos.**

Operaciones esenciales

- Construir el árbol
- Insertar un elemento
- Eliminar un elemento
- Mostrar el árbol: diferentes órdenes

Uso intensivo de **algoritmos recursivos**

No esenciales

- Verificar si está vacío / Obtener el tamaño
- Buscar un dato
- Obtener un subárbol
- Muchas más...

Recursividad (repaso)

Más comúnmente: "funciones que se llaman a sí mismas"

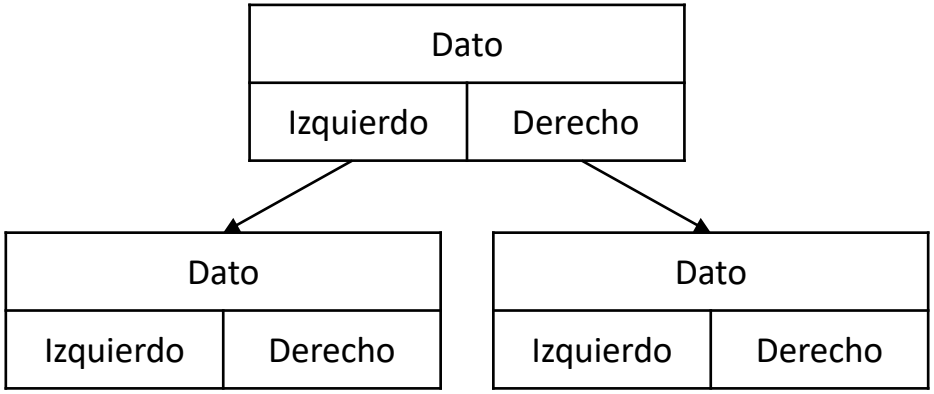
- Implica la búsqueda de algo que cambie entre cada llamada a función
- Se necesita encontrar una "condición de parada", para no caer en un bucle infinito

Ejemplo: "Implementar recursivamente una función que escriba "de 0 a 10"

```
MAX=10

def recursiva(num, max):
    print(num)
    if num < max:
        recursiva(num+1, max)

recursiva(0, MAX)
```

Implementación en Python

Implementación del Arbol Binario



La implementación normalmente tiene los siguientes ítems:

- Una referencia al primer nodo, conocida como *raíz* (salvo que esté vacía)
- Internamente cada Nodo tiene:
  - El dato en esa posición
  - Una referencia al hijo izquierdo
  - Una referencia al hijo derecho

En ocasiones cada nodo incluye también una referencia al nodo padre.

Implementación del Arbol Binario

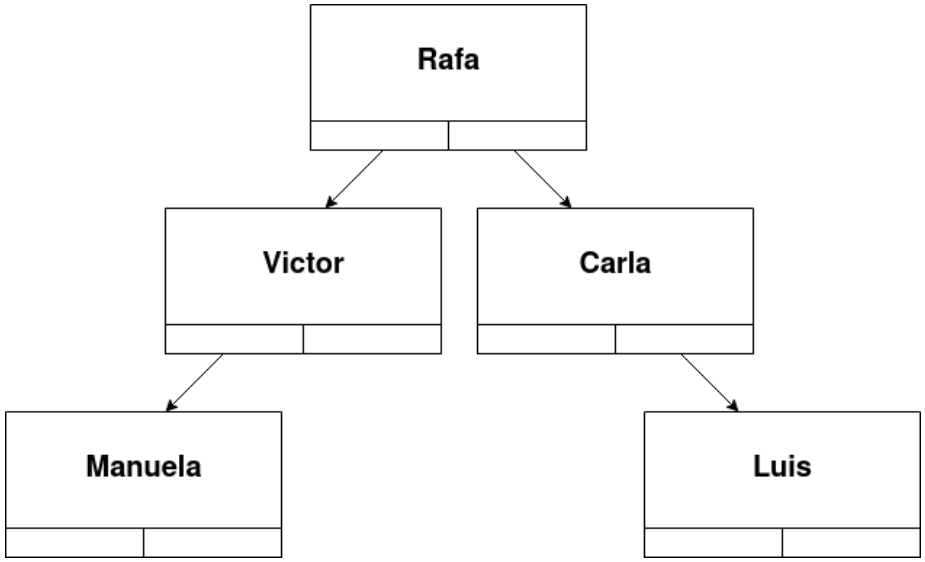
```
class Nodo:
    def __init__(self, dato):
        self.dato = dato
        self.izquierdo = None
        self.derecho = None

    # Método para comparar si este nodo es MENOR QUE otro
    def __lt__(self, other):
        return self.dato < other.dato


# Definimos la clase ArbolBinario
class ArbolBinario:
    def __init__(self):
        self.raiz = None
```

Insertar un elemento

Hay que buscar donde insertar el dato. Puede ser en una posición aleatoria o siguiendo un orden.

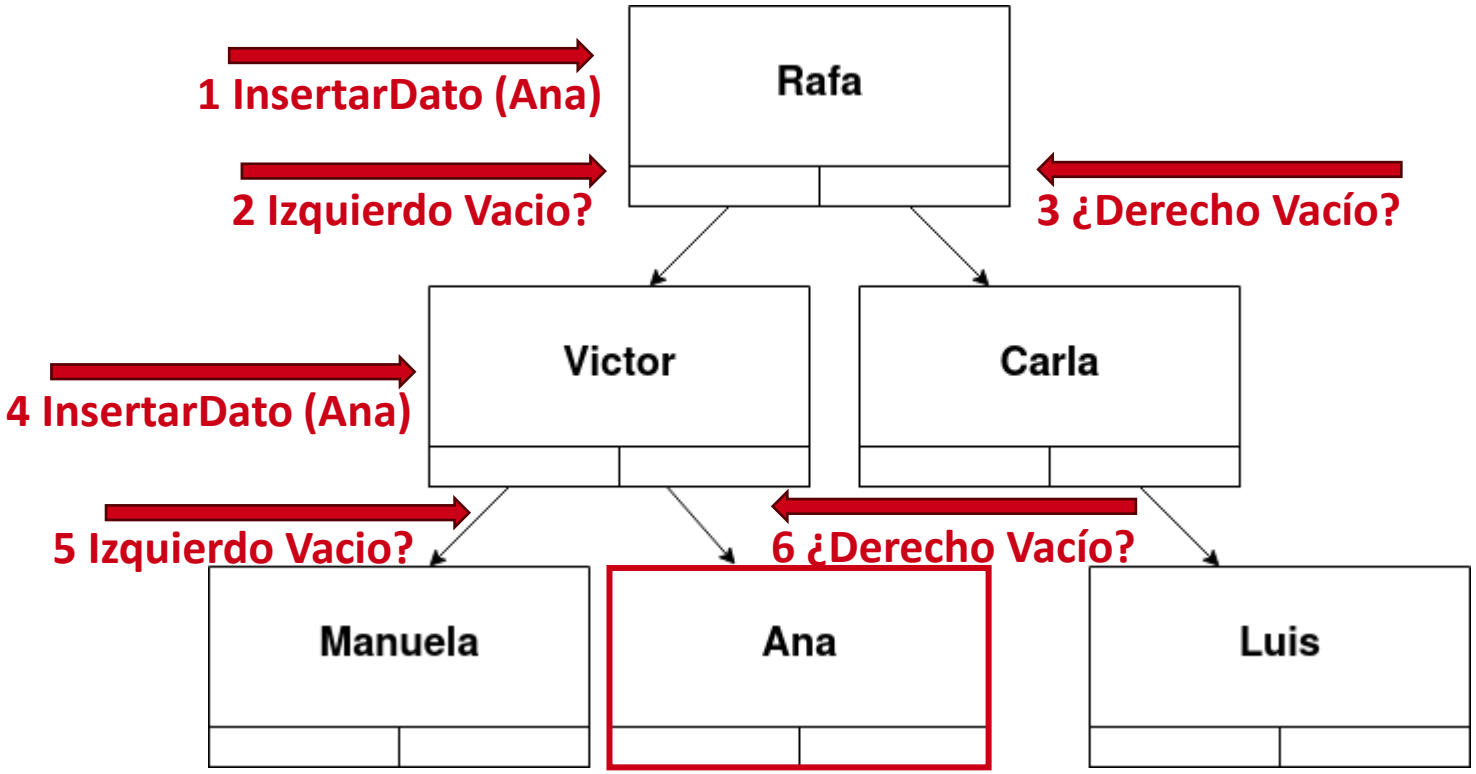
Se utiliza un **algoritmo recursivo** que navega por los niveles del árbol hasta llegar a un nodo donde se pueda insertar.

Ejemplo: se quiere insertar un nodo nuevo con el dato “Ana”



Insertar un elemento (sin ordenar)

Recorremos el árbol hasta encontrar un nodo al que le falte un hijo, y lo insertamos ahí.



Insertar un elemento (sin ordenar)

```
def insertar(self, valor):
    nodo_nuevo = Nodo(valor)
    # Si el árbol está vacío, el nodo se convierte en la raíz
    if self.raiz is None:
        self.raiz = nodo_nuevo
    # Si no, se llama al método auxiliar recursivo
    else:
        self._insertar(self.raiz, nodo_nuevo)

def _insertar(self, nodo, nodo_nuevo):
    if not nodo.izquierdo:
        nodo.izquierdo = nodo_nuevo
    elif not nodo.derecho:
        nodo.derecho = nodo_nuevo
    else:  # Elegir donde insertarlo al azar. El nodo ya tiene dos hijos
        if random.randint(0,1) == 0:
            self._insertar(nodo.izquierdo, nodo_nuevo)
        else:
            self._insertar(nodo.derecho, nodo_nuevo)
```

Recorrer el árbol

Para determinadas tareas (buscar, mostrar, etc.) hay que recorrer los nodos de un árbol. Hay tres formas de hacerlo, dependiendo del orden en el que visitamos los nodos:

- Pre-orden:
  - Visitar la raíz
  - Recorrer el subárbol izquierdo en pre-orden
  - Recorrer el subárbol derecho en pre-orden

- In-orden: izquierdo, raíz, derecho (recorriendo en in-orden)
- Post-orden: izquierdo, derecho, raíz (recorriendo en post-orden)

Algoritmo recursivo

Recorrer el árbol (y mostrarlo)

```
def inorden(self):
    self._inorden_recursivo(self.raiz)
    print("")

def _inorden_recursivo(self, nodo):
    if nodo is not None:
        self._inorden_recursivo(nodo.izquierdo)
        print(nodo.dato, end=", ")
        self._inorden_recursivo(nodo.derecho)

def preorden(self):
    self._preorden_recursivo(self.raiz)
    print("")

def _preorden_recursivo(self, nodo):
    if nodo is not None:
        print(nodo.dato, end=", ")
        self._preorden_recursivo(nodo.izquierdo)
        self._preorden_recursivo(nodo.derecho)
```

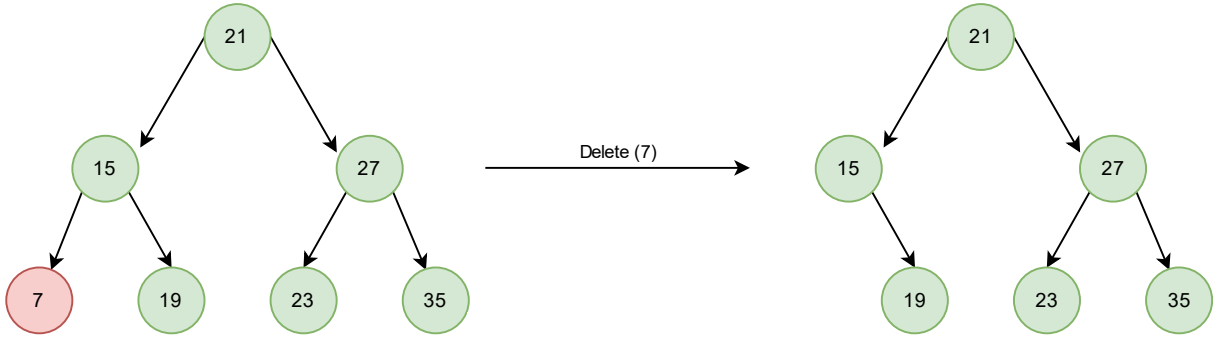
```
def postorden(self):
    self._postorden_recursivo(self.raiz)
    print("")

def _postorden_recursivo(self, nodo):
    if nodo is not None:
        self._postorden_recursivo(nodo.izquierdo)
        self._postorden_recursivo(nodo.derecho)
        print(nodo.dato, end=", ")
```

Eliminar un elemento

Para eliminar un nodo en un árbol tenemos que ocuparnos de múltiples posibilidades. Se dan tres casos:

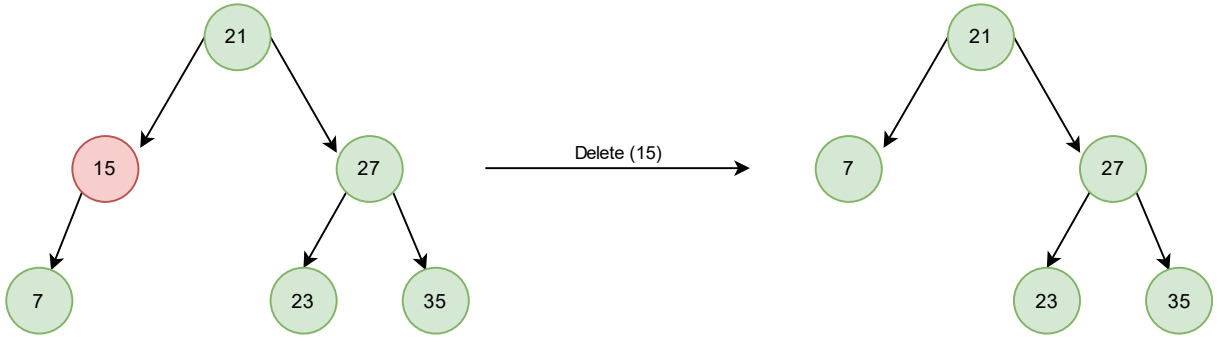
- El nodo que se va a eliminar es una hoja
- El nodo que se va a eliminar tiene un solo hijo
- El nodo que se va a eliminar tiene los dos hijos

Eliminar una hoja: el nodo 7 no tiene hijos, se puede eliminar del árbol sin problemas.

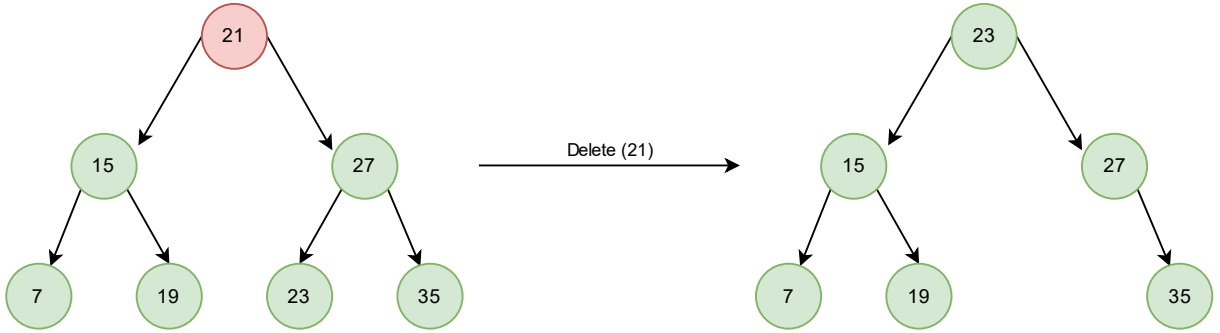


Eliminar un elemento

Eliminar un nodo con un solo hijo: el nodo 15 tiene un hijo 7; reemplazamos 15 por el subárbol de 7.



Eliminar un nodo con dos hijos: el nodo 21 tiene dos hijos: 15 y 27. Encontramos el elemento más pequeño en el subárbol derecho (23) y reemplazamos el 21, y luego llamamos a la recursividad para eliminar 23 del subárbol derecho.



Eliminar un elemento

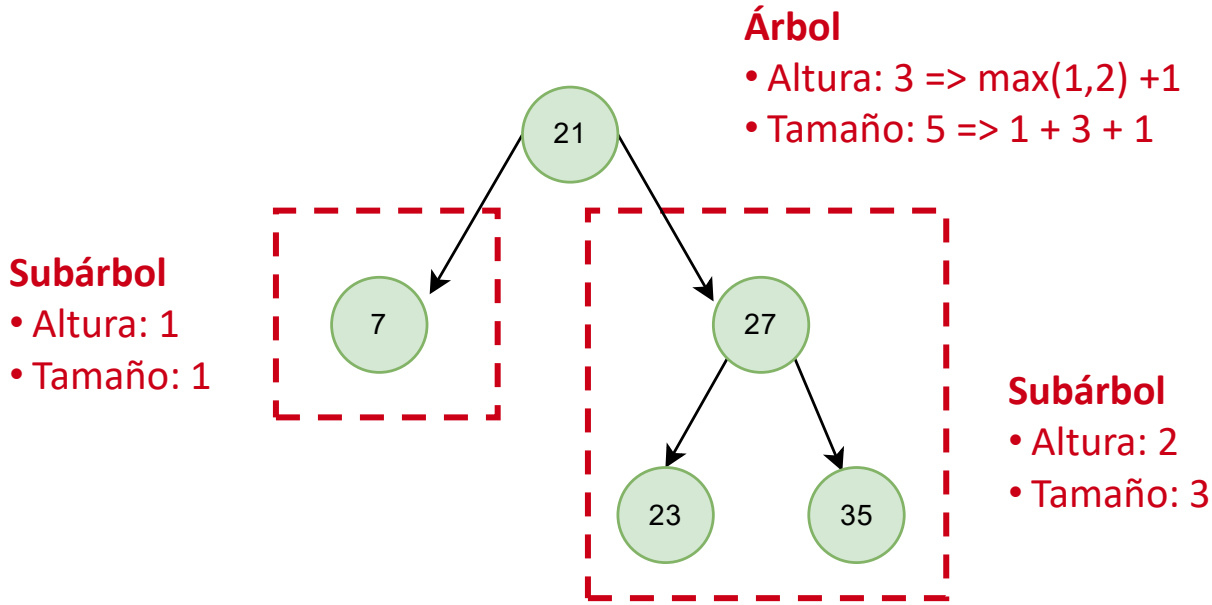
```
def eliminar(self, valor):
    self.raiz = self._eliminar(self.raiz, valor)

def _eliminar(self, nodo, valor):
    if nodo is None:
        return nodo

    if valor < nodo.dato:
        nodo.izquierdo = self._eliminar(nodo.izquierdo, valor)
    elif valor > nodo.dato:
        nodo.derecho = self._eliminar(nodo.derecho, valor)
    else:
        if nodo.izquierdo is None:
            return nodo.derecho
        elif nodo.derecho is None:
            return nodo.izquierdo

        nodo.dato = self._encontrar_min(nodo.derecho).dato
        nodo.derecho = self._eliminar(nodo.derecho, nodo.dato)

    return nodo
```

```
def _encontrar_min(self, nodo):
    while nodo.izquierdo is not None:
        nodo = nodo.izquierdo
    return nodo
```

Calcular altura y tamaño

La **altura** de un árbol o subárbol se puede calcular como la altura máxima de entre sus hijos (sus subárboles) más 1.

El **tamaño** de un árbol o subárbol es el número total de nodos, o lo que es lo mismo: si nos ubicamos en la raíz, el tamaño será la suma de los tamaños de cada hijo más 1.



Calcular altura y tamaño

```
def altura(self):
    return self._altura(self.raiz)

def _altura(self, nodo):
    if nodo is None:
        return 0
    altura_izquierda = self._altura(nodo.izquierdo)
    altura_derecha = self._altura(nodo.derecho)
    return max(altura_izquierda, altura_derecha) + 1
```

```
def tamano(self):
    return self._tamano(self.raiz)

def _tamano(self, nodo):
    if nodo is None:
        return 0
    return self._tamano(nodo.izquierdo) + self._tamano(nodo.derecho) + 1
```

Probando el árbol binario

- Crea una instancia de ArbolBinario que acabamos de definir
- Inserta el número 5
- Inserta el número 3
- Inserta el número 7
- Inserta el número 4
- Inserta el número 6
- Inserta el número 1
- Elimina el valor 7
- Elimina el valor 5
- Muestra el árbol tras cada operación para ver su estado
- Muestra el tamaño y la altura del árbol

Probando el árbol binario

```
# Probando el arbol binario
miarbol = ArbolBinario()
miarbol.insertar(5)
miarbol.inorden()
miarbol.insertar(3)
miarbol.inorden()
miarbol.insertar(7)
miarbol.inorden()
miarbol.insertar(4)
miarbol.inorden()
miarbol.insertar(6)
miarbol.inorden()
miarbol.insertar(1)
miarbol.inorden()
miarbol.eliminar(7)
miarbol.inorden()
miarbol.eliminar(5)
miarbol.inorden()

print("Tamaño:", miarbol.tamano())
print("Altura:", miarbol.altura())

print("Imprimiendo árbol inorden: ")
miarbol.inorden()

print("Imprimiendo árbol preorden: ")
miarbol.preorden()
```

Ejercicio 7

Queremos crear una filmoteca donde cada película tiene un título, un género, un año de estreno y una calificación.

Para facilitar su gestión, vamos a almacenarla como un árbol binario, que nos permita encontrar rápidamente las películas por título.

Para ello, vamos a definir una clase *Pelicula* que almacene sus datos, y una clase *Filmoteca* que herede de *ArbolBinario* y permita las operaciones de: inserción de películas, la búsqueda por título (que retorna el objeto película), y visualización de la BBDD completa.

El criterio de ordenación del árbol será el título de la película, de forma que las películas con títulos alfabéticamente menores se sitúen en la rama izquierda y las películas con títulos alfabéticamente mayores se sitúen en la rama derecha.