

Apuntes de Estructuras de Datos

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Índice

1. Taller de Python	1
1.1 Introducción a Python	1
1.2 Tipos de datos	5
1.3 Estructuras de control en Python	14
1.4 Ámbitos de ejecución	18
1.5 Funciones y paradigma funcional	29
1.6 Programación Orientada a Objetos (POO)	37
1.7 Manejo de excepciones	45
1.8 Introspección y reflexión	51
1.9 Archivos	54
1.10 Persistencia de datos	62
2. Grafos	75
2.1 Grafos	75
2.2 Recorridos	85
2.3 Orden Topológico	89
2.4 Caminos Mínimos	92
3. Representación y Adquisición de Datos	98
3.1 Registros de datos	98
3.2 JavaScript Object Notation (JSON)	110
3.3 EXtensible Markup Language (XML)	114
3.4 Expresiones Regulares (Regex)	120
3.5 La Web y las APIs	136
3.6 Web Scraping	145
4. Recuperación de la Información	156
4.1 Introducción a la recuperación de la información	156
4.2 Índices Invertidos	160
4.3 Compresión de Índices	176
4.4 Árboles B - Índices ordenados	183
5. Aplicaciones	192
5.1 Recuperación de la Información de las Redes Sociales	192
6. Anexos	203
6.1 Activación automática de entorno virtual en Git Bash	203
6.2 Registrarse como Desarrollador de Meta (Facebook)	205
6.3 Registrarse como Desarrollador de X (Twitter)	211
7. Referencias	216
7.1 Bibliografía	216
Bibliografía	217

1. Taller de Python

1.1 Introducción a Python

Python es un lenguaje de programación multipropósito, creado a fines de los años 80 por Guido van Rossum.



Guido Van Rossum es el principal autor de Python, y su continuo rol central en decidir la dirección de Python es reconocido, refiriéndose a él como Benevolente Dictador Vitalicio (en inglés: *Benevolent Dictator For Life*, BDFL); sin embargo el 12 de julio de 2018 declinó de dicha situación de honor sin dejar un sucesor o sucesora y con una declaración altisonante:

“Entonces, ¿qué van a hacer todos ustedes? ¿Crear una democracia? ¿Anarquía? ¿Una dictadura? ¿Una federación?”

— Guido Van Rossum

Más sobre la historia de Python en Wikipedia.

1.1.1 El zen de Python

El Zen de Python es una colección de principios que guían el diseño y la filosofía del lenguaje:

```
import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Estos principios enfatizan la importancia de escribir código claro, legible y elegante.

1.1.2 Características principales

1.1.2.1 Multiparadigma

Python es un lenguaje de programación **orientado a objetos**, **introspectivo** y **reflexivo**, **imperativo** y **funcional**.

Permite usar diferentes estilos de programación según las necesidades del proyecto, incluso combinando varios estilos en un mismo proyecto.

La programación imperativa se basa en la ejecución secuencial de instrucciones. Para realizar una tarea se debe programar paso a paso especificando **como** se debe hacer.

```
# Programación imperativa
def factorial(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado

print(f"Factorial de 5: {factorial(5)}")
```

La programación funcional se basa en el uso de funciones puras y evita el estado mutable. Se enfoca en *que* se debe hacer, utilizando funciones de orden superior y evitando efectos secundarios. En el capítulo de funciones profundizaremos un poco más en este paradigma.

```
# Quicksort en una línea (expresión)
qs = lambda lst: (
    lst
    if len(lst) <= 1
    else qs([x for x in lst[1:] if x < lst[0]])
    + [lst[0]]
    + qs([x for x in lst[1:] if x >= lst[0]])
)

lista = [3, 6, 8, 10, 1, 2, 1]
print(f"Lista ordenada: {qs(lista)}")
```

1.1.2.2 Fuertemente tipado y dinámico

Python es un lenguaje **fuertemente tipado**, lo que significa que no se permite realizar operaciones entre tipos de datos incompatibles sin conversión explícita.

```
# Ejemplo de tipado fuerte
resultado = "5" + 3 # Esto generará un error

# Conversión explícita necesaria
resultado_correcto = int("5") + 3
print(f"Resultado correcto: {resultado_correcto}")

# Tipado dinámico - las variables pueden cambiar de tipo
variable = 42 # int
print(f"Tipo: {type(variable)}, Valor: {variable}")

variable = "Python" # str
print(f"Tipo: {type(variable)}, Valor: {variable}")

variable = [1, 2, 3] # list
print(f"Tipo: {type(variable)}, Valor: {variable}")
```

En este fragmento el mismo identificador `variable` se liga a diferentes tipos de datos a lo largo del tiempo, demostrando el tipado dinámico de Python. No hace falta declarar el tipo de la variable al momento de su creación, Python lo infiere automáticamente.

1.1.2.3 Sintaxis clara y legible

Python se caracteriza por su sintaxis clara y legible:

```
def es_numero_primo(n):
    """
    Determina si un número es primo.

    Args:
        n (int): El número a evaluar

    Returns:
        bool: True si es primo, False en caso contrario
    """
    if n < 2:
        return False
```

```

for i in range(2, int(n**0.5) + 1):
    if n % i == 0:
        return False

return True

```

```

# Filtra una lista de números quedándose solo con los primos
numeros = [2, 3, 4, 5, 17, 25, 29]
primos = [num for num in numeros if es_numero_primo(num)]
print(f"Números primos: {primos}")

```

Los bloques de código se delimitan por indentación, lo que mejora la legibilidad y evita errores comunes de sintaxis.

En la línea 1 se define una función `es_numero_primo` que recibe un número entero `n` y devuelve `True` si es primo, o `False` en caso contrario. La definición de una función se realiza con la palabra clave `def`, seguida del nombre de la función, los parámetros entre paréntesis y dos puntos (`:`). Los dos puntos indican el inicio de un bloque de código que debe estar indentado, en este caso el cuerpo de la función.

El cuerpo de la función se extiende hasta la línea 18 a partir de la cual el código vuelve a estar alineado a la izquierda, indicando que ya no forma parte del bloque de la función.

Entre las líneas 2 y 10 encontramos la documentación de la función, que explica su propósito, los argumentos que recibe y el valor que devuelve. Esta documentación se escribe entre comillas triples (`"""`) y es accesible a través de la función `help(es_numero_primo)`. `help` es una función de Python que muestra la documentación de un objeto.

```
help(es_numero_primo)
```

Esta forma de documentar el código junto con el código mismo es una buena práctica que facilita la comprensión y el mantenimiento del código.

La forma de indentar los bloques de código es fundamental en Python. A diferencia de otros lenguajes que utilizan llaves (`{, }`) o palabras clave como `begin` y `end`, Python utiliza la indentación para definir el alcance de los bloques de código. Esto significa que todos los bloques deben estar correctamente indentados para evitar errores de sintaxis. Por ejemplo la indentación puede ser de 4 espacios, que es la convención más común en Python.

1.1.3 Aplicaciones de Python

Python se utiliza en una amplia gama de aplicaciones, gracias a su versatilidad y a la gran cantidad de bibliotecas y frameworks disponibles. Algunas de las áreas más comunes son:

Área	Frameworks/Bibliotecas	Ejemplos de uso
Desarrollo web	Django, Flask, FastAPI	Sitios web, APIs REST, aplicaciones web
Ciencia de datos	Pandas, NumPy, Matplotlib	Análisis de datos, visualización
Machine Learning	TensorFlow, PyTorch, Scikit-learn	Modelos predictivos, IA
Automatización	Selenium, Requests, BeautifulSoup	Web scraping, automatización de tareas
Desarrollo de juegos	Pygame, Panda3D	Juegos 2D y 3D

1.1.4 Instalación y configuración

1.1.4.1 Verificación de la instalación

Primero, verifica si Python ya está instalado:

```
import sys

print(f"Versión de Python: {sys.version}")
print(f"Plataforma: {sys.platform}")
```

1.1.4.2 Instalación por sistema operativo

Windows

1. Descarga el instalador desde python.org
2. Ejecuta el instalador
3. **Importante:** Marca "Add Python to PATH"
4. Verifica la instalación:

```
python --version
pip --version
```

Linux (Ubuntu/Debian)

```
# Actualizar repositorios
sudo apt update

# Instalar Python y pip
sudo apt install python3 python3-pip

# Verificar instalación
python3 --version
pip3 --version
```

MacOS

```
# Usando Homebrew (recomendado)
brew install python

# o usando el instalador oficial desde python.org

# Verificar instalación
python3 --version
pip3 --version
```

1.1.5 Recursos para aprender Python

Para profundizar en Python, puedes consultar los siguientes recursos:

- Documentación oficial de Python
- Tutorial de Python
- Tutorial de Python en Inglés (W3Schools)
- Guía de estilo PEP 8
- Dive Into Python 3 (libro gratuito)
- Comunidad Python en Argentina
- Apuntes del curso de Python - HEKTOR DOCS

1.2 Tipos de datos

Python tiene varios tipos de datos integrados. A continuación, veremos los más usados: **números, cadenas, listas, tuplas y diccionarios**.

1.2.1 Tipos numéricos

int Números enteros positivos o negativos, de cualquier tamaño. Los enteros se representan en memoria como una secuencia de bits, limitados por la memoria disponible

float Números con parte decimal, representados en memoria como un conjunto de bits en punto flotante

complex Números complejos, representados en memoria como dos números de punto flotante

1.2.1.1 Ejemplos

```
a = 5 # int
b = 3.14 # float
c = 2 + 3j # complex
```

1.2.1.2 Operaciones comunes

```
import math

x = 10 + 5 # suma
y = 10 / 3 # división (devuelve float)
u = 10 // 3 # división entera
w = 2**3 # potencia
v = 10 % 3 # módulo (resto de la división)
z = math.sqrt(-1) # Número complejo (0+j)
z2 = complex(0, 1) # También se puede crear un número complejo directamente
z3 = 1j # Representación alternativa de un número complejo
```

1.2.1.3 Conversión de tipos

En python los tipos numéricos se pueden convertir entre sí, ya sea de forma explícita o implícita.

```
float(5)
int(3.7)

x = 10 + 5.5 # Implícitamente convierte el int a float
print(x)
```

Un operador útil para verificar el tipo de una variable es `type()`:

```
y = 10 + (2 - 3j) # (12-3j)
print(type(y))
```

No hay que confundir la conversión implícita de tipos con tipado dinámico. En Python, el tipo de una variable se determina en tiempo de ejecución, pero las operaciones entre tipos incompatibles generarán un error.

```
# Esto generará un error de tipo
x = 5 + "10"
```

1.2.2 Cadenas de caracteres (str)

Una cadena es una secuencia **immutable** de caracteres.

Un objeto **immutable** es aquel cuyo contenido no puede ser modificado una vez creado. En el caso de las cadenas, esto significa que no se pueden cambiar los caracteres individuales de una cadena después de su creación.

```
mensaje = "Hola, mundo"
mensaje[0] = "h" # Esto generará un error
```

Si quiero modificar una cadena, debo crear una nueva cadena con el contenido deseado:

```
mensaje = "Hola, mundo"
mensaje = "h" + mensaje[1:] # Crea una nueva cadena
print(mensaje)
```

1.2.2.1 Indexado y *slicing*

Las cadenas de caracteres están indexadas, es decir, que se puede manipular cada carácter por su posición. El primer carácter tiene índice 0, el segundo 1, y así sucesivamente. También se pueden usar índices negativos para acceder a los caracteres desde el final de la cadena y tajadas o *slicing* para obtener subcadenas. En las tajadas, el primer parámetro es el índice inicial y el segundo es el índice final (no incluido), similar a Go.

```
nombre = "Python"
print(nombre[0]) # 'P'

nombre = "Python"
print(nombre[-1]) # 'n'

nombre = "Python"
print(nombre[1:4]) # 'yth'

nombre = "Python"
print(nombre[:2]) # 'Py'

nombre = "Python"
print(nombre[2:]) # 'thon'

nombre = "Python"
print(nombre[-3:]) # 'hon'
```

Las tajadas o *slices* en Python tienen un tercer parámetro opcional que indica el paso entre los índices. Por ejemplo, `nombre[:2]` devuelve cada segundo carácter de la cadena.

```
nombre = "Python"
print(nombre[:2]) # cadena con paso 2: 'Pto'

nombre = "Python"
print(nombre[::-1]) # invierte la cadena: 'nohtyP'
```

1.2.2.2 Métodos útiles para manipular cadenas

```
texto = "Hola mundo"
print(texto.upper()) # Convierte a mayúsculas: 'HOLA MUNDO'

texto = "Hola mundo"
print(texto.lower()) # Convierte a minúsculas: 'hola mundo'

texto = "Hola mundo"
# Reemplaza las apariciones de "mundo" por "Python"
print(texto.replace("mundo", "Python"))

texto = "Hola mundo"
# Divide la cadena en una lista de palabras, separando por espacios
print(texto.split())

texto = "Hola mundo"
print("Python" in texto) # verifica si 'Python' está en texto

texto = "Hola mundo"
len(texto) # devuelve la longitud de la cadena
```

1.2.2.3 Concatenación

Las cadenas se pueden concatenar usando el operador `+` o multiplicar por un número entero para repetirlas.

```
saludo = "Hola " + "mundo"
print(saludo) # 'Hola mundo'
```

```
saludo = "Hola mundo"
saludo2 = saludo * 3 # saludo+saludo+saludo
print(saludo2) # 'Hola mundoHola mundoHola mundo'

saludo = "Hola mundo"
saludo3 = (saludo + ". ") * 3 # Agrega un punto y un espacio al final
print(saludo3) # 'Hola mundo. Hola mundo. Hola mundo. '
```

1.2.2.4 Iteración sobre cadenas de caracteres

Las cadenas de caracteres son **iterables**, lo que significa que se pueden recorrer carácter por carácter usando un bucle for.

```
for caracter in "Python":
    print(caracter) # 'P', 'y', 't', 'h', 'o', 'n'
```

1.2.2.5 Formateo de cadenas

El formateo de cadenas permite insertar valores en una cadena de texto de manera más legible y flexible. Hay varias formas de hacerlo:

- Usando el método `format()`

```
nombre = "Juan"
edad = 30
mensaje = "Hola, mi nombre es {} y tengo {} años.".format(nombre, edad)
print(mensaje) # 'Hola, mi nombre es Juan y tengo 30 años.'
```

- Usando f-strings (Python 3.6+)

```
nombre = "Ana"
edad = 32
mensaje = f"Hola, mi nombre es {nombre} y tengo {edad} años."
print(mensaje) # 'Hola, mi nombre es Ana y tengo 32 años.'
```

La letra `f` antes de la cadena indica que es una f-string, lo que permite insertar variables directamente dentro de llaves `{}`.

- Usando el operador `%` (menos recomendado)

```
nombre = "Eva"
edad = 28
mensaje = "Hola, mi nombre es %s y tengo %d años." % (nombre, edad)
print(mensaje) # 'Hola, mi nombre es Eva y tengo 28 años.'
```

El carácter `%` se usa para formatear cadenas, donde `%s` es un marcador de posición para una cadena y `%d` para un número entero, pero es menos legible y flexible que las otras opciones.

El carácter de escape `\` se utiliza para insertar caracteres especiales en una cadena, como comillas, saltos de línea o tabulaciones.

```
mensaje = "Hola, \"mundo\".\n¿Cómo estás?"
print(mensaje)
```

`\n` inserta un salto de línea, y `\"` permite incluir comillas dobles dentro de una cadena delimitada por comillas dobles.

Otra forma de usar comillas dobles en una cadena es usar comillas simples para delimitar la cadena:

```
mensaje = 'Hola, "mundo".\n¿Cómo estás?'
print(mensaje)

mensaje = "Hola, \tmundo." # \t inserta una tabulación
print(mensaje)
```

1.2.3 Listas (list)

Las listas son colecciones **ordenadas**, **polimórficas** y **mutables** de elementos.

Ordenadas Los elementos de una lista tienen un orden definido, dado por su posición.
Polimórficas Las listas pueden contener elementos de diferentes tipos, como enteros, cadenas, flotantes, etc.
Mutables Los elementos de una lista pueden ser modificados, añadidos o eliminados después de su creación.

1.2.3.1 Operaciones básicas

```
numeros = [1, 2, 3, 4] # Lista de números
numeros.append(5) # Añade un elemento al final
print(numeros) # [1, 2, 3, 4, 5]

numeros = [1, 2, 3, 4] # Lista de números
print(numeros[0]) # 1

numeros = [1, 2, 3, 4] # Lista de números
print(numeros[1:3]) # [2, 3]

numeros = [1, 2, 3, 4] # Lista de números
numeros.remove(3) # Elimina el primer elemento que coincida con el valor
print(numeros) # [1, 2, 4]

mezcla = [1, "dos", 3.0, True] # Lista con diferentes tipos de datos
mezcla[0] = "uno" # Modifica el primer elemento
print(mezcla) # ['uno', 'dos', 3.0, True]

numeros = [1, 2, 3, 4] # Lista de números
mezcla = [1, "dos", 3.0, True] # Lista con diferentes tipos de datos
mezcla = mezcla + numeros # Concatenación de listas
print(mezcla) # [1, 'dos', 3.0, True, 1, 2, 3, 4]

numeros = [1, 2, 3, 4] # Lista de números
numeros = numeros * 2 # Repite la lista
print(numeros) # [1, 2, 3, 4, 1, 2, 3, 4]
```

La lista vacía se puede definir con corchetes vacíos [] o con la función list():

```
lista_vacia = [] # Lista vacía
print(lista_vacia) # []

lista_vacia2 = list() # Otra forma de crear una lista vacía
print(lista_vacia2) # []
```

1.2.3.2 Iteración sobre listas

```
numeros = [1, 2, 3, 4] # Lista de números
for n in numeros:
    print(n) # 1, 2, 3, 4
```

1.2.4 4. Tuplas (tuple)

Son similares a las listas, **ordenadas** y **polimórficas**, pero **inmutables**, es decir, una vez creada no se puede modificar.

```
coordenadas = (10.0, 20.5, 1)
print(type(coordenadas)) # <class 'tuple'>
```

Se definen con paréntesis y pueden contener diferentes tipos de datos, mientras que las listas se definen con corchetes.

1.2.4.1 Acceso

```
coordenadas = (10.0, 20.5, 1)
print(coordenadas[0]) # 10.0

coordenadas = (10.0, 20.5, 1)
print(coordenadas[-1]) # 1

coordenadas = (10.0, 20.5, 1)
print(coordenadas[1:]) # (20.5, 1)
```

1.2.4.2 Ventajas

- Más livianas que las listas.
- Se pueden usar como claves en diccionarios y se pueden empaquetar varios valores en una sola variable, lo que permite que las funciones puedan devolver múltiples valores, o usar tuplas como claves en diccionarios, entre otros usos.

La forma de empaquetar y desempaquetar tuplas es similar a las listas:

```
coordenadas = (10.0, 20.5, 1)
a, b, c = coordenadas # Desempaquetado
print("a = ", a) # 10.0
print("b = ", b) # 20.5
print("c = ", c) # 1

a = 10.0
b = 20.5
c = 1
tupla2 = (a, b, c) # Empaquetado
print(tupla2) # (10.0, 20.5, 1)
```

La tupla vacía se puede definir con paréntesis vacíos () o con la función tuple():

```
tupla_vacia = () # Tupla vacía
print(tupla_vacia) # ()

tupla_vacia2 = tuple() # Otra forma de crear una tupla vacía
print(tupla_vacia2) # ()
```

1.2.4.3 Anidamiento

Tanto las listas como las tuplas se pueden anidar, es decir, se pueden incluir dentro de otras listas o tuplas.

```
tupla_anidada = (1, 2, (3, 4), [5, 6])
for elemento in tupla_anidada:
    print(elemento) # 1, 2, (3, 4), [5, 6]

tupla_anidada = (1, 2, (3, 4), [5, 6])
print(tupla_anidada[3][0]) # Accede al primer elemento de la lista anidada: 5

tupla_anidada = (1, 2, (3, 4), [5, 6])
tupla_anidada[3].append(7) # Modifica la lista anidada
print(tupla_anidada) # (1, 2, (3, 4), [5, 6, 7])
```

La tupla no se modificó, sigue teniendo 4 elementos, pero la lista que está adentro de la tupla si se puede modificar.

Las tuplas se pueden iterar de la misma manera que las listas.

1.2.5 5. Diccionarios (dict)

Almacenan pares clave-valor. Las claves deben ser únicas e inmutables (por ejemplo, strings, números o tuplas).

```
d = dict() # Crear un diccionario vacío
d["clave1"] = "valor1" # Añadir un par clave-valor
d[25] = "valor2"
d[(1, 2)] = "valor3" # Añadir una clave de tipo tupla
print(d) # {'clave1': 'valor1', 25: 'valor2', (1, 2): 'valor3'}
```

También se pueden crear diccionarios, de forma explícita, usando llaves {}:

```
persona = {"nombre": "Ana", "edad": 30}
print(persona) # {'nombre': 'Ana', 'edad': 30}
```

1.2.5.1 Acceso y modificación

Los diccionarios permiten acceder a los valores mediante sus claves. También se pueden modificar, añadir o eliminar pares clave-valor. La sintaxis es similar a las listas o tuplas, pero en lugar de índices, se utilizan claves.

```
persona = {"nombre": "Ana", "edad": 30}
print(persona["nombre"]) # Ana

persona = {"nombre": "Ana", "edad": 30}
persona["edad"] = 31 # Modifica el valor asociado a la clave "edad"
print(persona) # {'nombre': 'Ana', 'edad': 31}

persona = {"nombre": "Ana", "edad": 30}
persona["email"] = "ana@mail.com" # Añade una nueva clave-valor
print(persona) # {'nombre': 'Ana', 'edad': 31, 'email': 'ana@mail.com'}

persona = {"nombre": "Ana", "edad": 30, "email": "ana@mail.com"}
del persona["edad"] # Elimina la clave "edad"
print(persona) # {'nombre': 'Ana', 'email': 'ana@mail.com'}
```

1.2.5.2 Métodos útiles

```
persona = {"nombre": "Ana", "email": "ana@mail.com"}
# Devuelve una lista con las claves del diccionario
print(persona.keys()) # dict_keys(['nombre', 'email'])

persona = {"nombre": "Ana", "email": "ana@mail.com"}
print(persona.values()) # Devuelve una lista con los valores del diccionario

persona = {"nombre": "Ana", "email": "ana@mail.com"}
# Devuelve una lista de tuplas con los pares clave-valor
print(persona.items()) # dict_items([('nombre', 'Ana'), ('email', 'ana@mail.com')])
```

Un método muy útil es `get()`, que permite acceder a un valor sin generar un error si la clave no existe:

```
persona = {"nombre": "Ana", "email": "ana@mail.com"}
print(persona.get("nombre", "No encontrado")) # Devuelve "Ana"
print(persona.get("edad", "No encontrado")) # Devuelve "No encontrado"
```

`setdefault()` es otro método que permite acceder a un valor y, si la clave no existe, añadirla con un valor por defecto:

```
persona = {"nombre": "Ana", "email": "ana@mail.com"}
# Devuelve 30 y añade la clave "edad" con valor 30
print(persona.setdefault("edad", 30))
print(persona) # {'nombre': 'Ana', 'email': 'ana@mail.com', 'edad': 30}

persona = {"nombre": "Ana", "edad": 30, "email": "ana@mail.com"}
# Añade la clave "telefonos" con una lista vacía
lista = persona.setdefault("telefonos", [])
lista.append("123-456-7890") # Añade un teléfono a la lista
print(persona)
```

1.2.5.3 Iteración sobre diccionarios

Los diccionarios se pueden iterar para acceder a las claves y valores.

```
for clave, valor in persona.items(): # Itera sobre los pares clave-valor
    print(f"{clave}: {valor}") # nombre: Ana, email: ana@mail.com, edad: 30
```

1.2.6 Conjuntos (set)

Los conjuntos son colecciones **no ordenadas** de elementos únicos. No permiten duplicados y no tienen un índice asociado a sus elementos.

```
conjunto = {1, 2, 3, 4, 5}
print(conjunto) # {1, 2, 3, 4, 5}
```

El conjunto vacío se puede definir con la función `set()`:

```
conjunto_vacio = set() # Conjunto vacío
print(conjunto_vacio) # set()

conjunto_vacio2 = {} # Esto crea un diccionario vacío, no un conjunto
print(conjunto_vacio2) # {}
```

Para agregar un elemento a un conjunto, se utiliza el método `add()`:

```
conjunto = {1, 2, 3, 4, 5}
conjunto.add(6) # Añade el elemento 6 al conjunto
print(conjunto) # {1, 2, 3, 4, 5, 6}
```

Si intentamos agregar un elemento que ya existe, no se producirá un error, pero el conjunto no cambiará:

```
conjunto = {1, 2, 3, 4, 5, 6}
conjunto.add(6) # No se añade, ya que 6 ya está en el conjunto
print(conjunto) # {1, 2, 3, 4, 5, 6}
```

Se puede crear un conjunto a partir de una lista o tupla usando la función `set()`:

```
lista = [1, 2, 3, 4, 5, 5]
conjunto_desde_lista = set(lista) # Crea un conjunto a partir de una lista
print(conjunto_desde_lista) # Elimina duplicados automáticamente
```

Para eliminar un elemento de un conjunto, se utiliza el método `remove()` o `discard()`. La diferencia es que `remove()` genera un error si el elemento no existe, mientras que `discard()` no lo hace:

```
conjunto = {1, 2, 3, 4, 5, 6}
conjunto.remove(7) # Genera un error, ya que 7 no está en el conjunto
print(conjunto)

conjunto = {1, 2, 3, 4, 5, 6}
conjunto.discard(7) # No genera error
print(conjunto) # {1, 2, 3, 4, 5, 6}
```

el operador `in` se puede usar para verificar si un elemento está en un conjunto:

```
conjunto = {1, 2, 3, 4, 5, 6}
print(3 in conjunto) # True
```

No se puede acceder a los elementos de un conjunto por índice, ya que no están ordenados. Sin embargo, se pueden iterar:

```
conjunto = {1, 2, 3, 4, 5, 6}
for elemento in conjunto:
    print(elemento) # 1, 2, 3, 4, 5, 6
```

Los conjuntos son útiles para realizar operaciones matemáticas como unión, intersección, diferencia y diferencia simétrica.

1.2.6.1 Operaciones con conjuntos

```
# Unión
conjunto1 = {1, 2, 3, 4, 5, 6}
conjunto2 = {4, 5, 6, 7}
print(conjunto1, " union ", conjunto2, "=", conjunto1 | conjunto2)
# {1, 2, 3, 4, 5, 6, 7}

# Intersección
conjunto1 = {1, 2, 3, 4, 5, 6}
conjunto2 = {4, 5, 6, 7}
print(conjunto1, " interseccion ", conjunto2, "=", conjunto1 & conjunto2)
# {4, 5, 6}

# Diferencia
conjunto1 = {1, 2, 3, 4, 5, 6}
```

```

conjunto2 = {4, 5, 6, 7}
print(conjunto1, " diferencia ", conjunto2, "=", conjunto1 - conjunto2)
# {1, 2, 3}
print(conjunto2, " diferencia ", conjunto1, "=", conjunto2 - conjunto1)
# {7}

# Diferencia simétrica
conjunto1 = {1, 2, 3, 4, 5, 6}
conjunto2 = {4, 5, 6, 7}
print(f"{conjunto1} diferencia simetrica {conjunto2} = "
      f"{conjunto1 ^ conjunto2}")
# {1, 2, 3, 7}
print(f"{conjunto2} diferencia simetrica {conjunto1} = "
      f"{conjunto2 ^ conjunto1}")
# {1, 2, 3, 7}

conjunto = {1, 2, 3, 4, 5, 6}
# Subconjunto
es_subconjunto = {1, 2} <= conjunto
print("{1, 2} es subconjunto de", conjunto, "?: ", es_subconjunto)
# True

conjunto = {1, 2, 3, 4, 5, 6}
# Superconjunto
es_superconjunto = {1, 2} >= conjunto
print("{1, 2} es superconjunto de", conjunto, "?: ", es_superconjunto)
# False

```

Existen otros tipos de conjuntos que permiten almacenar elementos únicos, pero que una vez creados no se pueden modificar, se llaman **conjuntos inmutables** o **frozensets**. Se crean usando la función `frozenset()`:

```

conjunto_inmutable = frozenset([1, 2, 3, 4, 5])
print(conjunto_inmutable)
# frozenset({1, 2, 3, 4, 5})

```

Los conjuntos inmutables son útiles cuando se necesita un conjunto que no cambie a lo largo del tiempo, por ejemplo, como claves en un diccionario o elementos en otro conjunto.

Los métodos de los conjuntos inmutables son limitados, ya que no se pueden modificar. Por ejemplo, no se pueden usar `add()` o `remove()`, pero sí se pueden usar operaciones como unión, intersección y diferencia.

```

conjunto_inmutable = frozenset([1, 2, 3, 4, 5])
conjunto2 = frozenset([4, 5, 6, 7])
print(conjunto_inmutable | conjunto2) # Unión
print(conjunto_inmutable & conjunto2) # Intersección
print(conjunto_inmutable - conjunto2) # Diferencia
print(conjunto2 - conjunto_inmutable) # Diferencia
print(conjunto_inmutable ^ conjunto2) # Diferencia simétrica

```

El resultado de estas operaciones es un nuevo conjunto inmutable, ya que el conjunto original no se modifica.

1.2.7 Generación de colecciones de datos por comprensión (*comprehension*)

Las comprensiones de listas, tuplas y diccionarios son una forma concisa de crear colecciones en Python. Permiten aplicar una expresión a cada elemento de una colección existente, filtrando o transformando los elementos según sea necesario.

```

numeros = [x for x in range(1, 6)] # Lista de números del 1 al 5
cuadrados = [x**2 for x in numeros]
print(cuadrados) # [1, 4, 9, 16, 25]

```

Si en lugar de corchetes `[]` se usan paréntesis `()`, se crea un generador en lugar de una lista:

```
numeros = (x for x in range(1, 6)) # Tupla de números del 1 al 5
cuadrados = (x**2 for x in numeros)
print(cuadrados) # (1, 4, 9, 16, 25)
```

Si se usan llaves {}, se crea un conjunto o un diccionario, dependiendo de si se especifica una clave o un par clave-valor:

```
numeros = {x for x in range(1, 6)} # Conjunto de números del 1 al 5
cuadrados = {x**2 for x in numeros}
print(cuadrados) # {1, 4, 9, 16, 25}

numeros = {"uno": 1, "dos": 2, "tres": 3}
cuadrados = {clave: valor**2 for clave, valor in numeros.items()}
print(cuadrados) # {'uno': 1, 'dos': 4, 'tres': 9}

diccionario = {"nombre": "Ana", "edad": 30, "email": "ana@example.com"}
tuplas = [(clave, valor) for clave, valor in diccionario.items()]
print(tuplas) # [('nombre', 'Ana'), ('edad', 30), ('email', 'ana@example.com')]
```

La comprensión es una característica funcional de Python muy poderosa.

1.2.8 Recursos para profundizar

- Tutorial de Python - Estructuras de datos
- `setdefault()` (KeepCoding)
- Comprensiones de listas (Python Docs)
- Comprensión de listas (Hektor Profe)
- Operadores Encadenados (Hektor Profe)
- Intérprete Python, para entender como funciona

1.3 Estructuras de control en Python

1.3.1 Condicionales

1.3.1.1 Sintaxis if-elif-else

```
# Sintaxis completa
edad = 25
salario = 45000

if edad < 18:
    categoria = "menor"
elif edad < 65 and salario > 30000:
    categoria = "adulto solvente"
elif edad < 65:
    categoria = "adulto"
else:
    categoria = "jubilado"

print(f"Categoría: {categoria}") # adulto solvente
```

1.3.1.2 Operadores lógicos y comparación

```
x, y, z = 5, 10, 15

if x < y and y < z: # and = && en Java/Go
    print("Orden ascendente")

if x == 5 or y == 5: # or = || en Java/Go
    print("Alguno es 5")

if not (x > y): # not = ! en Java/Go
    print("x no es mayor que y")

# Comparaciones encadenadas (única de Python)
if x < y < z:
    print("Orden ascendente (sintaxis pythónica)")
```

1.3.1.3 Expresión condicional (operador ternario)

```
# Equivalente al operador ?: de Java/Go
numero = 7
resultado = "par" if numero % 2 == 0 else "impar"
print(f"El número {numero} es {resultado}")
```

El fragmento anterior es una forma concisa de asignar un valor basado en una condición. Es útil para asignaciones simples y mejora la legibilidad del código. Es equivalente a:

```
if numero % 2 == 0:
    resultado = "par"
else:
    resultado = "impar"
print(f"El número {numero} es {resultado}")
```

1.3.2 Ciclos

1.3.2.1 Ciclo for: iteración sobre secuencias

```
# for-in: itera directamente sobre elementos (no índices)
frutas = ["manzana", "banana", "naranja"]

for fruta in frutas:
    print(fruta)

# Con índices usando enumerate()
frutas = ["manzana", "banana", "naranja"]
```

```
for i, fruta in enumerate(frutas):
    print(f"{i}: {fruta}")
```

La función `enumerate()` es útil para obtener tanto el índice como el valor del elemento en una lista.

```
help(enumerate)
```

```
# Equivalente a for(int i=0; i<frutas.length; i++) en Java
frutas = ["manzana", "banana", "naranja"]
```

```
for i in range(len(frutas)):
    print(f"{i}: {frutas[i]}")
```

1.3.2.2 Función `range()` para ciclos numéricos

La función `range()` genera una secuencia de números, útil para ciclos `for`. Es como si generara una lista de números, pero de forma más eficiente.

La sintaxis de `range()` es:

```
range(start, stop[, step])

# range(stop)
for i in range(5): # 0, 1, 2, 3, 4
    print(i)

# range(start, stop)
for i in range(1, 5): # 1, 2, 3, 4
    print(i)

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

# Decremento
for i in range(10, 0, -1): # 10, 9, 8, ..., 1
    print(i)
```

1.3.2.3 Ciclo `while`

```
# Sintaxis similar a otros lenguajes
contador = 0
while contador < 5:
    print(f"Contador: {contador}")
    contador += 1

# Ciclo infinito con break
while True:
    respuesta = input("¿Continuar? (s/n): ")
    if respuesta.lower() != "s":
        break
```

1.3.2.4 `break`, `continue` y `else` en ciclos

```
# break y continue funcionan igual que en otros lenguajes
for i in range(10):
    if i == 3:
        continue # Salta a la siguiente iteración
    if i == 7:
        break # Sale del ciclo
    print(i) # Imprime: 0, 1, 2, 4, 5, 6

# else en ciclos: ÚNICO DE PYTHON
# Se ejecuta si el ciclo termina normalmente (sin break)
for i in range(5):
    if i == 10: # Nunca se cumple
        break
else:
    print("Ciclo completado sin break") # Se ejecuta
```

```
# Ejemplo práctico: búsqueda
numeros = [1, 3, 5, 7, 9]
objetivo = 6

for num in numeros:
    if num == objetivo:
        print(f"Encontrado: {num}")
        break
else:
    print("No encontrado") # Se ejecuta porque no hubo break
```

1.3.3 Iteración sobre estructuras de datos

1.3.3.1 Diccionarios

```
# Solo claves
datos = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}

for clave in datos:
    print(clave)

# Solo valores
datos = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}

for valor in datos.values():
    print(valor)

# Claves y valores
datos = {"nombre": "Ana", "edad": 25, "ciudad": "Madrid"}

for clave, valor in datos.items():
    print(f"{clave}: {valor}")
```

1.3.3.2 Listas con múltiples variables

```
# Desempaquetado en ciclos
puntos = [(1, 2), (3, 4), (5, 6)]

for x, y in puntos:
    print(f"x={x}, y={y}")

# Con enumerate para índice + desempaquetado
puntos = [(1, 2), (3, 4), (5, 6)]
for i, (x, y) in enumerate(puntos):
    print(f"Punto {i}: ({x}, {y})")
```

1.3.4 No hay switch

Python 3.10+ tiene match-case, pero la siguiente construcción usando if-elif es más común.

```
opcion = "b"

if opcion == "a":
    resultado = "Opción A"
elif opcion == "b":
    resultado = "Opción B"
elif opcion in ["c", "d"]:
    resultado = "Opción C o D"
else:
    resultado = "Opción desconocida"

print(resultado)
```

1.3.5 Ejemplo práctico: procesamiento de datos

```
# Procesamiento típico de datos en Python
empleados = [
    {"nombre": "Ana", "salario": 50000, "departamento": "IT"},
```

```

{"nombre": "Carlos", "salario": 45000, "departamento": "Ventas"},
{"nombre": "María", "salario": 55000, "departamento": "IT"},
{"nombre": "Juan", "salario": 40000, "departamento": "RRHH"},
]

# Filtrar y procesar con sintaxis pythónica
for empleado in empleados:
    nombre = empleado["nombre"]
    salario = empleado["salario"]

    # Determinar categoría y bonus
    if salario >= 50000:
        categoria = "Senior"
        bonus = salario * 0.15
    elif salario >= 45000:
        categoria = "Mid"
        bonus = salario * 0.10
    else:
        categoria = "Junior"
        bonus = salario * 0.05

    print(f"{nombre}: {categoria} - Bonus: ${bonus:,.2f}")

```

1.3.6 Diferencias sintácticas resumidas

Característica	Python	Java/Go
Delimitadores	Indentación	{, }
Operadores lógicos	and, or, not	&&, , !
Ciclo for	for item in collection:	for (type item : collection)
else en ciclos	Si	No
Operador ternario	value_if_true if condition else value_if_false	condition ? value_if_true : value_if_false
switch/match	if - elif o match - case (Python 3.10+)	switch

La sintaxis de Python prioriza la legibilidad y expresividad, usando palabras en inglés en lugar de símbolos cuando es posible.

1.3.7 Recursos para profundizar

- Tutorial de Python - Estructuras de control
- Python for Loops (W3Schools)
- Python While Loops (W3Schools)
- Python Match Statement (W3Schools)

1.4 Ámbitos de ejecución

Este capítulo profundizaremos sobre el manejo de variables en Python, contrastándolo con lo que ya conocemos de Go y Java. Aunque los conceptos fundamentales de variables son universales, Python introduce matices importantes en su gestión, especialmente en lo que respecta a la inmutabilidad de ciertos tipos de datos, los ámbitos de ejecución y la poderosa característica de las clausuras.

1.4.1 Variables y asignación

En Go y Java, la declaración de variables a menudo implica especificar explícitamente el tipo de dato (aunque Go ofrece inferencia de tipos). Python, por otro lado, es un lenguaje de tipado dinámico. Esto significa que no se declara el tipo de una variable; el tipo se infiere en tiempo de ejecución según el valor que se le asigna.

Inferencia de tipos

Es la capacidad de un lenguaje de programación para deducir automáticamente el tipo de una expresión o variable en tiempo de compilación o ejecución, sin que el programador tenga que declararlo explícitamente. Para más información, consulta Wikipedia.

Una diferencia clave es que en Python, las variables son esencialmente referencias a objetos en memoria. Cuando se reasigna una variable, simplemente esa referencia pasa a apuntar a un objeto diferente, en lugar de cambiar el valor (esto es crucial para entender la inmutabilidad de ciertos tipos).

Cada vez que se asigna un valor a una variable, Python sigue los siguientes pasos:

1. Crea un objeto en memoria (si no existe ya).
2. Asigna una referencia a ese objeto.

Python garantiza que los pasos anteriores para asignar una variable son **atómicos**, es decir se ejecutan uno tras otro sin interrupciones, lo que asegura la consistencia del estado de las variables en un entorno multihilo.

Si la variable ya tenía una referencia a otro objeto, esa referencia se pierde (el objeto anterior puede ser desalojado de la memoria por el recolector de basura si no hay otras referencias a él).

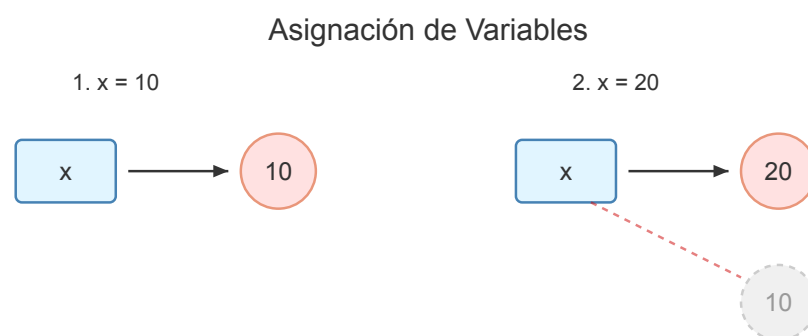


Figure 1.3: Asignación de Variables

Esto contrasta con Go y Java, donde la asignación de una variable puede implicar la creación de una copia del valor (especialmente para tipos primitivos).

1.4.2 Tipos de datos y mutabilidad

En Python **todo es un objeto**, por lo tanto tanto podemos pensar que todas las variables son referencias a objetos en el *heap*. La distinción importante es si un objeto es mutable o inmutable.

1.4.2.1 Tipos inmutables (como los “primitivos” en Java/Go)

- Booleanos (bool)
- Números (int, float, complex)
- Cadenas (str)
- Tuplas (tuple)
- Rangos (range)
- Conjuntos congelados (frozenset)

Cuando a una variable que ya tenía asignado un objeto inmutable, se le asigna otro valor, en realidad se crea un nuevo objeto inmutable en memoria y la referencia anterior se pierde.

```
s1 = "hola"
s2 = s1
s1 += " mundo" # Esto crea una nueva cadena "hola mundo"
               # y s1 ahora referencia a ella
print(f"s1: {s1}, s2: {s2}") # Salida: s1: hola mundo, s2: hola
```

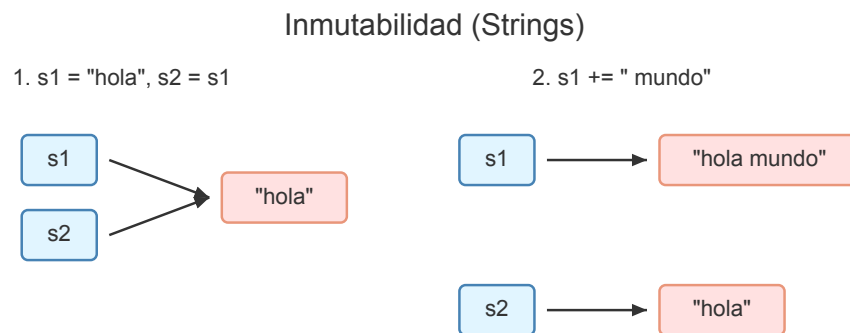


Figure 1.4: Inmutabilidad (Strings)

En este fragmento, s1 y s2 inicialmente referencian al mismo objeto, la cadena "hola". Al modificar s1, se crea un nuevo objeto cadena "hola mundo", y s1 ahora apunta a este nuevo objeto, mientras que s2 sigue apuntando al antiguo objeto "hola".

1.4.2.2 Tipos mutables (como los objetos en Java/Go)

- Listas (list)
- Diccionarios (dict)
- Conjuntos (set)
- Objetos de clases personalizadas

Cuando se modifica un objeto mutable, se altera el objeto en su lugar. Si múltiples variables referencian al mismo objeto mutable, todas verán los cambios.

```
lista1 = [1, 2, 3]
lista2 = lista1 # lista1 y lista2 referencian a la misma lista
lista1.append(4) # Modifica la lista original
print(f"lista1: {lista1}, lista2: {lista2}")
```

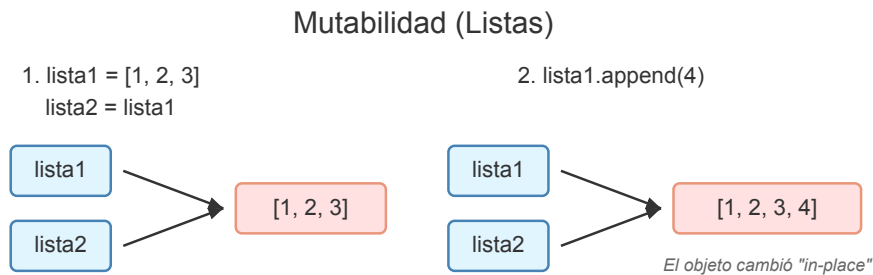


Figure 1.5: Mutabilidad (Listas)

En este caso, `lista1` y `lista2` referencian al mismo objeto lista. Al modificar `lista1`, `lista2` refleja el cambio porque ambas variables apuntan al mismo objeto en memoria.

1.4.3 Visibilidad de variables

En Python no existe el concepto de público, privado o protegido como en Java. En cambio, se utiliza una convención de nomenclatura para indicar la visibilidad de las variables:

Variables públicas Se definen sin guiones bajos al inicio del nombre. Son accesibles desde cualquier parte del código (ej. `mi_variable`).

Variables protegidas Se definen con un guion bajo al inicio del nombre (ej. `_variable`). Indica que la variable es para uso interno del módulo o clase, pero aún es accesible desde fuera.

Variables privadas Se definen con dos guiones bajos al inicio del nombre (ej. `__variable`). Esto activa el *name mangling*, lo que significa que el nombre de la variable se modifica internamente para evitar conflictos con nombres en subclases.

Variables especiales Se definen con dos guiones bajos al inicio y al final del nombre, estos son conocidos en la comunidad Python como *dunder methods* (ej. `__init__`). Estas son utilizadas por Python para definir métodos especiales y no deben ser modificadas directamente.

Advertencia

Todas las variables en Python son accesibles desde fuera del módulo o clase, incluso las privadas. La convención de nomenclatura es solo una guía para los desarrolladores y no impide el acceso a las variables.

1.4.4 Ámbitos de ejecución (*scopes*): La regla LEGB

Python define un sistema de ámbitos para resolver nombres (variables, funciones, clases, etc.). Este sistema se conoce comúnmente como la regla LEGB:

Local (L) Nombres definidos dentro de una función.

Enclosing (E) / Clausura Nombres en el ámbito de una función externa (función “envolvente”). Este ámbito define el contexto de ejecución de una función anidada.

Global (G) Nombres definidos en el nivel superior de un módulo (archivo `.py`).

Built-in (B) Nombres preasignados por Python (ej. `open`, `range`, `print`).

Cuando Python busca un nombre, sigue este orden: primero busca en el ámbito *Local*, luego en el *Enclosing*, después en el *Global* y finalmente en el *Built-in*.

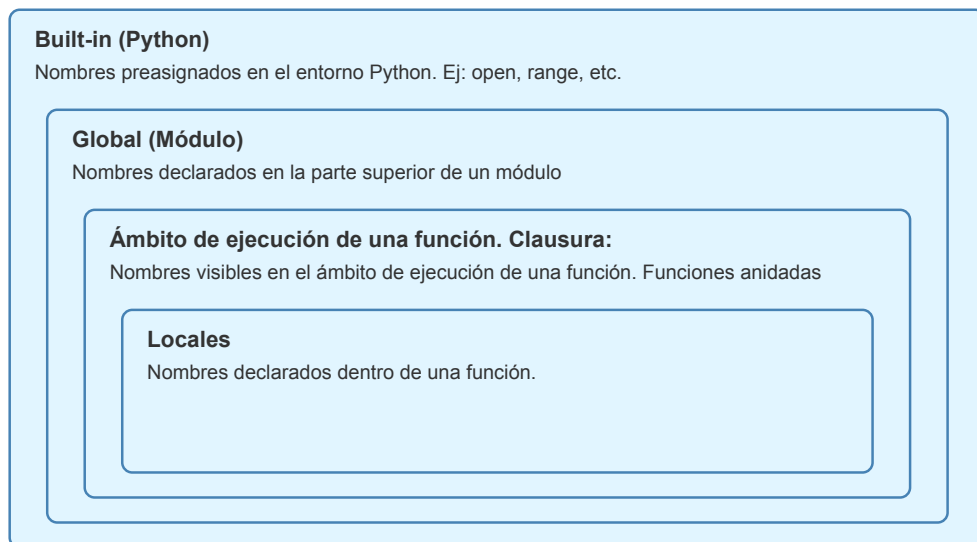


Figure 1.6: Ámbitos de Ejecución

1.4.4.1 Ámbito *Local* (L)

Las variables definidas dentro de una función son locales a esa función. Esto significa que solo son accesibles dentro de la función y no pueden ser accedidas desde fuera de ella. Una vez que la función termina su ejecución, las variables locales se eliminan de la memoria.

Si hay variables globales definidas con el mismo nombre de las variables locales, entonces las locales **ocultan** las globales. Esto se conoce como **Ocultamiento** de variables (*shadowing*).

```
mensaje = "Hola desde el ámbito global" # Variable global
```

```
def mi_funcion():  
    mensaje = "Hola desde la función" # Variable local  
    print(mensaje)
```

```
mi_funcion() # Llama a la función que imprime el mensaje local  
print(mensaje) # Acceso a la variable global
```

La variable local mensaje dentro de mi_funcion oculta la variable global del mismo nombre. Cuando se llama a mi_funcion, imprime el mensaje local, mientras que fuera de la función se accede a la variable global.

Si se necesita modificar una variable global desde dentro de una función, se debe usar la palabra clave `global` para indicar que se quiere referenciar a la variable global.

```
mensaje = "Hola desde el ámbito global" # Variable global
```

```
def mi_funcion():  
    global mensaje # Indica que se quiere usar la variable global  
    mensaje = "Hola desde la función" # Modifica la variable global  
    print(mensaje)
```

```
mi_funcion() # Llama a la función que modifica el mensaje global  
print(mensaje) # Acceso a la variable global modificada
```

Advertencia

El uso excesivo de global o de ocultamiento de variables puede llevar a código difícil de mantener y depurar. Preferiblemente, se deben pasar las variables como argumentos a las funciones y las funciones deben devolver valores explícitos.

1.4.4.2 Ámbito *Enclosing* (E) / Clausuras

En Python, las funciones son ciudadanos de primera clase, lo que significa que Python las trata como a un objeto más y por lo tanto se pueden asignar funciones a variables, pasarlas como argumentos y retornarlas desde otras funciones y también se pueden **anidar**, esto es definir funciones dentro de otras funciones.

Esta versatilidad de poder definir una función dentro de otra lleva a las clausuras, que son una característica poderosa y distintiva de Python. Las clausuras permiten que una función anidada acceda a variables del ámbito de su función envolvente, incluso después de que la función envolvente haya terminado su ejecución.

Para que una función sea una clausura, debe cumplir dos condiciones:

- Debe ser una función anidada (una función definida dentro de otra función).
- Debe referenciar variables de su ámbito externo (no global, no local a ella misma). Estas variables se conocen como **referencias externas**.

```
def fabrica_incrementos(y):  
    def incrementar(x):  
        return x + y # y está encapsulada en la función interna  
  
    return incrementar # Retorna la función interna
```

```
incrementar_2 = fabrica_incrementos(2) # Crea una función que incrementa en 2  
print(incrementar_2(5)) # Salida: 7
```

Al ejecutar el fragmento anterior ocurre lo siguiente:

1. En la línea 1 se define la función `fabrica_incrementos` que recibe un parámetro `y`. El código de la función (hasta la línea 4) se guarda en memoria. Es un valor más. El nombre de la función `fabrica_incrementos` se guarda en el ámbito global y es la referencia que permite acceder al objeto función.
2. En la línea 6 se llama a `fabrica_incrementos(2)` y el resultado de esa operación (la función interna `incrementar`) se va a asignar a la variable `incrementar_2`. En este momento, `y` tiene el valor 2 y se guarda en la clausura de la función interna `incrementar`.
3. El valor devuelto por `fabrica_incrementos` es una función que queda ligada a la variable `incrementar_2`. `incrementar_2` contiene el valor de `y`, al momento de su creación, en su clausura. Esto significa que `incrementar_2` “recuerda” el valor de `y` aunque `fabrica_incrementos` ya haya terminado su ejecución.
4. En la línea 7, se ejecuta `incrementar_2`. `incrementar_2` toma un parámetro `x` y retorna la suma de `x` más `y`. Si bien `fabrica_incrementos` ya ha terminado su ejecución y por lo tanto los valores de sus parámetros no están en la memoria, la referencia a `y` se mantiene en la clausura. La función realiza la operación `5 + 2`, donde 5 es el valor ligado al parámetro `x` y 2 es el valor de `y`, al momento de la creación de `incrementar_2` que se guardó en la clausura.
5. `incrementar_2(5)` retorna 7 al ámbito global, y `print` lo muestra en la salida.

Advertencia

Para Python todas las variables son referencias, incluido los nombres de las funciones. Al colocar paréntesis luego del nombre de la misma, se invoca la función y se ejecuta el código que contiene. Si no se colocan paréntesis, se obtiene una referencia a la función, que es un objeto más en memoria.

1.4.4.3 Ámbito *Global* (G)

El ámbito global se refiere a las variables definidas en el nivel superior de un módulo. Estas variables son accesibles desde cualquier parte del módulo, incluidas las funciones.

Al declarar un módulo se puede incluir variables y constantes globales que pueden ser utilizadas en todo el código del módulo. A modo de ejemplo podemos ver las constantes matemáticas definidas en el módulo `math`, como `math.pi` o `math.e`.

```
import math # Importa el módulo math

print("Constantes matemáticas:")
print(math.pi) # Imprime el valor de pi
print(math.e) # Imprime el valor de e
print(math.tau) # Imprime el valor de tau
print(math.inf) # Imprime el valor de infinito
print(math.nan) # Imprime el valor de NaN (Not a Number)
```

Para definir un módulo propio se crea un archivo con extensión `.py` y se pueden definir variables y funciones que serán accesibles desde otros módulos al importarlos. El nombre del módulo es el nombre del archivo sin la extensión `.py`.

A modo de ejemplo, se muestra un módulo simple que implementa una pila (*stack*) utilizando una lista y Objetos. Más adelante veremos en detalle la Programación Orientada a Objetos (POO) en Python, pero aquí se muestra un ejemplo de un módulo y como se documenta cada parte del código.

Click para ver el código

```
"""
Módulo Stack - Implementación de Pila (LIFO)

Este módulo proporciona una implementación de la estructura de datos pila
(stack) utilizando el principio LIFO (Last In, First Out - Último en entrar, primero
en salir).

La pila es una estructura de datos lineal que permite insertar y eliminar
elementos únicamente desde un extremo llamado "cima" o "tope".

Ejemplo de uso:
    >>> from stack import Stack
    >>> pila = Stack()
    >>> pila.push(10)
    >>> pila.push(20)
    >>> print(pila.peek()) # 20
    >>> print(pila.pop()) # 20
    >>> print(pila.size()) # 1

Clases:
    Stack: Implementación de pila usando lista interna de Python.
"""
```

Ejemplo de uso:

```
>>> from stack import Stack
>>> pila = Stack()
>>> pila.push(10)
>>> pila.push(20)
>>> print(pila.peek()) # 20
>>> print(pila.pop()) # 20
>>> print(pila.size()) # 1
```

Clases:

```
Stack: Implementación de pila usando lista interna de Python.
```

```
"""
```

```
from stack_exception import StackException
```

```
class Stack:
```

```
    """
```

```
    Clase Stack - Implementación de una pila (stack) usando lista interna.
```

```
    Una pila es una estructura de datos que sigue el principio LIFO
    (Last In, First Out). Los elementos se agregan y eliminan desde
    la misma posición llamada "cima" o "tope".
```

```
    Atributos:
```

```
_items (list): Lista interna que almacena los elementos de la pila.  
Se usa convención de nombre privado con underscore.
```

Operaciones principales:

- push: Agregar elemento a la cima
- pop: Eliminar y retornar elemento de la cima
- peek/top: Ver elemento de la cima sin eliminarlo
- is_empty: Verificar si está vacía
- size: Obtener número de elementos

Complejidad temporal:

- Todas las operaciones son $O(1)$ - tiempo constante

Ejemplo:

```
>>> stack = Stack()  
>>> stack.push(1)  
>>> stack.push(2)  
>>> stack.push(3)  
>>> print(stack.pop()) # 3  
>>> print(stack.peek()) # 2  
"""
```

```
def __init__(self):
```

```
    """
```

Inicializa una pila vacía.

Crea una nueva instancia de Stack con una lista interna vacía para almacenar los elementos.

Complejidad temporal: $O(1)$

Complejidad espacial: $O(1)$

Ejemplo:

```
>>> stack = Stack()  
>>> print(stack.is_empty()) # True  
>>> print(stack.size()) # 0  
"""
```

```
self._items = []
```

```
def push(self, item):
```

```
    """
```

Agrega un elemento a la cima de la pila.

Inserta el elemento dado en la posición superior de la pila. Esta operación siempre es exitosa y no tiene restricciones de capacidad (limitada solo por la memoria disponible).

Args:

item: El elemento a agregar a la pila. Puede ser de cualquier tipo.

Returns:

None

Complejidad temporal: $O(1)$

Complejidad espacial: $O(1)$

Ejemplo:

```
>>> stack = Stack()  
>>> stack.push(10)  
>>> stack.push("texto")  
>>> stack.push([1, 2, 3])  
>>> print(stack.size()) # 3  
"""
```

```
self._items.append(item)
```

```

def pop(self):
    """
    Elimina y retorna el elemento de la cima de la pila.

    Remueve el elemento que está en la posición superior de la pila
    y lo retorna. La pila se reduce en un elemento.

    Returns:
        El elemento que estaba en la cima de la pila.

    Raises:
        StackException: Si se intenta hacer pop en una pila vacía.

    Complejidad temporal: O(1)
    Complejidad espacial: O(1)

    Ejemplo:
        >>> stack = Stack()
        >>> stack.push(10)
        >>> stack.push(20)
        >>> elemento = stack.pop()
        >>> print(elemento)      # 20
        >>> print(stack.size()) # 1

        # Caso de error:
        >>> stack_vacia = Stack()
        >>> stack_vacia.pop()    # Lanza StackException
    """
    if self.is_empty():
        raise StackException("pop from empty stack")
    return self._items.pop()

def peek(self):
    """
    Retorna el elemento de la cima sin eliminarlo.

    Permite ver el elemento que está en la posición superior
    de la pila sin modificar la estructura. También conocido
    como operación 'top'.

    Returns:
        El elemento que está en la cima de la pila.

    Raises:
        StackException: Si se intenta hacer peek en una pila vacía.

    Complejidad temporal: O(1)
    Complejidad espacial: O(1)

    Ejemplo:
        >>> stack = Stack()
        >>> stack.push(10)
        >>> stack.push(20)
        >>> elemento = stack.peek()
        >>> print(elemento)      # 20
        >>> print(stack.size()) # 2 (no se eliminó)

        # Caso de error:
        >>> stack_vacia = Stack()
        >>> stack_vacia.peek()    # Lanza StackException
    """
    if self.is_empty():
        raise StackException("peek from empty stack")
    return self._items[-1]

def is_empty(self):

```

```

"""
Verifica si la pila está vacía.

Determina si la pila no contiene ningún elemento.
Es útil para validar antes de realizar operaciones
que requieren elementos (como pop o peek).

Returns:
    bool: True si la pila está vacía, False en caso contrario.

Complejidad temporal: O(1)
Complejidad espacial: O(1)

```

```

Ejemplo:
>>> stack = Stack()
>>> print(stack.is_empty()) # True
>>> stack.push(10)
>>> print(stack.is_empty()) # False
>>> stack.pop()
>>> print(stack.is_empty()) # True
"""
return len(self._items) == 0

```

```

def size(self):
    """
    Retorna el número de elementos en la pila.

    Obtiene la cantidad actual de elementos almacenados
    en la pila. Es útil para conocer el estado de la
    estructura de datos.

    Returns:
        int: Número de elementos en la pila (>= 0).

    Complejidad temporal: O(1)
    Complejidad espacial: O(1)

```

```

Ejemplo:
>>> stack = Stack()
>>> print(stack.size()) # 0
>>> stack.push(10)
>>> stack.push(20)
>>> print(stack.size()) # 2
>>> stack.pop()
>>> print(stack.size()) # 1
"""
return len(self._items)

```

```

def __str__(self):
    """
    Representación en string de la pila.

    Proporciona una representación legible de la pila
    mostrando los elementos desde la base hasta la cima.

    Returns:
        str: Representación en cadena de la pila.

```

```

Ejemplo:
>>> stack = Stack()
>>> stack.push(1)
>>> stack.push(2)
>>> stack.push(3)
>>> print(stack) # Stack: [1, 2, 3] (cima: 3)
"""
if self.is_empty():

```

```

        return "Stack: [] (vacía)"
        return f"Stack: {self._items} (cima: {self._items[-1]})"

def __repr__(self):
    """
    Representación técnica de la pila.

    Proporciona una representación que puede usarse
    para recrear el objeto.

    Returns:
        str: Representación técnica del objeto.
    """
    return f"Stack({self._items})"

def demo_stack():
    """
    Función de demostración del uso de la clase Stack.

    Muestra ejemplos de todas las operaciones disponibles
    y casos de uso típicos de una pila.
    """
    print("=== Demostración de la clase Stack ===\n")

    # Crear pila vacía
    stack = Stack()
    print(f"Pila creada: {stack}")
    print(f"¿Está vacía?: {stack.is_empty()}")
    print(f"Tamaño: {stack.size()}\n")

    # Agregar elementos
    print("Agregando elementos (push):")
    elementos = [10, 20, 30, "Python", [1, 2, 3]]
    for elemento in elementos:
        stack.push(elemento)
        print(f"  Push {elemento} -> {stack}")

    print(f"\nTamaño final: {stack.size()}")
    print(f"Elemento en la cima (peek): {stack.peek()}\n")

    # Eliminar elementos
    print("Eliminando elementos (pop):")
    while not stack.is_empty():
        elemento = stack.pop()
        print(f"  Pop -> {elemento}, pila restante: {stack}")

    print(f"\n¿Está vacía?: {stack.is_empty()}")

    # Demostrar manejo de errores
    print("\n=== Manejo de errores ===")
    try:
        stack.pop()
    except StackException as e:
        print(f"Error al hacer pop en pila vacía: {e}")

    try:
        stack.peek()
    except StackException as e:
        print(f"Error al hacer peek en pila vacía: {e}")

if __name__ == "__main__":
    demo_stack()

```

A partir de la función `demo_stack`, se muestra cómo se puede utilizar el módulo `stack` para crear una pila, agregar elementos y eliminarlos.

Es común que los módulos tengan un bloque de código al final que se ejecuta solo si el módulo se ejecuta directamente, no cuando se importa. Esto se logra utilizando la siguiente estructura:

```
if __name__ == "__main__":  
    demo_stack()
```

Si el módulo se importa desde otro módulo, el bloque `if __name__ == "__main__":` no se ejecuta, lo que permite que el código de demostración no interfiera con el uso del módulo como biblioteca.

1.4.4.4 Ámbito *Built-in* (B)

El ámbito *built-in* contiene nombres predefinidos por Python, como funciones y excepciones que están disponibles en todos los módulos sin necesidad de importarlos. Estos nombres son parte del núcleo del lenguaje y se pueden utilizar directamente en cualquier parte del código. Algunos ejemplos son `print`, `len`, `range`, `int`, `str`, entre otros.

Si se intenta redefinir un nombre *built-in*, se creará una variable local o global que ocultará temporalmente el nombre *built-in*, pero no se eliminará del ámbito *built-in*.

Peligro

No se recomienda bajo ningún punto de vista, redefinir nombres *built-in*, ya que esto puede causar confusión y errores difíciles de depurar. Es mejor utilizar nombres descriptivos y evitar conflictos con los nombres predefinidos de Python.

```
print(len("Hola")) # Llama a la función built-in len
```

```
def mi_funcion():  
    len = 4  
    print(len("Mundo")) # Error
```

```
mi_funcion() # Llama a la función que imprime la longitud de "Mundo"
```

1.4.5 Recursos para profundizar

- Tutorial de Python - Ámbitos de ejecución
- Python Scopes and Namespaces (W3Schools)
- Python Built-in Functions (W3Schools)

1.5 Funciones y paradigma funcional

En Python las funciones son ciudadanos de primera clase, lo que le otorga al lenguaje ciertas características del paradigma funcional. En este capítulo, exploraremos cómo definir y utilizar funciones, los diferentes tipos de pasajes de parámetros, y cómo Python implementa el paradigma funcional a través de funciones de orden superior, funciones anónimas (*lambda*), y más.

1.5.1 Pasajes de parámetros y devolución de valores

Las funciones se definen utilizando la palabra clave `def`, seguida del nombre de la función y paréntesis que pueden contener parámetros. Las funciones pueden retornar valores utilizando la palabra clave `return`.

Python permite varios tipos de pasajes de parámetros a funciones:

```
def funcion(posicionales, nombrados, *posicionales_variables,
**nombrados_variables):
    pass # sentencia que no hace nada
```

Posicionales Los parámetros se pasan en el orden en que están definidos.

Nombrados Los parámetros se pasan utilizando su nombre, lo que permite especificar solo algunos de ellos.

Posicionales variables Se pueden pasar un número variable de argumentos posicionales utilizando `*`.

Nombrados variables Se pueden pasar un número variable de argumentos nombrados utilizando `**`.

1.5.1.1 Parámetros Posicionales

Los parámetros se ligán con los argumentos de la función en el orden en que están definidos. Si se pasan menos argumentos de los esperados, se generará un error.

```
def concatenar_cadenas(cadena1, cadena2):
    return cadena1 + cadena2

print(concatenar_cadenas("Hola, ", "mundo!")) # Salida: Hola, mundo!
print(concatenar_cadenas("mundo!", "Hola, ")) # Salida: mundo!Hola,
```

1.5.1.2 Parámetros Nombrados

Al momento de invocar la función se pueden nombrar los parámetros y de esa forma no es necesario respetar el orden de los parámetros. Esto es útil cuando se tienen muchos parámetros y se quiere especificar solo algunos.

```
def concatenar_cadenas(cadena1, cadena2):
    return cadena1 + cadena2

print(concatenar_cadenas(cadena2="mundo!", cadena1="Hola, "))
# Salida: Hola, mundo!
```

1.5.1.3 Parámetros Posicionales Variables

Los parámetros posicionales variables se definen utilizando un asterisco (`*`) antes del nombre del parámetro. Esto permite pasar un número variable de argumentos posicionales a la función. Python internamente agrupa estos argumentos en una tupla.

```
def sumar(*numeros):
    print(f"Se recibieron {len(numeros)} numeros")
    print(f"El tipo de numeros es: {type(numeros)}")
    suma = 0
    for num in numeros:
        suma += num
    return suma
```

```
print(sumar(1, 2, 3)) # Salida: 6
print(sumar(4, 5, 6, 7, 8)) # Salida: 30
```

1.5.1.4 Parámetros Nombrados Variables

Los parámetros nombrados variables se definen utilizando dos asteriscos (**) antes del nombre del parámetro. Esto permite pasar un número variable de argumentos nombrados a la función. Python internamente agrupa estos argumentos en un diccionario.

```
def mostrar_info(**info):
    print(f"Se recibieron {len(info)} argumentos nombrados")
    print(f"El tipo de info es: {type(info)}")
    for clave, valor in info.items():
        print(f"{clave}: {valor}")
```

```
mostrar_info(nombre="Juan", edad=30, ciudad="Madrid")
```

1.5.1.5 Parámetros por defecto

Los parámetros por defecto permiten definir valores predeterminados para los parámetros de una función. Si no se pasa un argumento para ese parámetro, se utilizará el valor por defecto.

```
def saludar(nombre="mundo"):
    return f"Hola, {nombre}!"
```

```
print(saludar()) # Salida: Hola, mundo!
print(saludar("Juan")) # Salida: Hola, Juan!
```

Importante

Si en una misma función se utilizan parámetros posicionales, nombrados, posicionales variables y nombrados variables, los parámetros deben seguir el siguiente orden:

1. Parámetros posicionales
2. Parámetros nombrados
3. Parámetros posicionales variables (*args)
4. Parámetros nombrados variables (**kwargs)

Por ejemplo:

```
def funcion_ejemplo(param1, param2="valor_por_defecto", *args, **kwargs):
    print(f"param1: {param1}, param2: {param2}")
    print(f"args: {args}")
    print(f"kwargs: {kwargs}")
```

```
funcion_ejemplo(1, 2, 3, 4, clave1="valor1", clave2="valor2")
```

```
def funcion_ejemplo(param1, param2="valor_por_defecto", *args, **kwargs):
    print(f"param1: {param1}, param2: {param2}")
    print(f"args: {args}")
    print(f"kwargs: {kwargs}")
```

```
funcion_ejemplo(1, (2, 3), clave1="valor1", clave2="valor2")
```

```
def funcion_ejemplo(param1, param2="valor_por_defecto", *args, **kwargs):
    print(f"param1: {param1}, param2: {param2}")
    print(f"args: {args}")
    print(f"kwargs: {kwargs}")
```

```
funcion_ejemplo(1, clave1="valor1", clave2="valor2")
```

1.5.1.6 Devolución de valores

Las funciones pueden devolver valores utilizando la palabra clave `return`. Si no se especifica un valor de retorno, la función devolverá `None` por defecto. En Python, una función puede devolver múltiples valores separados por comas, que se empaquetan en una tupla.

```
def division_y_resto(dividendo, divisor):
    cociente = dividendo // divisor
    resto = dividendo % divisor
    return cociente, resto

cociente, resto = division_y_resto(10, 3)
print(f"Cociente: {cociente}, Resto: {resto}")
```

Como Python tiene tipado dinámico, no es necesario especificar el tipo de retorno de una función. Sin embargo, se pueden utilizar anotaciones de tipo para documentar el tipo esperado de los parámetros y el valor de retorno.

```
def sumar(a: int, b: int) -> int:
    """
    Suma dos números enteros y devuelve el resultado.
    """
    return a + b

print(sumar(3, 5)) # Salida: 8
print(sumar("a", "b")) # Salida: ab
```

Las anotaciones de tipo son opcionales y no afectan el comportamiento de la función, pero pueden ser útiles para la documentación y la verificación de tipos en tiempo de desarrollo.

En el fragmento anterior, la función `sumar` está anotada para indicar que espera dos enteros como parámetros y devuelve un entero. Sin embargo, Python no impone estas restricciones en tiempo de ejecución, por lo que se pueden pasar otros tipos de datos sin generar un error.

1.5.2 Paradigma funcional

La **programación funcional** es un paradigma de programación declarativo basado en el uso de funciones verdaderamente matemáticas. En este estilo de programación las funciones son *ciudadanas de primera clase*, porque sus expresiones pueden ser asignadas a variables como se haría con cualquier otro valor; además de que pueden crearse funciones de orden superior.

Funciones de orden superior

Son aquellas que pueden:

- Recibir otras funciones como argumentos.
- Retornar una función como resultado.
- Ser asignadas a variables.

En el paradigma funcional en general (y a diferencia del imperativo), la programación consiste en especificar el **Qué** y no el **Cómo** se resuelve un problema.

Por ejemplo:

```
def componer(func1, func2):
    def funcion_compuesta(x):
        return func2(func1(x))

    return funcion_compuesta

def cuadrado(x):
    return x * x
```

```
def doble(x):
    return x + x
```

```
doble_cuadrado = componer(cuadrado, doble)
print(doble_cuadrado(3)) # Salida: 18
```

En este ejemplo, `componer` es una función de orden superior que toma dos funciones como argumentos y devuelve una nueva función que es la composición de las dos. Para poder elevar un número al cuadrado y luego duplicarlo, se utiliza `doble_cuadrado`, que es el resultado de `componer` `cuadrado` y `doble`. Hay que prestar atención a que el orden en que se componen las funciones, ya que se aplica primero `cuadrado` y luego `doble`.

1.5.2.1 Funciones anónimas (lambda)

Las funciones anónimas, también conocidas como funciones *lambda*, son funciones sin nombre que se definen utilizando la palabra clave `lambda`. Son útiles para crear funciones pequeñas y rápidas sin necesidad de definir las formalmente.

```
suma = lambda x, y: x + y
print(suma(3, 5)) # Salida: 8
```

Define una función que recibe dos argumentos `x` e `y` y retorna su suma. Esta función queda asociada a la variable `suma`, que se puede utilizar para invocarla.

El fragmento anterior de composición de funciones también se puede reescribir utilizando funciones *lambda*:

```
def componer(func1, func2):
    return lambda x: func2(func1(x))
```

```
doble_cuadrado = componer(lambda x: x * x, lambda x: x + x)
print(doble_cuadrado(3)) # Salida:
```

Las funciones de orden superior, las funciones anónimas, la generación de datos por comprensión y las clausuras son características del paradigma funcional que hacen de Python un lenguaje versátil y poderoso. Algunas de los usos habituales de la programación funcional en Python incluyen:

Mapeo Aplicar una función a cada elemento de una colección.

La función que mapea los elementos de una colección a otra debe ser una **función que toma un solo argumento y devuelve un valor**.

```
def mapear(func, iterable):
    """
    Aplica la función `func` a cada elemento de `iterable`
    y devuelve una lista con los resultados.
    """
    return [func(x) for x in iterable]
```

```
numeros = [x for x in range(10)]
cuadrados = mapear(lambda x: x**2, numeros)
print(f"Cuadrados: {cuadrados}")
```

Python proporciona la función `map` para realizar mapeo de manera más concisa y que permite devolver un iterador en lugar de una lista. Como todo iterador, una vez que se consume, es decir que se itera sobre él, no se puede volver a utilizar. Por lo tanto, es común convertirlo a una lista o tupla para conservar los resultados.

```
numeros = [x for x in range(10)]
cuadrados = map(lambda x: x**2, numeros)
```

```
print(type(cuadrados)) # <class 'map'>
tupla = tuple(cuadrados) # Convierte el iterador a tupla
print(f"Cuadrados: {tupla}")
```

Filtrado Seleccionar elementos de una colección que cumplen con una condición.

La función que filtra los elementos de una colección debe ser una **función que toma un solo argumento y devuelve un valor booleano**.

```
def filtrar(func, iterable):
    """
    Filtra los elementos de `iterable` que cumplen con la
    condición definida en `func`.
    """
    return [x for x in iterable if func(x)]
```

```
numeros = [x for x in range(10)]
pares = filtrar(lambda x: x % 2 == 0, numeros)
print(f"Números pares: {pares}")
```

En este caso la función de filtrado es una función anónima `lambda x: x % 2 == 0`. Las funciones anónimas siempre devuelven el resultado de la última expresión evaluada, por lo que no es necesario utilizar `return`.

Python también proporciona la función `filter` para realizar filtrado de manera más concisa. `filter` devuelve un iterador que contiene los elementos de la colección que cumplen con la condición especificada por la función de filtrado.

```
numeros = [x for x in range(10)]
pares = filter(lambda x: x % 2 == 0, numeros)
print(type(pares)) # <class 'filter'>
lista = list(pares) # Convierte el iterador a lista
print(f"Números pares: {lista}") #
```

Reducción Combinar los elementos de una colección en un solo valor.

El módulo `functools` proporciona la función `reduce`, que es una función de orden superior que puede aplicar una función de reducción a los elementos de una colección, combinándolos en un solo valor. **La función de reducción debe tomar dos argumentos y devolver un solo valor**.

En este caso, se utiliza para sumar todos los números de la lista. Aplica la suma a los dos primeros elementos, luego aplica la suma al resultado y el siguiente elemento, y así sucesivamente hasta que se procesan todos los elementos de la lista.

```
from functools import reduce

numeros = [x for x in range(10)]
suma_total = reduce(lambda x, y: x + y, numeros)
# ((((((0+1)+2)+3)+4)+5)+6)+7)+8)+9 = 45
print(f"Suma total: {suma_total}")
```

La función de reducción es la función anónima `lambda x, y: x + y`, que toma dos argumentos y devuelve su suma. No hace falta utilizar `return` ya que la última expresión evaluada es justamente la suma de `x` e `y`.

Otro ejemplo de reducción con cadenas de caracteres:

```
from functools import reduce

palabras = ["Python", "mundo", "Hola"]
frase = reduce(lambda x, y: y + " " + x, palabras)
print(f"Frase: {frase}")
```

En este caso, la función de reducción concatena las palabras en orden inverso, creando una frase a partir de la lista de palabras. El orden inverso se debe al orden en que concatena los elementos la función de reducción.

1.5.3 Funciones avanzadas

Iteradores Son objetos que permiten recorrer una secuencia de elementos uno a uno. En Python todas las colecciones son iterables, lo que significa que se pueden recorrer directamente utilizando un bucle `for` o se puede obtener un iterador con `iter` y luego utilizar la función `next` para obtener cada uno de los valores. Cuando `next` no tiene más elementos para devolver, lanza una excepción `StopIteration`. En los iteradores de Python no hay una función `has_next` como en otros lenguajes, sino que utiliza excepciones para detectar el final de la iteración.

```
numeros = [x for x in range(10)]
iterador = iter(numeros) #
while True:
    try:
        numero = next(iterador)
        print(numero)
    except StopIteration:
        break
```

Nota

En el capítulo Excepciones veremos en más detalle el manejo de excepciones en Python. Por ahora basta con saber que una excepción interrumpe el flujo normal del programa protegido por un bloque `try` y pasa el control al bloque `except` correspondiente. En este caso el bloque `except` captura la excepción `StopIteration` para finalizar la iteración.

La sentencia `break` se utiliza para romper y salir del bucle infinito `while True`.

Decoradores Son funciones que modifican el comportamiento de otras funciones. Se utilizan para agregar funcionalidades adicionales a funciones existentes sin modificar su código.

```
def decorador(func):
    """
    Decora la función `func` para agregarle mensajes al valor de retorno.
    """

    def funcion_decorada(*args, **kwargs):
        resultado = f"El resultado de la operación es: {func(*args,
        **kwargs)}"
        return resultado

    return funcion_decorada

def funcion_original(x):
    return x * 2

funcion_decorada = decorador(funcion_original)
funcion_decorada(5)
```

La función `decorador` toma una función `func` como argumento y devuelve una nueva función, `funcion_decorada` que agrega el mensaje *El resultado de la operación es:* al resultado de la función original.

Nota

En este ejemplo se utiliza `*args` y `**kwargs` para permitir que la función decorada acepte cualquier número de argumentos posicionales y nombrados, lo que la hace más flexible. En la línea 7, la expresión `func(*args, **kwargs)` invoca a la función original con los argumentos que le pasaron a la función decorada.

Python proporciona una sintaxis especial para aplicar decoradores a funciones utilizando el símbolo `@` antes de la definición de la función. Esto es equivalente a decorar la función manualmente como se mostró anteriormente.

```
def decorador(func):
    def funcion_decorada(*args, **kwargs):
        resultado = f"El resultado de la operación es: {func(*args,
**kwargs)}"
        return resultado

    return funcion_decorada
```

```
@decorador
def funcion_original(x):
    return x * 2
```

```
print(funcion_original(5))
```

```
@decorador
def funcion_suma(a, b):
    return a + b
```

```
print(funcion_suma(3, 4))
```

Generadores Son funciones que permiten crear iteradores de manera eficiente. Utilizan la palabra clave `yield` para devolver un valor y pausar la ejecución de la función, permitiendo que se reanude más tarde desde el mismo punto.

```
def contador():
    i = 0
    while True:
        yield i
        i += 1
```

```
contador_gen = contador()
print(type(contador_gen)) # <class 'generator'>
print(next(contador_gen)) # Salida: 0
print(next(contador_gen)) # Salida: 1
print(next(contador_gen)) # Salida: 2
```

Este generador nos permite, de alguna manera, tener una lista infinita de números enteros, ya que cada vez que se llama a `next`, se obtiene el siguiente número en la secuencia, lo cual es posible gracias a la palabra clave `yield`, que suspende la ejecución de la función en ese punto, devuelve el valor actual de `i`, y guarda el estado de la función para que pueda reanudarse en la siguiente llamada a `next`. En este caso `next` no levantará una excepción `StopIteration` porque el generador está diseñado para ser infinito.

También se puede utilizar la clausura para obtener un comportamiento similar a los generadores:

```
def contador():
    i = 0
```

```
def siguiente():
    nonlocal i
    valor = i
    i += 1
    return valor

return siguiente

siguiente = contador()
print(siguiente()) # Salida: 0
print(siguiente()) # Salida: 1
print(siguiente()) # Salida: 2
```

La clave está en utilizar `nonlocal` para modificar la variable `i` dentro de la función interna `siguiente`, permitiendo que se mantenga el estado entre llamadas. `i` se almacena en la clausura de `siguiente`, lo que permite que su valor persista entre invocaciones.

1.5.4 Recursos para profundizar

- Tutorial de Python - Funciones
- Python Functions (W3Schools)
- Python Lambda Functions (W3Schools)
- Python Iterators (W3Schools)
- Funciones Decoradoras (Hektor Profe)
- Funciones Lambda (Hektor Profe)
- Funciones Generadoras (Hektor Profe)
- Función Filter (Hektor Profe)
- Función Map (Hektor Profe)

1.6 Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza “objetos” para representar datos y comportamientos. En Python, la POO se implementa a través de clases e instancias de esas clases, los objetos.

Las clases permiten definir nuevos tipos de datos, encapsulando atributos (datos) y métodos (funciones) que operan sobre esos datos. Esto facilita la organización del código, la reutilización y la creación de programas más complejos de manera estructurada.

1.6.1 Algunos conceptos clave de la POO

Atributos Son variables que pertenecen a una clase y representan el estado del objeto. Los atributos pueden ser de diferentes tipos, como enteros, cadenas, listas, etc. Cada objeto en memoria tiene sus propios valores para estos atributos. Estos atributos determinan el estado del objeto.

Métodos de Instancia Son funciones definidas dentro de una clase que operan sobre los atributos del objeto.

Métodos de Clase Son métodos que pertenecen a la clase en sí, no a las instancias de la clase. Es decir, son funciones que pueden ser llamadas sin necesidad de crear un objeto de la clase.

Comportamiento Se refiere al conjunto de métodos a los que puede responder un objeto. Los métodos definen cómo interactúa el objeto con otros objetos y cómo se comporta en diferentes situaciones.

Herencia Permite que una clase herede atributos y métodos de otra clase, lo que facilita la reutilización de código y la creación de jerarquías de clases. La clase que hereda se llama “subclase” o “clase derivada”, mientras que la clase de la que hereda se llama “superclase” o “clase base”. Python soporta herencia múltiple, lo que significa que una clase puede heredar de múltiples clases al mismo tiempo.

Constructor Es un método especial que se llama automáticamente cuando se crea un objeto de la clase. La finalidad del constructor es inicializar los atributos del objeto.

1.6.2 Clases y objetos

En Python, una clase es una plantilla para crear objetos, similar a los struct de Go, una clase permite definir nuevos tipos de datos. Un objeto, en cambio, es una instancia de una clase y puede tener atributos (datos) y métodos (funciones). Pueden existir múltiples objetos de la misma clase, cada uno con sus propios valores para los atributos.

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

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")
```

```
# Crear un objeto de la clase Persona
personal = Persona("Alice", 30)
personal.saludar()
print(type(personal))
```

En el fragmento de código anterior, se define una clase Persona con un constructor (__init__) que inicializa los atributos nombre y edad. También se define un método saludar que imprime un saludo. Luego, se crea un objeto personal de la clase Persona y se llama al método saludar.

El constructor en Python siempre es __init__ y se utiliza para inicializar los atributos del objeto. Si no se declara explícitamente, Python proporcionará un constructor por defecto que no hace nada.

El primer parámetro de los métodos de instancia es siempre `self`, que se refiere a la instancia actual del objeto. Esto permite acceder a los atributos y métodos del objeto dentro de la clase. Es similar al `this` de Java.

1.6.3 Herencia

Una clase puede heredar de otra clase, o dicho de otra manera, puede extender otra clase. Esto permite que una clase herede atributos y métodos de su ancestro, y los pueda utilizar directamente sin tener que declararlos nuevamente, lo que facilita la reutilización de código y la creación de jerarquías de clases.

```
class Docente(Persona):
    def __init__(self, nombre, edad, materia):
        super().__init__(nombre, edad) # Llama al constructor de la clase base
        self.materia = materia

    def presentar(self):
        self.saludar() # Usa método heredado
        print(f"Soy docente de {self.materia}.")

# Crear un objeto de la clase Docente
docente1 = Docente("Juan", 25, "Algoritmos y Programación")
docente1.presentar()
print(type(docente1))
```

En el ejemplo anterior, la clase `Docente` hereda de la clase `Persona`. Es decir un `Docente` **es una** `Persona` y por lo tanto tiene todos los atributos y métodos de cualquier `Persona` y además tiene nuevos atributos y métodos como `Docente`.

El constructor de la clase `Docente` espera todos los parámetros para poder instanciar un nuevo objeto del tipo `Docente`, esto es el nombre y la edad de la `Persona` y la materia de la que es `Docente`.

Lo primero que hace el constructor de la clase `Docente` es llamar al constructor de la clase base `Persona` (línea 3) utilizando `super().__init__(nombre, edad)`. Esto asegura que los atributos `nombre` y `edad` se inicialicen correctamente en el objeto `Docente`.

`super()` es una función que permite llamar a métodos de la clase base desde una subclase. Cuando se invoca `super().__init__(nombre, edad)`, se está llamando al constructor de la clase base `Persona`.

En este caso como tanto `Docente` como `Persona` tienen un constructor, `__init__`, es necesario usar `super()` para llamar al constructor de la clase base, caso contrario, se estaría llamando al constructor de la clase `Docente`.

Con esta estrategia se evita la duplicación de código y se asegura que los atributos de la clase base se inicialicen correctamente. Además, si por algún motivo hay que modificar el constructor de la clase base, no es necesario cambiar el código de la subclase.

En el método `presentar`, se llama al método `saludar` de la clase base `Persona` utilizando `self.saludar()`, lo que permite reutilizar el comportamiento definido en la clase base. En este caso como el método `saludar` no fue **sobrescrito** en la clase `Docente`, directamente se puede utilizar `self.saludar()` para llamar al método de la clase base, sin problemas.

1.6.3.1 Herencia múltiple

Python va más allá que otros lenguajes de programación orientados a objetos y permite la herencia múltiple, lo que significa que una clase puede heredar de múltiples clases al mismo tiempo. Esto se logra especificando múltiples clases base en la definición de la clase.

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

```

        self.edad = edad

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años")

class Docente(Persona):
    def __init__(self, legajo, nombre, edad, materia):
        Persona.__init__(self, nombre, edad)
        self.legajo = legajo # legajo del Docente como Empleado
        self.materia = materia

    def presentar(self):
        self.saludar()
        print(f"Soy docente de {self.materia}")

    def pagar(self):
        print(f"Pago realizado al docente {self.nombre}, "
              f"con legajo {self.legajo}.")

class Estudiante(Persona):
    def __init__(self, legajo, nombre, edad, carrera):
        Persona.__init__(self, nombre, edad)
        self.legajo = legajo # legajo del Estudiante como Alumno
        self.carrera = carrera

    def presentar(self):
        self.saludar()
        print(f"Soy estudiante de {self.carrera}")

    def mostrar_legajo(self):
        print(f"Legajo: {self.legajo}, Nombre: {self.nombre}.")

    def actualizar_legajo(self, nuevo_legajo):
        self.legajo = nuevo_legajo
        print(f"Legajo actualizado a: {self.legajo}")

class Ayudante(Estudiante, Docente):
    def __init__(self, legajo, nombre, edad, materia, carrera):
        Estudiante.__init__(self, legajo, nombre, edad, carrera)
        Docente.__init__(self, legajo, nombre, edad, materia)

    def presentar(self):
        self.saludar()
        print(f"Soy estudiante de {self.carrera} y ayudante en "
              f"{self.materia}")

# Crear un objeto de la clase Docente
docente1 = Docente(64781, "Juan", 30, "Algoritmos y Programación")
docente1.presentar()
print()

# Crear un objeto de la clase Estudiante
estudiante1 = Estudiante(30415, "Ana", 20, "Ingeniería en Computación")
estudiante1.presentar()
print()

# Crear un objeto de la clase Ayudante
ayudante1 = Ayudante(
    29478, "Luis", 23, "Algoritmos y Programación", "Ingeniería en Sonido"
)
ayudante1.presentar()
print()

```

```

print(type(ayudante1))

print("Pagar a docentes")
for docente in [docente1, ayudante1]:
    docente.pagar() # Llama al método pagar de Docente
print()

print("Mostrar legajo de estudiantes")
for estudiante in [estudiante1, ayudante1]:
    estudiante.mostrar_legajo()
print()

# Actualizamos el legajo del ayudante
ayudante1.actualizar_legajo(12345)
ayudante1.pagar()

```

En el fragmento anterior se definen cuatro clases: Persona, Docente, Estudiante y Ayudante. Docente y Estudiante heredan de Persona, mientras que Ayudante hereda de Estudiante y Docente a través de la herencia múltiple. Lo que le permite a un Ayudante heredar el comportamiento de ambas clases además de sumar sus propios atributos y métodos.

En el siguiente *diagrama de clases* se puede observar la relación entre las clases y los atributos y métodos de cada una.

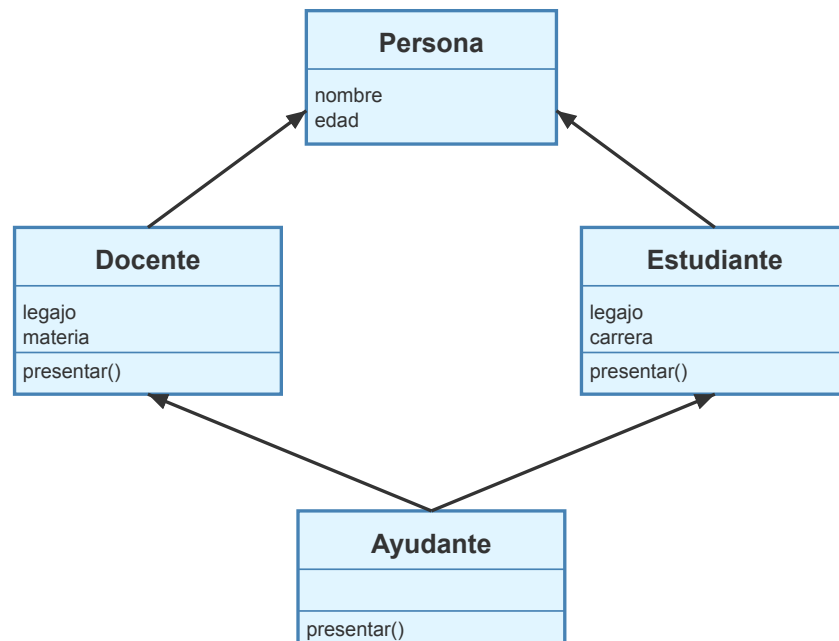


Figure 1.7: Diagrama de clases de Ayudante

Nota

Un **diagrama de clases**, es un diagrama estático que muestra la estructura de un sistema mediante las clases que lo componen y sus relaciones. Una flecha con una línea continua y un triángulo en la punta indica herencia, y en cada clase se pueden ver los atributos y métodos que se definen en cada una. En general no se muestran los atributos y métodos heredados, ni los constructores, pero sí los métodos que se sobrescriben, es decir, que se redefinen en una subclase. En el diagrama anterior el método presentar de la clase Ayudante sobrescribe el método presentar de las clases Docente y Estudiante.

Cuando hay herencia múltiple se recomienda utilizar el nombre de la clase base explícitamente en el constructor de la subclase en lugar de `super()`. En la clase `Docente`, línea 12 se utiliza directamente `Persona` en lugar de usar `super()`, y en la línea 26 de la clase `Estudiante` también, para evitar ambigüedades en la resolución de métodos y atributos.

En el ejemplo anterior, tanto `Docente` como `Estudiante` tienen un atributo `legajo`, por lo que al crear un objeto de la clase `Ayudante`, se debe especificar explícitamente a qué clase base se está llamando.

`Ayudante` hereda de ambas clases que tienen un atributo `legajo`, sin embargo el atributo `legajo` no se duplica en un objeto de la clase `Ayudante`. Lo que permite que con el mismo `legajo` de `Estudiante` se pueda pagar al `Docente`.

El siguiente fragmento de código inspecciona los atributos del objeto `ayudante1` y los imprime en la consola (En Inspección veremos más en detalle como los objetos pueden observarse y modificarse a si mismos en tiempo de ejecución)

```
print("Atributos de ayudante1:")
atributos = vars(ayudante1)
for key, value in atributos.items():
    print(f"{key}: {value}")
```

1.6.4 Polimorfismo

El polimorfismo es un concepto clave en la POO que permite que diferentes clases implementen métodos con el mismo nombre, pero con comportamientos diferentes. Esto significa que se puede tratar a objetos de diferentes clases de manera uniforme, utilizando el mismo nombre de método.

En el ejemplo anterior el método `presentar` se define tanto en `Docente` como en `Estudiante`. La clase `Ayudante` hereda ambos métodos y además lo sobrescribe.

Cuando hay herencia múltiple se debe tener cuidado como se resuelve el polimorfismo, ya que puede haber ambigüedades si dos clases base tienen un método con el mismo nombre. En Python, se utiliza el **Orden de Resolución de Métodos (MRO)** para determinar qué método se llama en caso de ambigüedad.

```
print("Orden de Resolución de Métodos (MRO) de Ayudante:")
orden = Ayudante.__mro__
for cls in orden:
    print(cls.__name__)
```

Es decir cuando se invoca un método cualquier de un objeto de la clase `Ayudante`, Python primero busca el método en la propia clase `Ayudante`, luego en `Estudiante`, luego en `Docente`, luego en `Persona` y finalmente en `object`.

Por ese motivo no se puede usar `super()` en el constructor de la clase `Estudiante` ya que si se hiciera, se generaría una ambigüedad en la resolución del método a llamar, ya que `super()` buscaría el siguiente método en la jerarquía de clases, que en este caso sería el constructor de `Docente`, lo que no es lo que se desea.

Nota

`object` es la clase base de todas las clases en Python. Todas las clases heredan de `object`, lo que significa que todas las instancias de clases en Python son también instancias de `object`. Esto proporciona una serie de métodos y atributos comunes a todas las clases.

Por eso cuando se dice que en Python todo es un objeto, se refiere a que todas las clases heredan de `object`, y por lo tanto, todas las instancias de clases son también instancias de `object`. Esto permite que todas las clases tengan un comportamiento común, como la capacidad de ser comparadas, impresas, etc.

1.6.4.1 Duck typing

El **duck typing** es un concepto en Python que se basa en la idea de que el tipo de un objeto se determina por su comportamiento en lugar de su clase. Es decir, si un objeto tiene los métodos y atributos necesarios para realizar una tarea, se puede tratar como si fuera de un tipo específico, sin necesidad de verificar su clase.

1.6.5 “Si camina como un pato, nada como un pato y grazna como un pato, entonces probablemente sea un pato.”

Principio fundamental del *duck typing* en Python

Este concepto es muy poderoso en Python, ya que permite que diferentes objetos puedan ser utilizados de manera intercambiable si cumplen con la interfaz esperada, sin necesidad de heredar de una clase específica. Todo ocurre de manera implícita, sin necesidad de declarar explícitamente que un objeto es de un tipo específico y sin necesidad de especificar interfaces o clases abstractas como en Java o Go. Esta versatilidad es gracias al sistema de tipos dinámico de Python.

1.6.5.1 Ejemplo

En el siguiente ejemplo, cuyo diagrama de clases se muestra a continuación. Primero se define Punto que representa un punto en el plano cartesiano y luego se definen las Figuras Geométricas Cuadrado, Rectángulo, Elipse y Triángulo que están **compuestas** por Punto. Cada figura tiene un método `area` que calcula su área, pero cada figura lo implementa de manera diferente.

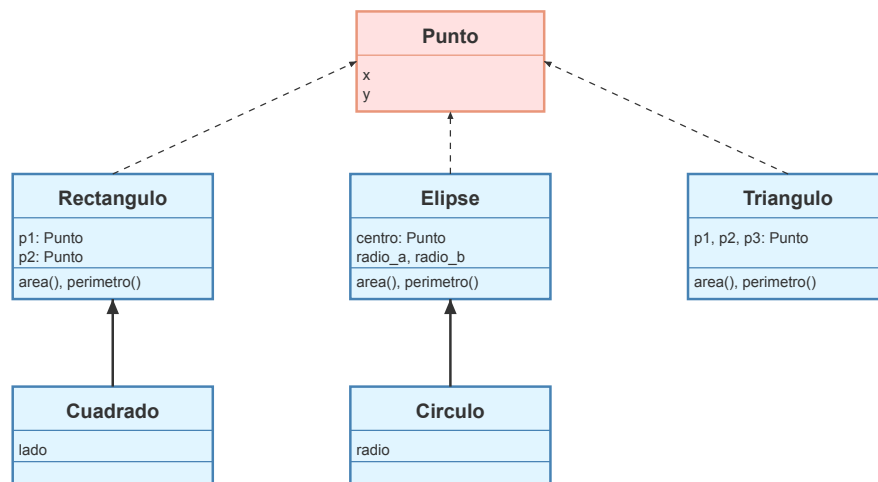


Figure 1.8: Diagrama de clases de Figuras Geométricas

```
"""
```

```
Demostración de polimorfismo con figuras geométricas.
```

```
Este módulo muestra cómo diferentes tipos de figuras pueden ser  
tratadas de manera uniforme gracias al polimorfismo de Python.
```

```
"""
```

```
from punto import Punto
from rectangulo import Rectangulo
from cuadrado import Cuadrado
from circulo import Circulo
from elipse import Elipse
from triangulo import Triangulo
```

```

def imprimir_propiedades_figura(figura):
    """
    Imprime las propiedades de una figura usando polimorfismo.

    Args:
        figura: Cualquier objeto que tenga métodos perimetro() y area().
    """
    print(f"Figura: {figura}")
    print(f"  Perímetro: {figura.perimetro():.2f}")
    print(f"  Área: {figura.area():.2f}")
    print()

def demo_polimorfismo():
    """
    Demostración de polimorfismo con una lista de diferentes figuras.
    """
    print("=== Demostración de Polimorfismo con Figuras ===\n")

    # Crear diferentes tipos de figuras
    figuras = [
        # Rectángulo de 3x4
        Rectangulo(Punto(0, 0), Punto(3, 4)),
        # Cuadrado de lado 5
        Cuadrado(Punto(1, 1), 5),
        # Círculo de radio 3
        Circulo(Punto(2, 2), 3),
        # Elipse con radios 4 y 2
        Elipse(Punto(0, 0), 4, 2),
        # Triángulo rectángulo
        Triangulo(Punto(0, 0), Punto(3, 0), Punto(0, 4)),
        # Triángulo equilátero aproximado
        Triangulo(Punto(0, 0), Punto(2, 0), Punto(1, 1.732)), # altura ≈ √3
    ]

    # Demostrar polimorfismo: mismo código funciona para todas las figuras
    total_perimetro = 0
    total_area = 0

    for i, figura in enumerate(figuras, 1):
        print(f"--- Figura {i} ---")
        imprimir_propiedades_figura(figura)

        # Acumular totales
        total_perimetro += figura.perimetro()
        total_area += figura.area()

    # Mostrar totales
    print("\n * 50")
    print(f"Total de figuras: {len(figuras)}")
    print(f"Perímetro total: {total_perimetro:.2f}")
    print(f"Área total: {total_area:.2f}")

def demo_herencia():
    """
    Demostración de relaciones de herencia.
    """
    print("\n=== Demostración de Herencia ===\n")

    # Crear instancias
    rectangulo = Rectangulo(Punto(0, 0), Punto(4, 3))
    cuadrado = Cuadrado(Punto(0, 0), 4)
    elipse = Elipse(Punto(0, 0), 3, 2)
    circulo = Circulo(Punto(0, 0), 3)

```

```

        print("Relaciones de herencia:")
        print(f"¿Cuadrado es instancia de Rectángulo? " f"{isinstance(cuadrado, Rectangulo)}")
        print(f"¿Círculo es instancia de Elipse? " f"{isinstance(circulo, Elipse)}")
        print(f"Rectángulo es instancia de Cuadrado? " f"{isinstance(rectangulo, Cuadrado)}")
        print()

        # Mostrar jerarquías de clases
        print("Jerarquías de clases:")
        print(f"Cuadrado MRO: " f"{[cls.__name__ for cls in Cuadrado.__mro__]}")
        print(f"Círculo MRO: " f"{[cls.__name__ for cls in Circulo.__mro__]}")

def demo_casos_especiales():
    """
    Demostración de casos especiales y validaciones.
    """
    print("\n=== Casos Especiales ===\n")

    # Figuras con valores por defecto
    print("Figuras con valores por defecto:")
    figuras_default = [Rectangulo(), Cuadrado(), Circulo(), Elipse(), Triangulo()]

    for figura in figuras_default:
        print(f"{figura} -> Área: {figura.area():.2f}")

    print()

    # Casos límite
    print("Casos límite:")

    # Triángulo degenerado (área = 0)
    triangulo_degenerado = Triangulo(Punto(0, 0), Punto(1, 0), Punto(2, 0))
    print(f"Triángulo degenerado: {triangulo_degenerado}")
    print(f"Área: {triangulo_degenerado.area():.2f}")

    # Círculo muy pequeño
    circulo_pequeno = Circulo(Punto(0, 0), 0.001)
    print(f"Círculo pequeño: {circulo_pequeno}")
    print(f"Área: {circulo_pequeno.area():.6f}")

if __name__ == "__main__":
    demo_polimorfismo()
    demo_herencia()
    demo_casos_especiales()

```

Program 1.9: Programación Orientada a Objetos - Figuras Geométricas

[Descargar código completo de Figuras Geométricas](#)

1.6.6 Recursos para profundizar

- Tutorial de Python - Clases
- Programación Orientada a Objetos (Hektor Profe)
- Python - Clases y Objetos (W3Schools)

1.7 Manejo de excepciones

Las excepciones son eventos que ocurren durante la ejecución de un programa y que interrumpen su flujo normal. Este mecanismo no solo permite manejar errores en tiempo de ejecución, sino que también permite gestionar otras situaciones excepcionales que pueden surgir.

El mecanismo de manejo de excepciones se basa en el uso de bloques try, except, else y finally. Por cada bloque try puede haber uno o varios bloques except, uno por cada tipo de excepción que se quiera manejar. A continuación, se describen cada uno de estos bloques

```
try:
    # Bloque protegido: Código que puede generar una excepciones
    llamado_a_funcion_que_puede_fallar()

except ExceptionType1:
    print("ocurrió una excepción del tipo ExceptionType1")

except (ExceptionType2, ExceptionType3):
    print("ocurrió una excepción que puede ser de tipo ExceptionType2 o "
          "ExceptionType3")

except ExceptionType4 as e:
    # Manejador de excepciones del tipo ExceptionType4. `e` es una variable
    # con información sobre la excepción que se puede usar dentro del handler
    print("ocurrió un error del tipo ExceptionType4 con mensaje", e)

except:
    print("se produjo una excepción no controlada")

else:
    print("no ocurrió ninguna excepción")

finally:
    print("este bloque se ejecutará siempre")
```

try Este bloque contiene el código que podría generar una excepción. Mientras se ejecuta este bloque, Python “vigila” la aparición de errores. Si se produce una excepción, la ejecución del bloque try se interrumpe inmediatamente y se transfiere el control al primer except que pueda manejarla. Se suele decir que el código dentro de un try está **protegido**, porque ante un error no provoca que el programa finalice abruptamente, sino que permite reaccionar y manejar la situación. Por ejemplo, si debemos dividir dos números cuyo valor no conocemos de antemano, existe la posibilidad de una división por cero; por eso el cálculo puede colocarse dentro de un bloque try para atraparlo y manejarlo.

except En Python, un bloque except se ejecuta solo si ocurre una excepción en el bloque try asociado. Este bloque actúa como un **manejador de excepciones** (*exception handler*) y puede realizar diversas acciones: registrar el error en un log, mostrar un mensaje al usuario, o incluso intentar recuperarse ejecutando una operación alternativa. El comportamiento del except depende del tipo de excepción y de la lógica del programa. Por ejemplo, si se produce una división por cero (ZeroDivisionError), el except podría mostrar un mensaje de error o sustituir el divisor por un valor por defecto para continuar la ejecución. Si no ocurre ninguna excepción en el bloque try, ningún except se ejecuta y el flujo continúa normalmente después del try/except. Cuando se lanza una excepción, Python busca el primer bloque except que pueda manejarla. La búsqueda se hace de arriba hacia abajo en el orden en que están escritos, por lo que conviene colocar primero los except que capturan excepciones más específicas y dejar al final un except genérico. El except genérico no especifica un tipo de excepción y atrapa cualquier excepción que sea instancia de Exception.

else Es un bloque opcional que se ejecuta solo si el bloque try termina sin generar excepciones. Se usa para colocar el código que depende de que la operación haya salido bien, pero que no conviene poner directamente en el try para no atrapar errores de

forma innecesaria. En otras palabras: `try` arrow.r si no hay error arrow.r `else`. Si hay error arrow.r `except` (y el `else` no se ejecuta).

finally Es un bloque opcional que se ejecuta siempre, sin importar si el `try` terminó con o sin excepción. Sirve para ejecutar tareas de limpieza o liberación de recursos que deben realizarse pase lo que pase, como cerrar un archivo, liberar memoria o cerrar una conexión de red. Incluso si en el `try` o `except` se usa `return` o se lanza una nueva excepción, el `finally` siempre se ejecuta antes de que el flujo salga del bloque.

```
divisor = "10" # cadena de caracteres

try:
    # Código que puede generar excepciones
    resultado = 10 // int(divisor) # // División entera

except ZeroDivisionError as e:
    # Manejo de la excepción división por cero
    print(f"No se puede dividir por cero: {e}")

except ValueError as e:
    # Manejo de la excepción para valores no numéricos
    print(f"Error: {e}")

except:
    # Manejador de excepciones genérico
    print("Se produjo una excepción no controlada")

else:
    # Este bloque se ejecuta si no hay excepciones
    print("La división fue exitosa:", resultado)

finally:
    # Este bloque se ejecuta siempre
    print("Bloque finally ejecutado")
```

Actividad

Copia el fragmento anterior y modifícalo para que divida por 0. Por ejemplo podés probar asignando distintos valores a `divisor` como ser "0", la cadena "cero", números enteros, números decimales, y todo lo que se te ocurra.

¿Qué pasa en cada caso? ¿Es buena idea tener un bloque `except` genérico?

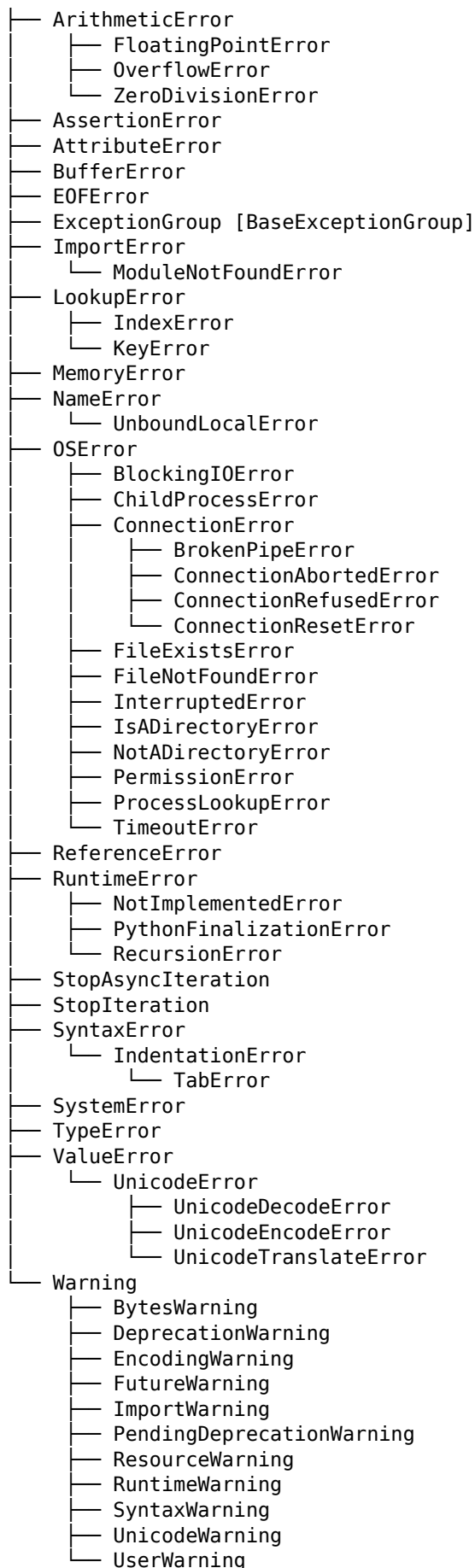
1.7.1 Jerarquía de excepciones

En Python existen muchas excepciones predefinidas que están organizadas en forma jerárquica. Es decir que algunas excepciones son subclases de otras. Por ejemplo, `ZeroDivisionError` es una subclase de `ArithmeticError`, que a su vez es una subclase de `Exception`. Esto significa que si se captura una excepción de tipo `ArithmeticError`, también se capturarán las excepciones de tipo `ZeroDivisionError`.

La jerarquía de excepciones permite manejar las excepciones de manera más específica y detallada. Por ejemplo, si se desea manejar todas las excepciones relacionadas con operaciones aritméticas, se puede capturar la excepción `ArithmeticError`. Si se desea manejar solo las excepciones de división por cero, se puede capturar la excepción `ZeroDivisionError`.

Click para ver la jerarquía de excepciones

```
BaseException
├── BaseExceptionGroup
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
└── Exception
```



KeyboardInterrupt Esta excepción se genera cuando el usuario interrumpe la ejecución del programa, generalmente presionando Ctrl+C en la consola. Es una subclase de

`BaseException`, lo que significa que no es una excepción común y no se debe capturar a menos que se tenga un motivo específico para hacerlo.

SystemExit Esta excepción se genera cuando se llama a la función `sys.exit()`. Se utiliza para finalizar un programa de manera controlada. Al igual que `KeyboardInterrupt`, es una subclase de `BaseException`, por lo que no se debe capturar a menos que se tenga un motivo específico para hacerlo.

Exception Esta es la clase base para todas las excepciones que no son errores del sistema. Todas las excepciones que se generan durante la ejecución de un programa son subclases de `Exception`.

Warning Esta clase base se utiliza para advertencias que no son errores, pero que pueden indicar problemas potenciales en el código. Las advertencias no interrumpen la ejecución del programa, pero pueden ser útiles para identificar problemas que podrían surgir en el futuro. Por ejemplo, si se utiliza una función que está obsoleta, Python generará una advertencia de deprecación. Las advertencias se pueden capturar y manejar de manera similar a las excepciones, pero generalmente no se recomienda hacerlo, ya que las advertencias son más informativas que críticas.

En el siguiente fragmento de código se observa un bucle donde se le pide al usuario que ingrese un número entero por teclado y se acumula la suma para finalmente mostrarla. Para salir del bucle el usuario debe presionar `Ctrl-C`

```
"""Suma números enteros ingresados por el usuario."""

suma = 0
while True:
    try:
        numero = int(input("Ingrese un número entero (Ctrl-C para salir): "))
        suma += numero
    except ValueError:
        print("Error: Debe ingresar un número entero.")
    except KeyboardInterrupt:
        print("\nSaliendo...")
        break # rompe el bucle while True
    else:
        print(f"Suma parcial: {suma}")
print(f"Total acumulado: {suma}")
```

1.7.2 Excepciones creadas por el usuario

Además de las excepciones predefinidas, Python permite a los desarrolladores crear sus propias excepciones personalizadas. Esto es útil cuando se desea manejar situaciones específicas que no están cubiertas por las excepciones predefinidas. Por ejemplo si tenemos una clase pila, podríamos querer lanzar una excepción si se intenta desapilar un elemento de una pila vacía. Podemos crear una excepción personalizada llamada `StackException` para manejar esta situación. Para crear una excepción personalizada, se debe definir una nueva clase que herede de la clase `Exception` o de alguna de sus subclases. En general se recomienda heredar directamente de `Exception`, a menos que se tenga un motivo específico para hacerlo de otra manera, por ejemplo si se desea crear una jerarquía de excepciones personalizadas.

El cuerpo de la clase puede estar vacío, ya que la funcionalidad principal de la excepción se hereda de la clase base. Sin embargo, se pueden agregar atributos o métodos adicionales si se desea proporcionar información adicional sobre la excepción.

```
class StackException(Exception):
    pass
```

1.7.2.1 ¿Porqué crear excepciones personalizadas?

Crear excepciones personalizadas tiene varias ventajas, entre ellas:

Claridad Las excepciones personalizadas pueden tener nombres descriptivos que indican claramente el tipo de error que ha ocurrido. Esto facilita la comprensión del código y la

identificación de problemas. Por ejemplo si se produce una excepción al intentar desapilar de una pila vacía que está implementada usando listas, una excepción personalizada llamada `StackException` es más clara que una excepción genérica como `IndexError`.

Manejo específico Al crear excepciones personalizadas, se puede manejar de manera específica cada tipo de error. Esto permite implementar lógica de manejo de errores más precisa y adecuada para cada situación.

Jerarquía de excepciones Al crear una jerarquía de excepciones personalizadas, se puede organizar y estructurar el manejo de errores de manera más efectiva. Por ejemplo, se puede crear una excepción base para un módulo o una clase, y luego crear subclasses para manejar errores específicos dentro de ese contexto.

Reutilización Las excepciones personalizadas pueden ser reutilizadas en diferentes partes del código o en diferentes proyectos, lo que facilita la consistencia en el manejo de errores.

Documentación Las excepciones personalizadas pueden incluir documentación adicional que explique cuándo y por qué se deben lanzar, lo que ayuda a otros desarrolladores a entender su propósito y uso.

1.7.3 Lanzando excepciones

Las excepciones se lanzan utilizando la instrucción `raise`, seguida de una instancia de la excepción que se desea lanzar. Por ejemplo, si se desea lanzar una excepción personalizada llamada `StackException`, se puede hacer de la siguiente manera:

```
raise StackException("La pila está vacía")
```

En el manejo de excepciones hay dos momentos bien definidos, cuando se **lanza una excepción** y cuando se **captura una excepción**.

El lanzamiento se realiza cuando surge una situación excepcional que el programa no puede manejar de manera normal. En este punto, se utiliza la instrucción `raise` para generar la excepción y transferir el control a un bloque `except` correspondiente.

La captura la realiza el *usuario* de nuestro programa. Este *usuario* puede intentar manejar la excepción y recuperarse. Si una excepción no se captura, entonces el programa se detendrá y mostrará un mensaje de error.

Importante

Si nuestro programa lanza una excepción y la atrapa en el mismo módulo es probable que la lógica de manejo de excepciones no sea adecuada. Ya que si lanzó una excepción es porque no podía manejar la situación, entonces si luego en el mismo bloque la atrapa entonces desde el principio no era una situación inmanejable como para lanzar una excepción

Otra forma de lanzar una excepción en forma condicional es con la sentencia `assert`.

`assert` es una declaración que se utiliza para realizar pruebas de afirmaciones en el código. Si la afirmación es falsa, se lanza una excepción `AssertionError`. Esto puede ser útil para verificar condiciones que deberían ser verdaderas en un punto determinado del programa. Por ejemplo para chequear se cumplan las precondiciones cuando se llama a una función.

```
def dividir(a, b):  
    assert b != 0, "El divisor no puede ser cero"  
    return a / b
```

```
if __name__ == "__main__":  
    try:  
        resultado = dividir(10, 0)  
    except AssertionError as e:  
        print(f"Error: {e}")
```

1.7.4 Consideraciones en el diseño y uso de excepciones en Python

Limitar el alcance del bloquetry Colocar dentro del `try` solo el código que pueda generar la excepción que se desea manejar para evitar atrapar errores no relacionados y facilitar la identificación de la causa.

Evitar atrapar excepciones genéricas sin necesidad No usar `except Exception:` ni `except:` a menos que realmente se quiera interceptar cualquier excepción. Atrapar todo oculta errores y dificulta el depurado.

Usar excepciones específicas primero Ordenar los `except` de más específico a más genérico, ya que Python ejecuta el primero que coincide. Ejemplo:

```
except FileNotFoundError:
    ...
except OSError:
    ...
except Exception:
    ...
```

No usar excepciones para control de flujo normal Las excepciones son para manejar situaciones excepcionales, no para reemplazar estructuras de control (`if`, `for`, etc.).

Registrar o informar el error Si la excepción se maneja sin mostrar o registrar nada, puede ser difícil saber qué ocurrió. Se recomienda usar `logging` o `print` (en entornos simples) para tener contexto de la excepción.

Liberar recursos siempre Usar `finally` para cerrar archivos, conexiones, o liberar memoria, sin importar si hubo excepción o no.

No silenciar excepciones sin justificación Evitar `except: pass` sin explicación. Si es necesario ignorar un error, se debe documentar por qué.

Relanzar cuando sea necesario Si el bloque `except` no puede manejar la excepción de manera útil, se debe volver a lanzarla (`raise`) para que sea tratada en un nivel superior.

Mantener el bloque `except` lo más simple posible Los `handlers` deben ser bloques de código simple, ya que un nuevo error que se genere allí podría ocultar la situación excepcional original.

Personalizar excepciones si es necesario Crear clases de excepciones propias cuando la aplicación puede tener errores específicos, para que sea más fácil distinguirlos. Por ejemplo `StackException`.

Usar `else` para el código dependiente de éxito Colocar en `else` las operaciones que deben ejecutarse solo si no hubo excepción, en lugar de ponerlas en el `try`. Esto ayuda a mantener el código más claro y a separar la lógica de manejo de errores de la lógica normal del programa.

1.7.5 Recursos para profundizar

- Documentación oficial de Python sobre manejo de excepciones
- Excepciones incorporadas en Python
- Errores y Excepciones (Hektor Profe)
- Python Exceptions: An Introduction (Real Python)

1.8 Introspección y reflexión

1.8.1 Definiciones

Introspección Es la capacidad de un programa para **examinar su propia estructura y estado en tiempo de ejecución**. En Python, gracias a su naturaleza dinámica, podemos inspeccionar tipos, atributos y métodos de objetos, incluso sin conocerlos de antemano.

Reflexión Va un paso más allá: no solo inspecciona, sino que **modifica el comportamiento o la estructura** de objetos, clases o módulos en tiempo de ejecución.

Nota

En **Java** estas capacidades existen mediante *Reflection API*, mientras que en **Go** se logran con el paquete `reflect`. En Python, estas herramientas están integradas en el propio lenguaje y son mucho más accesibles.

1.8.2 Herramientas comunes de introspección y reflexión

Función	Descripción
<code>type(obj)</code>	Devuelve el tipo del objeto.
<code>dir(obj)</code>	Lista atributos y métodos disponibles.
<code>id(obj)</code>	Identificador único en memoria.
<code>vars(obj)</code>	Diccionario de atributos de instancia.
<code>getattr(obj, name[, default])</code>	Obtiene un atributo dinámicamente.
<code>setattr(obj, name, value)</code>	Asigna un atributo dinámicamente.
<code>hasattr(obj, name)</code>	Verifica si un atributo existe.
<code>callable(obj)</code>	Indica si el objeto es invocable como función o método.
<code>help(obj)</code>	Muestra la documentación.

1.8.3 Ejemplo básico de introspección

```
class Persona:
    """Clase simple para ejemplo de introspección."""

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

    def saludar(self):
        print(f"Hola, soy {self.nombre} y tengo {self.edad} años.")

# Crear instancia
persona = Persona("Alice", 30)

# Inspección
print("Tipo de objeto:", type(persona))
print("Atributos de instancia:", vars(persona))
print("ID (memoria):", id(persona))
print("¿Tiene atributo 'nombre'?", hasattr(persona, "nombre"))
print("Nombre:", getattr(persona, "nombre"))

# Modificación controlada
setattr(persona, "edad", 31)
print("Edad actualizada:", getattr(persona, "edad"))
```

```
# Verificar invocabilidad
print("¿Es 'persona' invocable?", callable(persona))
print("¿Es 'persona.saludar' invocable?", callable(persona.saludar))
```

1.8.4 Ejemplo de reflexión con cautela

En Python es posible agregar atributos o métodos a un objeto existente en tiempo de ejecución.

Esto otorga flexibilidad, pero puede volver el código difícil de mantener si se abusa.

```
import types

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

    def saludar(self):
        print(f"Hola, soy {self.nombre} y tengo {self.edad} años.")

juan = Persona("Juan", 40)
ana = Persona("Ana", 35)

# Agregar atributos en tiempo de ejecución
juan.telefono = "123-456-7890"
ana.domicilio = "La Merced 123"

# Agregar un método personalizado solo a 'juan'
def saludar_con_telefono(self):
    print(
        f"Hola, soy {self.nombre}, tengo {self.edad} años y mi teléfono es "
        f"{self.telefono}."
    )

juan.saludar = types.MethodType(saludar_con_telefono, juan)

# Uso
juan.saludar()
ana.saludar()
```

1.8.5 Uso de help() para documentación

help() es una función incorporada en Python que proporciona información sobre objetos, funciones y módulos. Es especialmente útil para obtener documentación sobre cómo usar un objeto o qué métodos y atributos tiene.

```
class Persona:
    """Clase simple para ejemplo de documentación."""

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

    def saludar(self):
        """Método para saludar."""
        print(f"Hola, soy {self.nombre} y tengo {self.edad} años.")

# Crear instancia
alice = Persona("Alice", 30)
```

```
# Usar help()
help(alice)
```

1.8.6 Uso de `eval()` para evaluar expresiones

La función `eval()` permite ejecutar expresiones Python desde una cadena de texto. Esto puede ser útil para evaluar dinámicamente código, pero **debe usarse con extrema precaución** debido a implicaciones de seguridad.

Por ejemplo, el siguiente código es peligroso si la variable `expresion` proviene de un usuario no confiable:

```
expresion = input("Introduce una expresión: ")
resultado = eval(
    expresion
) # ¡Peligroso! El usuario puede ejecutar cualquier código Python.
print("Resultado de la expresión:", resultado)
```

Advertencia de seguridad

Nunca se debe utilizar `eval()` con entradas que provengan de usuarios o fuentes no confiables. Hacerlo puede permitir la ejecución de código malicioso y comprometer la seguridad del sistema.

```
x = 10
expresion = "x * 2"
resultado = eval(expresion)
print("Resultado de la expresión:", resultado)
```

1.8.7 Buenas Prácticas

La introspección y reflexión son herramientas poderosas pero que se deben utilizar con responsabilidad para construir *código flexible* y no *código frágil*. Algunas recomendaciones son:

- Usar introspección para depuración
- No usar introspección y reflexión como sustituto de un buen diseño
- Evitar abusar de la reflexión para modificar clases/objetos, ya que reduce la previsibilidad del código.
- Documentar cambios dinámicos para facilitar mantenimiento.
- Si se requiere reflexión, documentar claramente y mantener el cambio localizado.
- En proyectos grandes, preferir patrones de diseño que hagan explícitas las extensiones (por ejemplo, decoradores).
- Preferir alternativas explícitas (herencia, interfaces, polimorfismo) cuando sea posible.

1.8.8 Recursos para profundizar

- Documentación oficial de Python sobre introspección
- Artículos sobre metaprogramación en Python (Real Python)

1.9 Archivos

En casi cualquier lenguaje de programación, interactuar con archivos es fundamental. En **Java** y **Go**, esto se hace a través de librerías específicas (`java.io`, `os`), pero en **Python** es mucho más directo y expresivo.

1.9.1 Representación de archivos y carpetas

En un sistema operativo (SO), un **archivo** es básicamente una secuencia de *bytes* almacenada en un medio físico (disco, SSD, memoria externa). Por lo general los archivos se encuentran en el disco dentro de un **sistema de archivos** o *filesystem*. Este **sistema de archivo** depende de cada sistema operativo. Por ejemplo Linux utiliza un **sistema de archivos** denominado **ext4**, mientras que Windows utiliza **NTFS**.

La principal diferencia entre distintos sistemas de archivos es como se gestionan los metadatos (información sobre el archivo, como su nombre, tamaño, permisos, etc.) y la estructura de directorios.

Archivo Contiene datos (texto, imágenes, binarios, etc.).

Carpeta (o directorio) Estructura que agrupa archivos y otras carpetas.

Para que una aplicación o programa pueda crear, leer o escribir un archivo o carpeta debe realizar una solicitud al SO, principal responsable de la gestión del hardware.

El SO responde a la petición con un **descriptor de archivo** (*file descriptor*). Este **descriptor de archivo** es un número entero que identifica de manera única un archivo abierto en ese momento.

Mientras la aplicación tiene el archivo abierto, puede leer y escribir en él. Cuando la aplicación ya no necesita más el archivo debe *cerrarlo*. Al cerrar el archivo se notifica al SO que se terminó de utilizar y por lo tanto queda disponible para que otra aplicación o programa pueda manipularlo.

1.9.1.1 Lectura y escritura

Cuando abrimos un archivo:

1. El SO localiza el archivo y asigna un *file descriptor*.
2. Python crea un **objeto archivo** que envuelve ese descriptor.
3. Las operaciones de lectura/escritura se hacen en **buffers** (bloques de memoria intermedia) para optimizar el rendimiento.

Ejemplo: si quiere leer un archivo grande, Python no trae todo de golpe, sino trozos que se van entregando al programa. Lo mismo ocurre al escribir, en lugar de escribir todo de una vez, Python lo hace en partes.

Cuando se cierra un archivo en el que se escribieron datos, Python se asegura que todos los datos se hayan escrito correctamente en el disco, volcando toda la información de los buffers.

En este contexto es fundamental usar los bloques `try/finally` para garantizar que los archivos se cierren adecuadamente, incluso si ocurre un error durante la lectura o escritura. Como el bloque `finally` se ejecuta siempre, podemos asegurarnos de que el archivo se cierre en cualquier situación, incluso si hay excepciones, y que todos los datos escritos se guarden correctamente, liberando el archivo para su uso futuro o parte de otro programa.

1.9.1.2 Archivos de texto vs archivos binarios

Texto Interpretan *bytes* según una codificación (por ejemplo UTF-8). Ejemplo: `"hola"`
`arrow.r 68 6f 6c 61` (bytes) interpretados como caracteres.

Binarios Los *bytes* se usan tal cual (imágenes, ejecutables, audio, etc.).

En Python, esto se define al abrir el archivo con `"t"` (texto) o `"b"` (binario). En general si se omite el modo, se abrirá en modo texto.

1.9.1.3 Saltos de línea: \n vs \r\n

En **Linux** y **macOS** los saltos de línea se representan con el carácter \n, mientras que en **Windows** se usan dos caracteres \r\n. Python **traduce automáticamente** al trabajar en modo texto, así que no hay que preocuparse si el programa se ejecuta en un entorno **Windows**, **Linux** o **macOS**, salvo que estemos manipulando el archivo en modo binario, donde es responsabilidad del programador.

1.9.2 Operaciones con carpetas

Para manipular carpetas y rutas, Python ofrece los módulos `os` y `pathlib`.

1.9.2.1 Rutas o *paths* absolutos y relativos

Path Absoluto Especifica toda la ruta desde la raíz. Por ejemplo: `/home/usuario/archivo.txt` o `C:\Users\Usuario\archivo.txt`

Path Relativo Se interpreta desde el directorio donde se ejecuta el programa. Por ejemplo: `datos/archivo.txt`

El carácter especial `.` representa el directorio actual, mientras que `..` representa el directorio padre, con lo cual se pueden gestionar rutas relativas de manera más sencilla.

Algunas funciones útiles del módulo `os` para manipular archivos y carpetas son:

Función	Descripción	Ejemplo
<code>os.getcwd()</code>	Obtiene el directorio de trabajo actual	<code>"/home/usuario/proyecto"</code>
<code>os.chdir(path)</code>	Cambia el directorio de trabajo actual	<code>os.chdir('/home/usuario/docs')</code>
<code>os.listdir(path)</code>	Lista archivos y carpetas en un directorio	<code>['archivo1.txt', 'carpeta1', 'imagen.png']</code>
<code>os.mkdir(path)</code>	Crea un directorio	<code>os.mkdir('nueva_carpeta')</code>
<code>os.makedirs(path)</code>	Crea directorios anidados (recursivo)	<code>os.makedirs('carpeta/subcarpeta')</code>
<code>os.rmdir(path)</code>	Elimina un directorio vacío	<code>os.rmdir('carpeta_vacia')</code>
<code>os.removedirs(path)</code>	Elimina directorios vacíos recursivamente	<code>os.removedirs('carpeta/subcarpeta')</code>
<code>os.remove(path)</code>	Elimina un archivo	<code>os.remove('archivo.txt')</code>
<code>os.rename(old, new)</code>	Renombra archivo o directorio	<code>os.rename('viejo.txt', 'nuevo.txt')</code>
<code>os.stat(path)</code>	Obtiene información del archivo (tamaño, permisos, etc.)	<code>os.stat('archivo.txt')</code>
<code>os.path.exists(path)</code>	Verifica si existe archivo o directorio	True o False
<code>os.path.isfile(path)</code>	Verifica si es un archivo	True o False
<code>os.path.isdir(path)</code>	Verifica si es un directorio	True o False
<code>os.path.join(...)</code>	Une partes de una ruta de forma portable	<code>os.path.join('carpeta', 'archivo.txt')</code>
<code>os.path.basename(path)</code>	Obtiene el nombre del archivo	<code>"archivo.txt"</code> de <code>"/ruta/archivo.txt"</code>
<code>os.path.dirname(path)</code>	Obtiene el directorio padre	<code>"/ruta"</code> de <code>"/ruta/archivo.txt"</code>
<code>os.path.splitext(path)</code>	Separa nombre y extensión	<code>('archivo', '.txt')</code>

Función	Descripción	Ejemplo
<code>os.path.abspath(path)</code>	Convierte ruta relativa a absoluta	<code>"/home/usuario/archivo.txt"</code>
<code>os.path.getsize(path)</code>	Obtiene tamaño del archivo en bytes	1024

```
import os

# Ejemplo práctico de uso
directorio_actual = os.getcwd()
print(f"Directorio actual: {directorio_actual}")

print("Archivos en directorio actual:")
for archivo in os.listdir("."):
    print(f" - {archivo}")

# Cambia al directorio temporal
os.chdir(tmp_dir)

# Crear estructura de carpetas
if not os.path.exists("datos"):
    os.makedirs("datos/procesados")
    print(f"Estructura de carpetas creada: "
          f"{os.path.abspath('datos/procesados')}")

# Trabajar con rutas
# Construir una ruta a un archivo.
# Se recomienda usar `os.path.join` y no concatenar cadenas ya que el módulo
# `os` puede construir rutas de forma portable para cualquier sistema
# operativo. Es decir este programa funcionará en cualquier sistema operativo
# sin modificaciones.
ruta_archivo = os.path.join("datos", "archivo.txt")
print(f"Ruta construida: {ruta_archivo}")
print(f"Ruta absoluta: {os.path.abspath(ruta_archivo)}")
print(f"¿Existe la ruta?: {os.path.exists(ruta_archivo)}")

os.chdir(directorio_actual)
```

1.9.2.2 Módulo `pathlib` (Recomendado para proyectos nuevos)

Python 3.4+ incluye `pathlib`, que ofrece una interfaz más moderna y orientada a objetos:

Función/Método	Descripción	Ejemplo
<code>Path.cwd()</code>	Directorio actual	<code>Path.cwd()</code>
<code>Path.home()</code>	Directorio home del usuario	<code>Path.home()</code>
<code>Path.exists()</code>	Verifica existencia	<code>Path('archivo.txt').exists()</code>
<code>Path.is_file()</code>	Verifica si es archivo	<code>Path('archivo.txt').is_file()</code>
<code>Path.is_dir()</code>	Verifica si es directorio	<code>Path('carpeta').is_dir()</code>
<code>Path.mkdir()</code>	Crea directorio	<code>Path('nueva').mkdir(parents=True)</code>
<code>Path.unlink()</code>	Elimina archivo	<code>Path('archivo.txt').unlink()</code>
<code>Path.rmdir()</code>	Elimina directorio vacío	<code>Path('carpeta').rmdir()</code>
<code>Path.iterdir()</code>	Itera sobre contenido	<code>list(Path('.').iterdir())</code>
<code>Path.glob(pattern)</code>	Busca archivos por patrón	<code>Path('.').glob('*.txt')</code>
<code>Path.chdir(path)</code>	Cambia el directorio actual	<code>Path.chdir('/nueva/ruta')</code>

```
from pathlib import Path
```

```
# Ejemplo con pathlib (más pythónico)
directorio_actual = Path.cwd()
print(f"Directorio act: {directorio_actual}")

# Crear ruta de forma elegante
archivo = Path(tmp_dir) / "datos" / "ejemplo.txt"
print(f"Ruta del archivo: {archivo}")

# Crear directorio si no existe
archivo.parent.mkdir(parents=True, exist_ok=True)
print(f"Directorio creado: {archivo.parent}")

# Buscar archivos por patrón
archivos_md = list(Path(".").glob("**/*.md"))
print(f"Archivos Markdown encontrados: {len(archivos_md)}")
```

Recomendación

Para código nuevo, se recomienda usar pathlib ya que es más legible y moderno. Para compatibilidad con código antiguo o scripts simples, `os.path` sigue siendo válido.

1.9.2.3 “Caminar” por el sistema de archivos

`os.walk()` permite recorrer todas las carpetas y archivos a partir de una ubicación dada

```
import os
from datetime import datetime

# Ejemplo de uso de os.walk()
for raiz, dirs, archivos in os.walk("."):
    # Excluir directorios ocultos (como .ipynb_checkpoints)
    dirs[:] = [d for d in dirs if not d.startswith('.')]

    print(f"Carpeta: {raiz}")
    for archivo in archivos:
        if archivo.startswith('.'):
            continue

        ruta_completa = os.path.join(raiz, archivo)

        # Obtener información del archivo
        try:
            info = os.stat(ruta_completa)

            print(f""" - {archivo}
Ruta absoluta: {os.path.abspath(ruta_completa)}

Tamaño: {info.st_size} bytes
Última modificación: {datetime.fromtimestamp(info.st_mtime)}
Permisos: {info.st_mode & 0o777:#o}
Propietario: {info.st_uid}
Grupo: {info.st_gid}
""")
        except FileNotFoundError:
            print(f" - {archivo} (No encontrado)")
```

1.9.3 Operaciones básicas sobre archivos

1.9.3.1 Apertura de archivos

La función básica para abrir archivos es `open()`:

`open(nombre, modo, encoding)`

Modo	Significado	Crea archivo si no existe	Borra contenido previo
------	-------------	---------------------------	------------------------

Modo	Significado	Crea archivo si no existe	Borra contenido previo
"r"	<i>read</i> (lectura)	No	No
"w"	<i>write</i> (escritura)	Si	Si
"a"	<i>append</i> (agregar)	Si	No
"x"	<i>exclusive write</i> (escritura exclusiva)	Si	•

El modo "x" es similar a "w", pero **lanza una excepción** si el archivo ya existe. Se usa para asegurarse que no estamos borrando el contenido de un archivo creado previamente.

El modo "a" permite agregar contenido al final del archivo sin borrar el contenido existente. Si el archivo no existe previamente, se crea.

Los modos de apertura de archivos por defecto abren los archivos como texto, si se trata de un archivo binario se debe especificar el modo "b". Por ejemplo: "rb" para lectura binaria o "wb" para escritura binaria.

[help\(open\)](#)

1.9.3.2 Lectura de archivos

Una vez abierto un archivo hay varias formas de leerlo, se puede leer completamente y cargarlo en memoria, o leerlo línea por línea, o posicionar el cursor en una parte específica del archivo para leer desde allí la cantidad de caracteres deseada, etc.

El archivo de texto que vamos a usar de prueba tiene texto en castellano y en chino tradicional, con caracteres especiales.

1.9.3.2.1 Leer todo el documento en una variable

```
archivo = os.path.join(original_cwd, "../_static/code/archivos/edd.txt")
```

```
try:
    f = open(
        archivo, "r", encoding="utf-8"
    ) # Puede levantar FileNotFoundError
except FileNotFoundError:
    print("Archivo no encontrado")
else:
    try:
        contenido = f.read() # Puede levantar otras excepciones
        print(contenido)
    except Exception as e:
        print(f"Error inesperado: {e}")
    finally:
        f.close()
```

1.9.3.2.2 Leer línea por línea en una lista

```
archivo = os.path.join(original_cwd, "../_static/code/archivos/edd.txt")
```

```
try:
    f = open(
        archivo, "r", encoding="utf-8"
    ) # Puede levantar FileNotFoundError
except FileNotFoundError:
    print("Archivo no encontrado")
else:
    try:
        contenido = f.readlines() # Puede levantar otras excepciones
        print(contenido)
    except Exception as e:
        print(f"Error inesperado: {e}")
```

```
finally:
    f.close()
```

1.9.3.2.3 Iterar línea por línea

```
# Iterar con while
archivo = os.path.join(original_cwd, "../_static/code/archivos/edd.txt")
```

```
try:
    f = open(
        archivo, "r", encoding="utf-8"
    ) # Puede levantar FileNotFoundError
except FileNotFoundError:
    print("Archivo no encontrado")
else:
    try:
        while (linea := f.readline()):
            print(linea)
    except Exception as e:
        print(f"Error inesperado: {e}")
    finally:
        f.close()
```

```
# Iterar con for
```

```
archivo = os.path.join(original_cwd, "../_static/code/archivos/edd.txt")
```

```
try:
    f = open(archivo, "r", encoding="utf-8")
    # Puede levantar FileNotFoundError
except FileNotFoundError:
    print("Archivo no encontrado")
else:
    try:
        for linea in f:
            print(linea)
    except Exception as e:
        print(f"Error inesperado: {e}")
    finally:
        f.close()
```

1.9.3.2.4 Leer una porción específica del archivo

```
archivo = os.path.join(original_cwd, "../_static/code/archivos/edd.txt")
```

```
try:
    f = open(archivo, "r", encoding="utf-8") # Puede levantar
FileNotFoundError
except FileNotFoundError:
    print("Archivo no encontrado")
else:
    try:
        f.seek(100) # Posicionar el cursor en el caracter 100
        print("Leer desde la posición 100\n")
        print(f.read(100)) # Leer 100 caracteres a partir de la posición 100
        print("\nLeer desde el inicio\n")
        f.seek(0) # Volver al inicio del archivo
        print(f.read(100)) # Leer 100 caracteres desde el inicio
    except Exception as e:
        print(f"Error inesperado: {e}")
    finally:
        f.close()
```

1.9.3.2.5 Leer un archivo binario

Si leemos el mismo archivo de texto pero en formato binario veremos dígitos en hexadecimal que se usan para representar los caracteres especiales.

```

archivo = os.path.join(original_cwd, "../_static/code/archivos/edd.txt")

try:
    f = open(archivo, "rb") # Puede levantar FileNotFoundError
except FileNotFoundError:
    print("Archivo no encontrado")
else:
    try:
        contenido = f.read() # Puede levantar otras excepciones
        print(contenido)
    except Exception as e:
        print(f"Error inesperado: {e}")
    finally:
        f.close()

```

1.9.3.3 Escritura de archivos

De forma similar a la lectura, podemos escribir en un archivo utilizando el modo de apertura "w" (*write*) o "a" (*append*). El modo "w" sobrescribe el archivo si ya existe, mientras que "a" agrega contenido al final del archivo.

1.9.3.3.1 Escribir todo el contenido de una vez

```

archivo = "texto.txt"

try:
    f = open(archivo, "w", encoding="utf-8")
except Exception as e:
    print(f"Error inesperado: {e}")
else:
    try:
        f.write("Nuevo contenido para el archivo\n")
    except Exception as e:
        print(f"Error inesperado: {e}")
    else:
        print("Contenido escrito correctamente")
    finally:
        f.close()

```

1.9.3.3.2 Escribir línea por línea

```

archivo = "texto.txt"

try:
    f = open(archivo, "w", encoding="utf-8")
except Exception as e:
    print(f"Error inesperado: {e}")
else:
    try:
        lista = []
        for i in range(5):
            lista.append(f"Línea {i}\n")
        f.writelines(lista)
    except Exception as e:
        print(f"Error inesperado: {e}")
    else:
        print("Contenido escrito correctamente")
    finally:
        f.close()

archivo = "texto.txt"

try:
    f = open(archivo, "a", encoding="utf-8")
except Exception as e:
    print(f"Error inesperado: {e}")
else:

```

```

try:
    lista = []
    for i in range(5):
        lista.append(f"Línea {i}\n")
    f.writelines(lista)
    f.write("學科基礎\n")
except Exception as e:
    print(f"Error inesperado: {e}")
else:
    print("Contenido agregado correctamente")
finally:
    f.close()

```

1.9.3.4 Entorno seguro para manipular archivos

Python provee un entorno seguro para manipular archivos con la sentencia `with` que nos asegura que siempre se cierra el archivo cuando se sale del bloque.

```

def copiar_archivo(origen, destino):
    with open(origen, "r") as f_origen, open(destino, "w") as f_destino:
        while bloque := f_origen.read(1024):
            f_destino.write(bloque)

```

```

def mostrar_archivo(archivo):
    with open(archivo, "r") as f:
        for linea in f:
            print(linea)

```

```

# Copiamos texto.txt a copia.txt
copiar_archivo(
    "texto.txt", "copia.txt"
)
mostrar_archivo("copia.txt")

```

El siguiente script nos permite comparar *byte a byte* dos archivos para ver si son iguales

```

def comparar_archivos(archivo1, archivo2):
    with open(archivo1, "rb") as f1, open(archivo2, "rb") as f2:
        while True:
            bloque1 = f1.read(1024)
            bloque2 = f2.read(1024)
            if bloque1 != bloque2:
                print("Los archivos son diferentes")
                return
            if not bloque1:
                break
        print("Los archivos son iguales")

```

```

# Comparamos texto.txt con copia.txt
comparar_archivos(
    "texto.txt", "copia.txt"
)

```

1.9.4 Recursos para profundizar

- Documentación oficial de Python sobre manejo de archivos
- Módulo `os`
- Módulo `pathlib`
- Módulo `shutil`: copia y eliminación de archivos

1.10 Persistencia de datos

En programación, **persistencia** se refiere a la capacidad de un programa para **almacenar datos más allá de su ejecución**. Cuando un programa finaliza, normalmente los datos almacenados en memoria (RAM) se pierden. Para conservarlos, es necesario guardarlos en un medio de almacenamiento **persistente** como un archivo, una base de datos o la nube.

Ejemplos comunes de persistencia:

- Guardar configuraciones de usuario en un archivo.
- Registrar el progreso de un videojuego.
- Almacenar datos temporales de un análisis para retomarlo luego.
- Guardar resultados de una simulación que tardó horas en correr.

En general, para poder persistir datos de objetos en memoria, primero se deben **serializar**.

1.10.1 ¿Qué es la serialización?

La **serialización** es el proceso de **convertir un objeto en memoria en una secuencia de bytes o en un formato estándar (como texto JSON)**, de modo que pueda:

- Guardarse en un archivo.
- Transmitirse por una red.
- Reconstruirse posteriormente en su estado original (**deserialización**).

En otras palabras:

Serialización Objeto \rightarrow Bytes/Texto (para guardar o enviar).

Deserialización Bytes/Texto \rightarrow Objeto en memoria

Sin serialización, no podríamos almacenar **objetos vivos** con un estado determinado por el valor de sus atributos y métodos en un momento dado.

1.10.2 Persistencia y serialización en Python

Python incluye varios módulos que permiten serializar y persistir datos de manera sencilla. Los más conocidos son:

pickle Serialización binaria de objetos de Python.

dill Extensión de pickle con mayor cobertura de tipos.

json Serialización en formato **texto legible** y estándar.

1.10.3 Módulo pickle

pickle convierte objetos de Python en una representación binaria que puede guardarse en un archivo o transmitirse. No es interoperable con otros lenguajes, es decir los objetos serializados y persistidos con pickle no pueden ser leídos por programas en otros lenguajes.

Como se puede intuir, efectivamente constituye una brecha de seguridad cuando se usa fuera del ámbito de una computadora privada (en redes o en Internet, por ejemplo), ya que los datos pueden ser manipulados o leídos por terceros no autorizados.

Se puede hacer un pickle con:

- None, True, False.
- Enteros, números en punto flotante y complejos.
- Cadenas, bytes, array de bytes, tuplas, listas y diccionarios que contienen sólo objetos con los que se puede hacer un pickle.
- Funciones definidas en el nivel más externo de un módulo (usando def y no lambda).
- Clases (con algunas limitaciones) definidas en el nivel más externo de un módulo.

```
import pickle
```

```
class Persona:
    def __init__(self, nombre):
```

```

        self.nombre = nombre

    def __str__(self):
        """Permite que al imprimir una instancia de Persona
        se muestre su nombre."""
        return self.nombre

if __name__ == "__main__":
    ana = Persona("Ana Suarez")
    juan = Persona("Juan Perez")
    carla = Persona("Carla Sanchez")

    with open(
        os.path.join(tmp_dir, "personas.p"), "wb"
    ) as contenedor:
        pickle.dump(ana, contenedor)
        pickle.dump(juan, contenedor)
        pickle.dump(carla, contenedor)

    with open(
        os.path.join(tmp_dir, "personas.p"), "rb"
    ) as contenedor:
        for linea in contenedor:
            print(linea)
            print()

```

En el ejemplo anterior se crean tres personas y se serializan en un archivo utilizando el módulo pickle. Luego, se lee el archivo tal como está guardado, mostrando los bytes en su forma cruda. Para deserializar los objetos, se debe usar `pickle.load()` en lugar de intentar leer el archivo directamente. `pickle.load()` se encarga de reconstruir el objeto original a partir de su representación en bytes.

```

import pickle

lista = []

with open(os.path.join(tmp_dir, "personas.p"), "rb") as contenedor:
    try:
        while contenedor:
            obj = pickle.load(contenedor)
            lista.append(obj)
    except EOFError:
        pass
    except:
        raise

for p in lista:
    print(f"Persona: {p}. Tipo: {type(p)}")

```

Ahora se modifica la clase Persona, se elimina el atributo nombre y se agrega el atributo dni.

```

import pickle

class Persona:
    """Nueva versión de la clase Persona, se agrega el atributo dni y el
    método
    get_dni"""

    def __init__(self, dni=""):
        self.dni = dni

    def __str__(self):
        """Permite que al imprimir una instancia de Persona se muestre su
        dni."""

```

```

        return self.dni

    def get_dni(self):
        return self.dni

lista = []

with open(os.path.join(tmp_dir, "personas.p"), "rb") as contenedor:
    try:
        while contenedor:
            obj = pickle.load(contenedor)
            lista.append(obj)
    except EOFError:
        pass
    except:
        raise

ana = lista[0]
juan = lista[1]
carla = lista[2]

for atributo, valor in vars(ana).items():
    print(atributo + ": " + valor)

```

Vemos que debido a que Python es un lenguaje dinámico no tiene ningún problema en leer los objetos del archivo y recrearlos en memoria como fueron serializados, aún después de que la clase `Persona` ha cambiado. Esto es posible porque `pickle` almacena la información necesaria para reconstruir el objeto, incluyendo su estructura y atributos, lo que permite que los cambios en la implementación de la clase no afecten la capacidad de deserializar objetos previamente serializados. Sin embargo, es importante tener en cuenta que si se eliminan atributos o se cambian sus tipos, esto puede causar problemas al intentar acceder a esos atributos en objetos deserializados.

1.10.3.1 Algunos detalles de la serialización con `pickle`

- De las funciones (tanto del sistema como definidas por el usuario) lo único que se conserva es su nombre, no su valor. O sea que en el momento de recuperarlas hay que tener acceso a su valor (cuerpo de la función) para poderlas ejecutar.
- Cuando se conserva una instancia de clase como `pickle`, lo único que se guardan son los valores de los atributos, no su código asociado, de modo tal que se puedan luego recuperar instancias que se crearon en versiones anteriores de la clase sin problema.

1.10.3.2 Funciones más usadas con `pickle`

1.10.4 Funciones más usadas del módulo `pickle`

Función / Elemento	Descripción breve	Documentación oficial (español)
<code>pickle.dump</code>	Serializa un objeto y lo escribe en el archivo binario. Permite opcionalmente especificar el protocolo de serialización.	Documentación de <code>dump</code>
<code>pickle.dumps</code>	Serializa un objeto, retornándolo como un objeto bytes. Ideal para enviar por red o guardar en memoria.	Documentación de <code>dumps</code>

Función / Elemento	Descripción breve	Documentación oficial (español)
<code>pickle.load</code>	Lee datos serializados desde un archivo binario y reconstruye el objeto original.	Documentación de <code>load</code>
<code>pickle.loads</code>	Reconstruye un objeto Python a partir de datos serializados en bytes.	Documentación de <code>loads</code>
<code>pickle.Pickler</code>	Clase que serializa objetos en un flujo controlado. Permite mayor control sobre el proceso de serialización.	Documentación de <code>Pickler</code>
<code>pickle.Unpickler</code>	Clase que deserializa objetos desde un flujo de datos. Proporciona control avanzado sobre el proceso de deserialización.	Documentación de <code>Unpickler</code>
Excepciones (<code>PickleError</code> , <code>PicklingError</code> , <code>UnpicklingError</code>)	Clases de excepciones específicas para errores durante la serialización y deserialización.	Documentación de excepciones en <code>pickle</code>

1.10.5 Módulo `dill`

El módulo `dill` es una extensión del módulo `pickle` que permite la serialización de una gama más amplia de objetos de Python, incluyendo funciones, funciones `lambda`, clases y módulos. Esto lo hace especialmente útil en situaciones donde se necesita serializar objetos más complejos que no son compatibles con `pickle`.

`dill` no es un módulo estándar de Python. Para utilizarlo, primero hay que instalarlo:

```
pip install dill
```

A continuación se serializa y persiste una función que en cuya clausura se encuentra un mensaje cifrado y la clave para descifrarlo.

```
import dill

def cifrar_mensaje(msj, password):
    def descifrar(x):
        if x == password:
            return msj
        else:
            return None
    return descifrar

mensaje_cifrado = cifrar_mensaje("Este es el mensaje cifrado", "secreto")

with open(os.path.join(tmp_dir, "msj_cifrado.dill"), "wb") as contenedor:
    dill.dump(mensaje_cifrado, contenedor)
```

Si leemos el archivo creado `msj_cifrado.dill`, veremos que contiene una representación binaria del objeto serializado, que incluye la función `descifrar` y su clausura con el mensaje y la clave.

```

b'\x80\x04\x95\xef\x01\x00\x00\x00\x00\x00\x00\x8c\n'
b'dill._dill\x94\x8c\x10_create_function\x94\x93\x94(h\x00\x8c\x0c_create_code\x94\x93\x94
\x00g\x00\x94N\x85\x94)\x8c\x01\x94\x85\x94\x8c"/tmp/
ipykernel_22388/1380843275.py\x94\x8c\tdescifrar\x94\x8c!
cifrar_mensaje.<locals>.descifrar\x94K\x04C\x12\xf8\x80\x00\xd8\x0b\x0c\x90\x08\x8b=\xd8\x
b'__main__\n'
b'h\x0bNh\x00\x8c\x0c_create_cell\x94\x93\x94N\x85\x94R\x94h\x15N\x85\x94R\x94\x86\x94t\x9
h\x17h.\x8c\x1aEste es el mensaje cifrado\x94\x87\x94R0.'
```

```

import dill

with open(os.path.join(tmp_dir, "msj_cifrado.dill"), "rb") as contenedor:
    mensaje_cifrado = dill.load(contenedor)

print(mensaje_cifrado("incorrecto"))
print(mensaje_cifrado("secreto"))

```

1.10.6 Serialización de objetos y la seguridad de la información

Peligro

Los módulos pickle y dill no son seguros ya que se puede construir datos maliciosos que ejecuten código arbitrario durante el proceso de deserialización. Por lo tanto, es fundamental tener precaución al utilizar estos módulos y evitar cargar datos de fuentes no confiables.

1.10.6.1 Ejemplo de riesgo de seguridad

A continuación se crea un archivo log.log en el directorio de trabajo actual para graficar como se puede ejecutar código malicioso.

```

with open(os.path.join(tmp_dir, "log.log"), "w") as f:
    f.write("Este es un archivo de registro.\n")
    f.write("La información registrada es muy sensible y se debe
resguardar\n")

```

Podemos ver que el archivo existe y se puede leer.

```

with open(os.path.join(tmp_dir, "log.log"), "r") as f:
    contenido = f.read()
    print(contenido)

```

A continuación se crea un pickle con un objeto malicioso que ejecuta un comando.

```

# Este código simula la creación de un archivo malicioso que borra log.log
import pickle
import os

```

```

class Malicioso:
    def __reduce__(self):
        return (os.remove, (os.path.join(tmp_dir, "log.log"),))

```

```

# Serializa el objeto malicioso
with open(os.path.join(tmp_dir, "malicioso.p"), "wb") as f:
    pickle.dump(Malicioso(), f)

```

Si alguien deserializa este archivo sin saber su contenido borra el archivo log.log.

```

import pickle

with open(os.path.join(tmp_dir, "malicioso.p"), "rb") as f:
    obj = pickle.load(f) # Esto borra log.log

```

Al intentar leer de nuevo el archivo log.log vemos que no existe más

```
with open(os.path.join(tmp_dir, "log.log"), "r") as f:
    contenido = f.read()
    print(contenido)
```

1.10.6.2 Funciones más usadas del módulo dill

Función / Elemento	Descripción breve	Documentación oficial (inglés)
dill.dump	Serializa un objeto y lo escribe en el archivo binario. A diferencia de pickle, soporta funciones, lambdas, generadores y más.	Documentación de dump
dill.dumps	Serializa un objeto y lo devuelve como bytes. Soporta más tipos de Python que pickle.	Documentación de dumps
dill.load	Lee datos serializados desde un archivo binario y reconstruye el objeto original. Puede restaurar funciones y objetos complejos.	Documentación de load
dill.loads	Reconstruye un objeto Python a partir de datos serializados en bytes.	Documentación de loads
dill.dump_session	Guarda el estado completo de la sesión interactiva de Python (variables, funciones, imports) en un archivo.	Documentación de dump_session
dill.load_session	Restaura una sesión previamente guardada con dump_session. Muy útil en debugging y experimentación.	Documentación de load_session
dill.detect.trace	Permite depurar el proceso de serialización mostrando qué objetos pueden o no serializarse.	Documentación de detect
Excepciones (PickleError, PicklingError, UnpicklingError)	dill reutiliza las mismas excepciones que pickle para manejar errores durante la serialización y deserialización.	Documentación de excepciones en pickle

1.10.7 Módulo json

El módulo json se basa en el estandar **JSON** (JavaScript Object Notation) y presenta un enfoque diferente al de pickle y dill, ya que se basa en texto plano y no permite la ejecución de código al deserializar. Esto lo convierte en una opción más segura para la serialización y el intercambio de datos simples, como diccionarios y listas. Los archivos JSON son legibles por humanos y pueden ser fácilmente compartidos entre diferentes lenguajes de programación.

Nota

JSON (*JavaScript Object Notation*) es un **formato de intercambio de datos basado en texto**. Se originó en el ecosistema de **JavaScript**, pero rápidamente se convirtió en un **estándar independiente del lenguaje** debido a su simplicidad y legibilidad.

Características principales de JSON:

- Está basado en una notación muy parecida a los **objetos de JavaScript**.
- Es **independiente de plataforma y lenguaje** (lo entienden Python, Java, Go, C#, etc.).
- Es **legible para humanos** y fácil de generar por máquinas.
- Es el formato más usado en **APIs REST, microservicios, configuración de aplicaciones y bases de datos NoSQL** como MongoDB.

Objetos Se representan como pares clave-valor (`{"clave": "valor"}`).

Arreglos Se representan como listas ordenadas de elementos (`["valor1", "valor2", "valorN"]`).

Valores primitivos Se representan como números, cadenas, booleanos (`true`, `false`) y `null` para `None`.

1.10.7.1 Ejemplo de uso de json

```
import json

# Datos a serializar
datos = {"nombre": "Juan", "edad": 30, "ciudad": "Madrid"}

# Serializar a JSON
with open(os.path.join(tmp_dir, "datos.json"), "w") as f:
    json.dump(datos, f)

# leer el archivo como texto
with open(os.path.join(tmp_dir, "datos.json"), "r") as f:
    contenido = f.read()
    print(contenido)

# Deserializar de JSON
with open(os.path.join(tmp_dir, "datos.json"), "r") as f:
    datos_cargados = json.load(f)
    print(datos_cargados)
```

Un archivo **JSON** solo puede contener un único diccionario o una lista, por lo que si hay que guardar múltiples objetos, se deben agrupar en una lista.

```
import json

usuarios = [
    {"nombre": "Ana", "edad": 25},
    {"nombre": "Luis", "edad": 30},
    {"nombre": "Marta", "edad": 28},
]

# Guardamos todo en un solo archivo JSON
with open(os.path.join(tmp_dir, "usuarios.json"), "w") as f:
    json.dump(usuarios, f, indent=4)

# Recuperamos
with open(os.path.join(tmp_dir, "usuarios.json"), "r") as f:
    lista_usuarios = json.load(f)

print(lista_usuarios)
print(lista_usuarios[0]["nombre"]) # Acceso al primer usuario
```

Nota

Existe un formato derivado de **JSON** denominado **JSONL** (*JSON Lines*), que consiste en una serie de objetos JSON separados por saltos de línea. Es útil para el procesamiento de grandes volúmenes de datos, ya que permite leer y escribir un objeto a la vez.

Este enfoque es muy usado en big data y machine learning, porque permite procesar el archivo registro por registro sin necesidad de cargarlo entero en memoria.

1.10.7.2 Funciones más usadas del módulo json

Función / Elemento	Descripción breve	Documentación oficial (español)
<code>json.dump</code>	Serializa un objeto Python y lo escribe en un archivo en formato JSON. Opcionalmente permite configurar indentación y codificación.	Documentación de dump
<code>json.dumps</code>	Serializa un objeto Python y lo devuelve como una cadena de texto JSON.	Documentación de dumps
<code>json.load</code>	Lee un archivo JSON y lo convierte en el objeto Python correspondiente (diccionarios, listas, etc.).	Documentación de load
<code>json.loads</code>	Convierte una cadena de texto JSON en el objeto Python correspondiente.	Documentación de loads
<code>json.JSONEncoder</code>	Clase que define cómo convertir objetos Python en JSON. Se puede extender para serializar tipos personalizados.	Documentación de JSONEncoder
<code>json.JSONDecoder</code>	Clase que define cómo convertir JSON en objetos Python. Se puede extender para deserializar estructuras personalizadas.	Documentación de JSONDecoder
Excepciones (<code>JSONDecodeError</code>)	Excepción que se lanza cuando un documento JSON no tiene el formato correcto.	Documentación de JSONDecodeError

1.10.8 Tabla Comparativa: pickle vs dill vs json

Característica	<code>pickle</code>	<code>dill</code>	<code>json</code>
Formato	Binario	Binario	Texto (legible por humanos)
Compatibilidad	Solo Python	Solo Python	Multilenguaje (estándar mundial)
Tipos soportados	Objetos de Python (casi todos)	Objetos de Python (incluye funciones, lambdas, generadores)	Tipos básicos (dict, list, str, int, float, bool, null)

Característica	pickle	dill	json
Seguridad al deserializar	Riesgo de ejecutar código malicioso	Riesgo de ejecutar código malicioso	Seguro (no ejecuta código)
Legibilidad	No legible (binario)	No legible (binario)	Legible (formato JSON)
Usos comunes	Persistencia local de objetos	Persistencia avanzada, guardar funciones	Intercambio de datos entre sistemas, APIs
Ventaja principal	Fácil y rápido para Python	Más flexible que pickle	Estándar universal, interoperable
Desventaja principal	No interoperable, inseguro	Igual que pickle (pero más pesado)	No soporta objetos complejos de Python

1.10.9 Organización de los datos

Si bien pickle y dill permiten guardar objetos complejos de Python, y json se limita a estructuras de datos más simples, en todos los casos los datos se almacenan como **un único objeto serializado por archivo**. Esto funciona bien para persistir estructuras completas (listas, diccionarios, clases), pero puede resultar incómodo cuando se quiere manejar una colección de objetos con acceso directo mediante una **clave**.

Para resolver esto, Python ofrece módulos como shelve y dbm, que permiten organizar la información de manera similar a una **base de datos ligera de pares clave-valor**, sin necesidad de instalar un gestor externo.

shelve Permite almacenar objetos de Python en un archivo de forma similar a un diccionario persistente. Se accede a los datos por clave, y cada valor puede ser un objeto complejo serializado automáticamente con pickle. Es muy útil cuando se quieren mantener estructuras de datos de Python sin necesidad de escribir el proceso de serialización/deserialización manualmente.

dbm Proporciona acceso a una familia de bases de datos simples, en las que cada clave se asocia a un valor binario. A diferencia de shelve, en dbm tanto las claves como los valores deben ser **cadenas de bytes** (bytes). Es más básico y portable, pero no admite directamente objetos de Python, sólo datos crudos en forma de texto o binario.

1.10.10 Módulo shelve

Una analogía útil

Los pickles son conservas que se guardan en frascos; a esos frascos se les puede agregar eneldo (dill) para darles más sabor, y finalmente todos los frascos se organizan y etiquetan en una estantería (shelve).

Un shelve actúa como un diccionario persistente en disco, permitiendo almacenar y recuperar objetos de Python utilizando claves. Esto facilita la gestión de colecciones de objetos sin necesidad de preocuparse por la serialización manual.

```
import shelve
```

```
# Abrir (o crear) una "base de datos"
```

```
with shelve.open(os.path.join(tmp_dir, "estudiantes.db")) as db:
    db["123"] = {"nombre": "Ana", "carrera": "Ingeniería Informática"}
    db["456"] = {"nombre": "Luis", "carrera": "Computación"}
```

```
# Recuperar los datos
```

```
with shelve.open(os.path.join(tmp_dir, "estudiantes.db")) as db:
    print(db["123"]) # {'nombre': 'Ana', 'carrera': 'Ingeniería Informática'}
```

1.10.10.1 Ventajas de shelve

- Se maneja como un diccionario común de Python.
- Permite almacenar objetos complejos sin preocuparse por serialización.
- Persistencia automática en disco.

1.10.10.2 Limitaciones

- No es seguro para acceso concurrente desde múltiples procesos.
- No es portable entre diferentes versiones de Python (ya que usa internamente pickle).

1.10.11 Módulo dbm

El módulo dbm implementa una base de datos clave-valor simple, con distintas variantes (dbm.gnu, dbm.ndbm, etc.) dependiendo del sistema. Cada entrada se almacena como una clave y un valor, ambos en forma de cadenas de bytes. Esto lo hace más ligero y portable, pero también más limitado en cuanto a los tipos de datos que puede manejar.

```
import dbm

# Crear y guardar pares clave-valor
with dbm.open(os.path.join(tmp_dir, "usuarios"), "c") as db:
    db["ana"] = "ingenieria"
    db["luis"] = "computacion"

# Recuperar datos
with dbm.open(os.path.join(tmp_dir, "usuarios"), "r") as db:
    print(db["ana"].decode("utf-8")) # "ingenieria"
    print(db["luis"].decode("utf-8")) # "computacion"
```

1.10.11.1 Ventajas de dbm

- Muy rápido y ligero.
- Ideal para guardar pares clave-valor simples (cadenas).
- Compatible con múltiples implementaciones de bases de datos en sistemas Unix.

1.10.11.2 Limitaciones de dbm

- Solo admite bytes como claves y valores.
- No guarda estructuras complejas de Python (habría que serializarlas manualmente).
- Menos flexible que shelve.

1.10.12 Ejemplo de una agenda con shelve y pickle

Ejemplo de una agenda simple que permite gestionar contactos con nombre, apellido, correos electrónicos y teléfonos. Para copiar, modificar y ejecutar:

```
"""
Agenda persistente usando shelve y pickle.

Permite agregar, buscar y eliminar contactos con múltiples teléfonos y correos.
"""

import shelve
import pickle

class Contacto:
    """
    Representa un contacto de la agenda.

    Atributos:
        nombre (str)
        apellido (str)
        correos (list[str])
        telefonos (list[str])
    """
```

```

def __init__(self, nombre, apellido, correos=None, telefonos=None):
    self.nombre = nombre
    self.apellido = apellido
    self.correos = correos if correos else []
    self.telefonos = telefonos if telefonos else []

def __str__(self):
    return (
        f"{self.nombre} {self.apellido}\n"
        f"  Correos: {'', '.join(self.correos)}\n"
        f"  Teléfonos: {'', '.join(self.telefonos)}"
    )

def agregar_contacto(agenda, contacto):
    """Agrega un contacto a la agenda usando nombre+apellido como clave."""
    clave = f"{contacto.nombre.lower()}_{contacto.apellido.lower()}"
    agenda[clave] = pickle.dumps(contacto)
    print("Contacto agregado.")

def buscar_contacto(agenda, nombre, apellido):
    """Busca un contacto por nombre y apellido."""
    clave = f"{nombre.lower()}_{apellido.lower()}"
    if clave in agenda:
        contacto = pickle.loads(agenda[clave])
        print(contacto)
    else:
        print("Contacto no encontrado.")

def eliminar_contacto(agenda, nombre, apellido):
    """Elimina un contacto por nombre y apellido."""
    clave = f"{nombre.lower()}_{apellido.lower()}"
    if clave in agenda:
        del agenda[clave]
        print("Contacto eliminado.")
    else:
        print("Contacto no encontrado.")

def listar_agenda(agenda):
    """Muestra todos los contactos de la agenda."""
    if not agenda:
        print("Agenda vacía.")
        return
    for clave in agenda:
        contacto = pickle.loads(agenda[clave])
        print(contacto)
        print("-" * 30)

def menu():
    """
    Muestra el menú principal de la agenda.
    """
    with shelve.open("agenda_db") as agenda:
        while True:
            print("\n--- Agenda ---")
            print("1. Agregar contacto")
            print("2. Buscar contacto")
            print("3. Eliminar contacto")
            print("4. Listar agenda")
            print("0. Salir")
            opcion = input("Opción: ")

```

```

if opcion == "1":
    nombre = input("Nombre: ")
    apellido = input("Apellido: ")
    correos = input("Correos (separados por coma): ").split(",")
    telefonos = input("Teléfonos (separados por coma): ").split(",")

    contacto = Contacto(
        nombre,
        apellido,
        [c.strip() for c in correos if c.strip()],
        [t.strip() for t in telefonos if t.strip()],
    )
    agregar_contacto(agenda, contacto)
elif opcion == "2":
    nombre = input("Nombre: ")
    apellido = input("Apellido: ")
    buscar_contacto(agenda, nombre, apellido)
elif opcion == "3":
    nombre = input("Nombre: ")
    apellido = input("Apellido: ")
    eliminar_contacto(agenda, nombre, apellido)
elif opcion == "4":
    listar_agenda(agenda)
elif opcion == "0":
    print("Saliendo...")
    break
else:
    print("Opción inválida.")

if __name__ == "__main__":
    menu()

```

1.10.13 Recursos para profundizar

- Working with JSON data in Python (Real Python)
- JSON de Python (W3Schools)
- Serialize your data with Python (Real Python)
- Tutorial de Pickle en Python: Serialización de objetos (DataCamp)

2. Grafos

2.1 Grafos

Un grafo es un modelo matemático que permite representar relaciones entre objetos. Un grafo está compuesto por nodos (o vértices) y aristas (o enlaces) que conectan pares de nodos.

Formalmente un grafo se define como

$$G = (V, E) \quad (2.1)$$

donde:

V Conjunto de vértices

E Conjunto de aristas. Las aristas conectan pares de vértices. Además se cumple que $E \subseteq V \times V$

$|V|$ Cardinal de V (denota la cantidad de vértices del grafo).

$|E|$ Cardinal de E (denota la cantidad de aristas).

Entre un par cualquiera de vértices sólo puede haber una arista. Por lo tanto siempre se cumple que:

$$|E| \leq |V|^2 \quad (2.2)$$

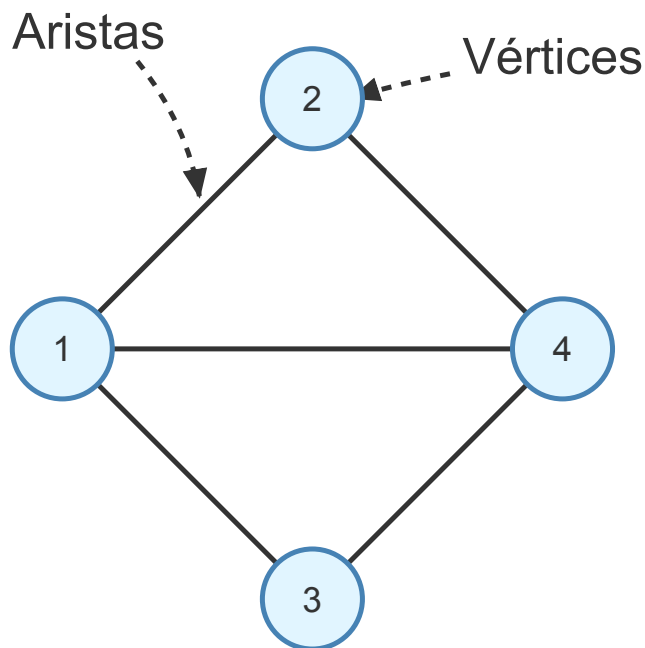


Figure 2.1: Ejemplo de un grafo

Si entre cada par de vértices se puede tener más de una arista, entonces se trata de un **multigrafo**. En esta materia no vamos a estudiar **multigrafos**.

Ver más sobre multigrafos

2.1.1 Grafos dirigidos y no dirigidos

Los grafos se pueden clasificar en dirigidos y no dirigidos. En un grafo dirigido, las aristas tienen una dirección y conectan un vértice de origen con un vértice de destino. En cambio, en un grafo no dirigido, las aristas no tienen dirección y simplemente conectan dos vértices.

Por ejemplo, en la siguiente imagen se observa las relaciones de amistad en una red social, donde las relaciones son simétricas, es decir, si A es amigo de B, entonces B es amigo de A. Estas relaciones se pueden representar con un grafo no dirigido.

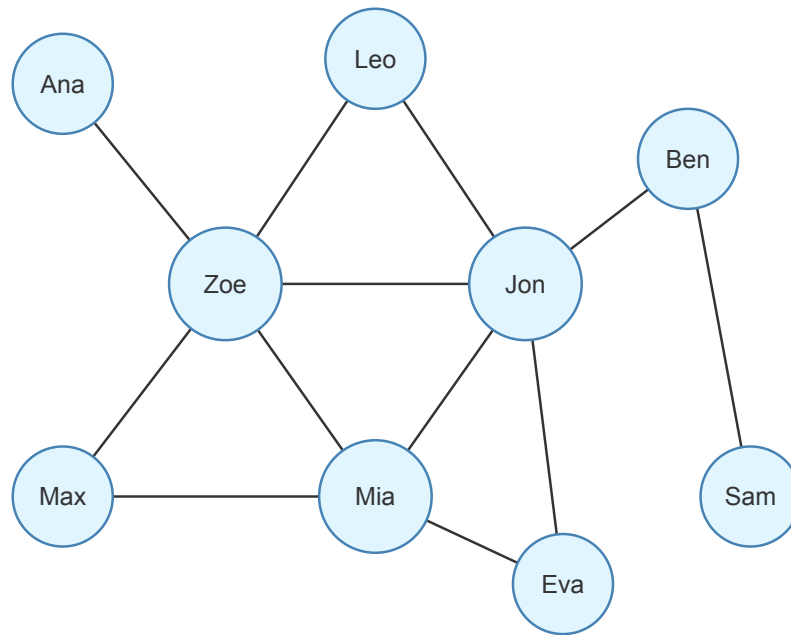


Figure 2.2: Grafo de amistades en una red social

Los grafos dirigidos permiten representar relaciones asimétricas entre dos nodos. Por ejemplo el plan de estudios de una carrera se puede modelar con un grafo dirigido, donde las materias son los nodos y las aristas indican las correlativas que se deben aprobar antes de cursar una materia.

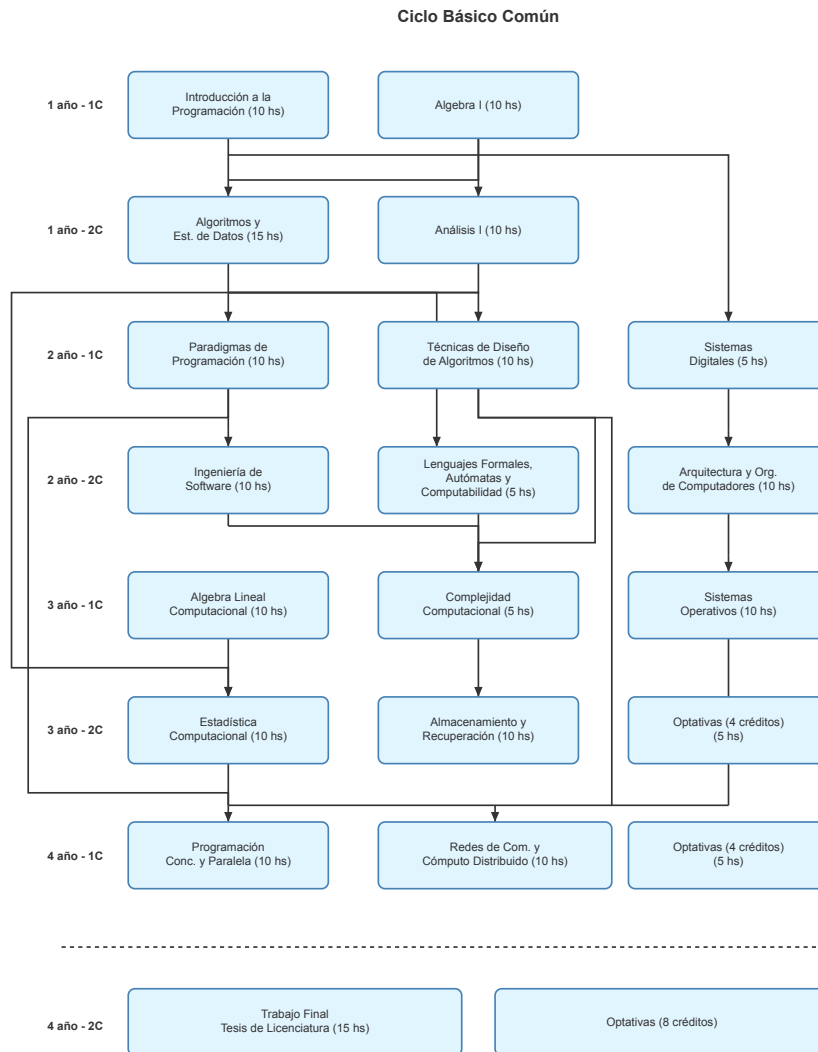


Figure 2.3: Grafo de correlativas en un plan de estudios

Los **grafos dirigidos** también se denominan **digrafos**.

Como se observa en la última figura hay vértices que no están conectados a otros vértices. Un grafo puede tener vértices “*desconectados*”. Un grafo se dice que es **conexo** si existe un camino entre cada par de vértices, es decir, para cualquier par de vértices, se puede llegar de uno al otro siguiendo las aristas del grafo. Para grafos conexos, se cumple que

$$|V| - 1 \leq |E| \leq |V|^2 \quad (2.3)$$

2.1.2 Grafos ponderados y no ponderados

Un grafo se dice que es **ponderado** si cada arista tiene un peso o costo asociado. Este peso puede representar diferentes cosas, como la distancia entre dos nodos o el tiempo necesario para recorrer una arista. Por otro lado, un grafo es **no ponderado** si sus aristas no tienen pesos.

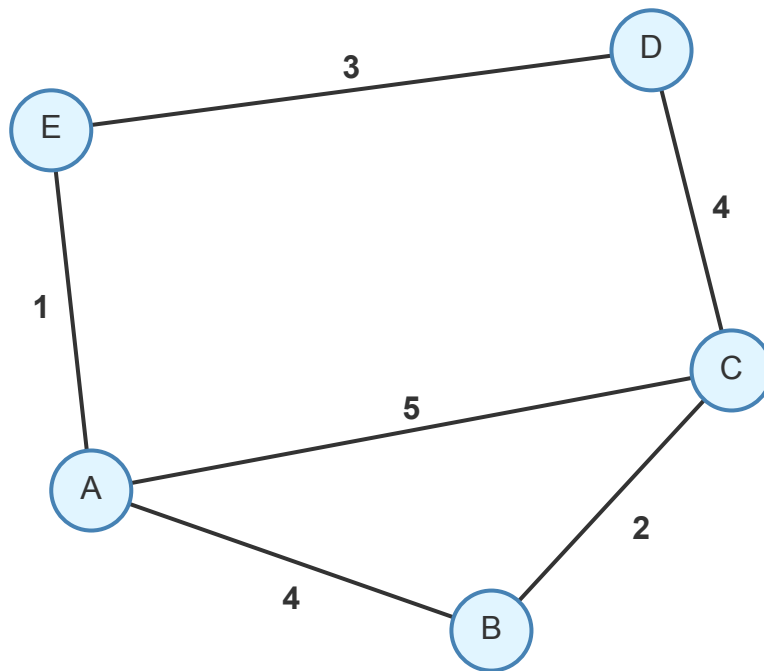


Figure 2.4: Grafo ponderado con costos en las aristas

2.1.3 Definiciones

2.1.3.1 Camino

Un camino en un grafo es una secuencia de vértices en la que cada par de vértices adyacentes está conectado por una arista. Un camino puede ser **simple** (sin vértices repetidos) o tener **ciclos** (vértices repetidos). En general cuando se habla sólo de camino se refiere a un **camino simple** sin ciclos.

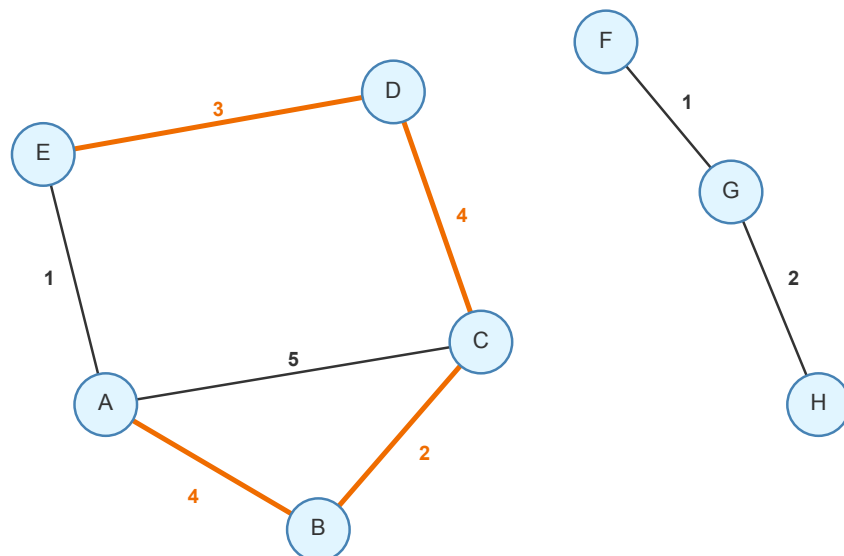


Figure 2.5: Grafo que representa un camino simple. El camino conecta los vértices A, B, C, D y E. Se observa que no hay ningún camino entre los vértices A y G por ejemplo.

2.1.3.2 Costo de un camino

El costo de un camino en un grafo ponderado es la suma de los pesos de las aristas que lo componen. En un grafo no ponderado, se puede considerar que todas las aristas tienen el mismo costo (costo de 1) y el costo del camino representa la cantidad de aristas que lo componen.

Por ejemplo, en la siguiente tabla se observa el peso de las aristas que componen el camino $A - E$:

Arista	Peso
(A, B)	4
(B, C)	2
(C, D)	4
(D, E)	3

por lo tanto el costo del camino $A - E$ es:

$$4 + 2 + 4 + 3 = 13 \quad (2.4)$$

2.1.3.3 Grafo Dirigido Acíclico

Es un grafo cuyas aristas son dirigidas y no presenta ciclos, también conocidos como **DAG** por su sigla en inglés (*Directed Acyclic Graph*). Los **DAG** son un tipo de grafo con amplias aplicaciones y se utilizan en diversas áreas como la informática, la biología, etc.

Visto de otra forma si partimos de un vértice cualquiera del grafo no existe ningún camino que permita regresar al mismo vértice.

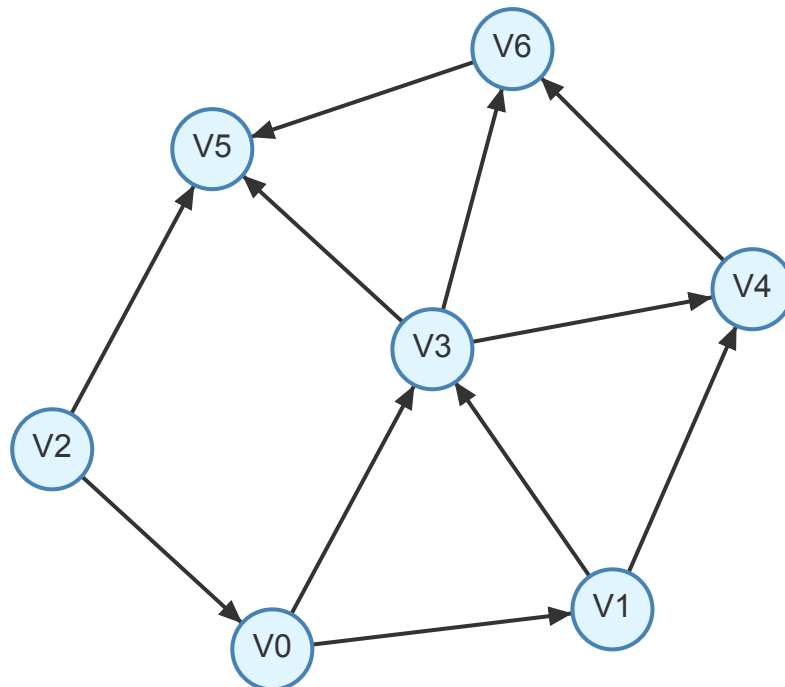


Figure 2.7: Grafo Dirigido Acíclico

2.1.3.4 Grado de entrada

El grado de entrada de un vértice en un grafo dirigido es el número de aristas que llegan a ese vértice. En otras palabras, es la cantidad de aristas entrantes que tiene un vértice.

Por ejemplo en el grafo anterior el grado de entrada del vértice V_5 es 3

2.1.3.5 Grado de salida

El grado de salida de un vértice en un grafo dirigido es el número de aristas que salen de ese vértice. En otras palabras, es la cantidad de aristas salientes que tiene un vértice.

Por ejemplo en el grafo anterior el grado de salida del vértice V_5 es 0

2.1.3.6 Fuente

Es un vértice cuyo grado de entrada es 0

2.1.3.7 Sumidero

Es un vértice cuyo grado de salida es 0

Importante

Un **DAG** siempre tiene al menos un vértice **fuentes** y un vértice **sumidero**.

En el grafo de la figura anterior, el vértice V_2 es una fuente y el vértice V_5 es un sumidero.

2.1.4 Representación de grafos

En una computadora hay al menos dos formas de representar un grafo. Usando una **lista de adyacencia** o una **matriz de adyacencia**.

2.1.4.1 Matriz de adyacencia

Dado el siguiente grafo $G = (V, A)$

donde

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\} \quad (2.5)$$

$$A = \{(V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10), (V_2, V_0, 4), (V_2, V_3, 2), (V_3, V_2, 5), (V_3, V_4, 2), (V_3, V_5, 8), (V_3, V_6, 4), (V_4, V_6, 5), (V_6, V_5, 1)\} \quad (2.6)$$

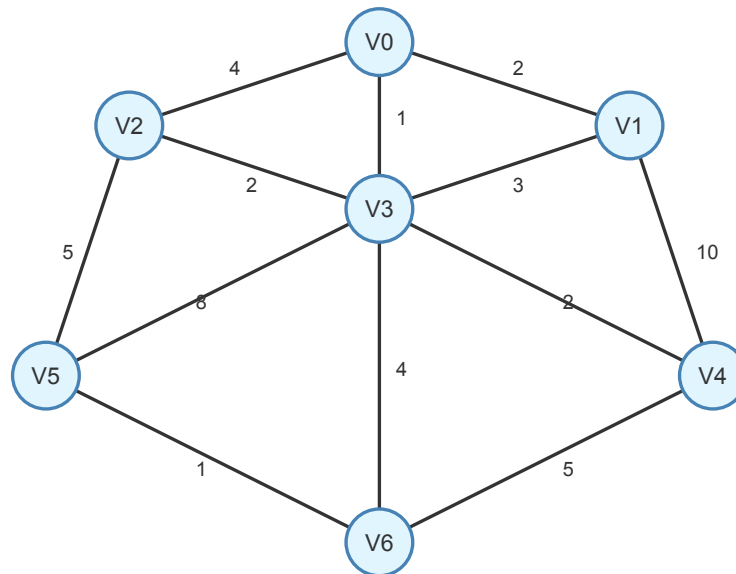


Figure 2.8: Grafo dirigido

Se numeran los vértices del grafo desde 0 hasta $n - 1$ y se genera una matriz de adyacencia M de tamaño $n \times n$ donde $M[i][j]$ representa el peso de la arista que conecta el vértice V_i

con el vértice V_j . Si no hay arista entre V_i y V_j , se puede representar con un valor especial, como ∞ o $-$.

	V_0	V_1	V_2	V_3	V_4	V_5	V_6
V_0	.	2	.	1	.	.	.
V_1	.	.	.	3	10	.	.
V_2	4	5	.
V_3	.	.	2	.	2	8	4
V_4	5
V_5
V_6	1	.

La matriz de adyacencia es una representación eficiente para grafos densos, donde el número de aristas es cercano al número máximo posible ($|V|^2$). Sin embargo, puede ser ineficiente en términos de espacio para grafos dispersos, donde el número de aristas es mucho menor que el número máximo posible. En la matriz de ejemplo solo unos pocos elementos son diferentes de '- '.

Una ventaja de esta representación es que permite verificar rápidamente si existe una arista entre dos vértices, simplemente consultando el valor en la matriz.

Para representar grafos no ponderados se acostumbra poner un 1 donde hay una arista y 0 en el resto de la matriz.

Si el grafo es no dirigido, la matriz de adyacencia será simétrica, por ejemplo la matriz de adyacencia para el siguiente grafo:

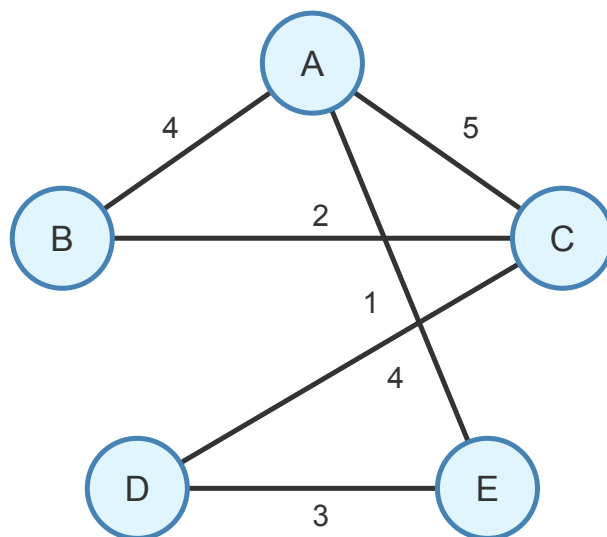


Figure 2.10: Grafo no dirigido

	A	B	C	D	E
A	.	4	5	.	1
B	4	.	2	.	.

<i>C</i>	5	2	•	4	•
<i>D</i>	•	•	4	•	3
<i>E</i>	1	•	•	3	•

2.1.4.2 Listas de adyacencias

La lista de adyacencia es otra forma de representar un grafo. En lugar de usar una matriz, se utiliza una lista de listas (o un diccionario) donde cada vértice tiene una lista de sus vecinos adyacentes y el peso de la arista que los conecta.

Para el grafo dirigido anterior la lista de adyacencia sería:

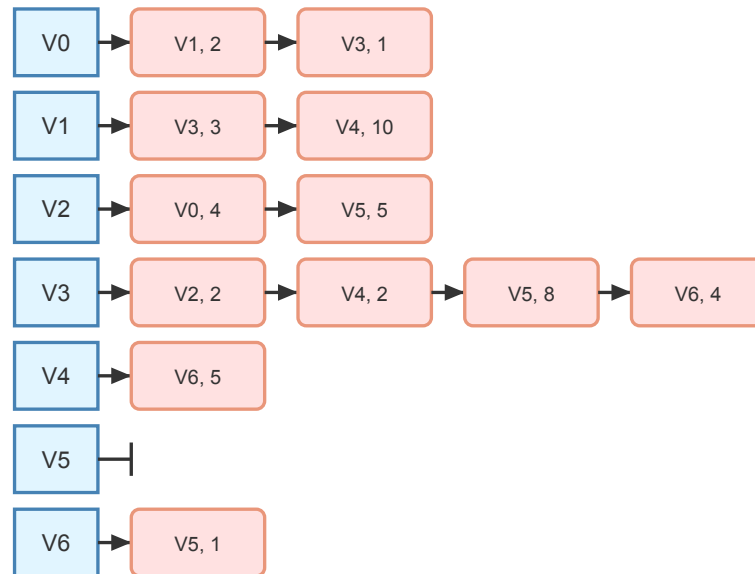


Figure 2.12: Lista de adyacencia del grafo dirigido

En cada nodo de la lista se almacena el par (*vecino*, *peso*) que representa la arista que conecta el vértice con su vecino.

A continuación la lista de adyacencia para el grafo no dirigido del punto anterior:

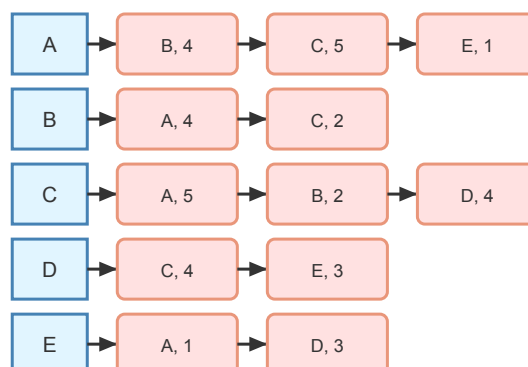


Figure 2.13: Lista de adyacencia del grafo no dirigido

2.1.5 Grafos en Python

En Python existen varias bibliotecas que facilitan la representación y manipulación de grafos. Vamos a usar **NetworkX** para representar grafos y **Matplotlib** para visualizarlos.

NetworkX proporciona estructuras de datos y algoritmos eficientes para trabajar con grafos, lo que facilita tareas como la búsqueda de caminos, la detección de ciclos y el análisis de redes.

Ambas bibliotecas se deben instalar previamente.

```
pip install networkx matplotlib
```

Los vértices deben ser de tipos *hashables*, es decir, cualquier tipo que pueda ser clave de un diccionario. Los tipos inmutables como cadenas, números o tuplas suelen ser una buena elección porque son hashables y funcionan bien como identificadores de nodos.

2.1.5.1 Crear grafos

Existen varias formas de crear un grafo. A continuación algunas de las que más usaremos:

Graph() Para crear un grafo simple, vacío y no dirigido.

DiGraph() Para crear un grafo dirigido inicialmente vacío.

Para agregar aristas se puede usar:

add_edge(u, v, weight=w) Para agregar una arista desde el nodo *u* al nodo *v* con un peso *w*. Si el grafo es no dirigido, también se agregará la arista en la dirección opuesta. Si los nodos *u* y *v* no existen en el grafo, se agregarán automáticamente.

add_weighted_edges_from(iterable) Para agregar múltiples aristas de una sola vez desde un iterable de tuplas (*u, v, w*). Si se omite el peso, se asumirá un peso de 1.

add_edges_from(iterable) Para agregar múltiples aristas de una sola vez desde iterable de tuplas (*u, v, d*). El último parámetro, opcional, puede ser un diccionario con atributos de la arista.

También se pueden agregar nodos individuales

add_node(n) Para agregar un nodo *n* al grafo.

add_nodes_from(iterable) Para agregar múltiples nodos desde un iterable. Cada elemento puede ser un nodo o una tupla de la forma (*n, d*), donde *n* es el nodo y *d* es un diccionario con los atributos opcionales.

2.1.5.2 Graficar grafos

Networkx incluye interfaces para la visualización de grafos en **Matplotlib** y **Graphviz**.

La interfaz `networkx.drawing.nx_pyplot` provee funciones que permiten crear y usar figuras con ejes de **Matplotlib**.

```
import networkx as nx
import matplotlib.pyplot as plt # permite crear figuras, mostrarlas, etc.
```

```
# Crear un grafo dirigido
G = nx.DiGraph()
```

```
# Agregar nodos y aristas
G.add_weighted_edges_from([(0, 1, 4), (0, 2, 5), (1, 2, 2), (2, 3, 4), (3, 4, 3)])
```

```
# Dibujar el grafo
# pos = nx.spring_layout(G) # posiciones para todos los nodos
pos = nx.circular_layout(G)
# pos = nx.shell_layout(G)
# pos = nx.kamada_kawai_layout(G)
nx.draw(
    G,
    pos,
```

```

with_labels=True, # etiquetar nodos
node_size=700, # tamaño de los nodos
node_color="lightblue", # color de los nodos
font_size=12, # tamaño de la fuente
edge_color="gray", # color de las aristas
width=2, # grosor de las aristas
)
edge_labels = nx.get_edge_attributes(G, "weight") # Obtener los pesos de las
aristas
nx.draw_networkx_edge_labels(
    G, pos, edge_labels=edge_labels
) # Dibujar las etiquetas de las aristas
plt.title("Grafo Dirigido") # Título de la figura
plt.show() # Mostrar la figura

```

Un **layout** es un diccionario {nodo: (x, y)} con la posición de cada vértice. **NetworkX** incluye varios algoritmos para calcular layouts.

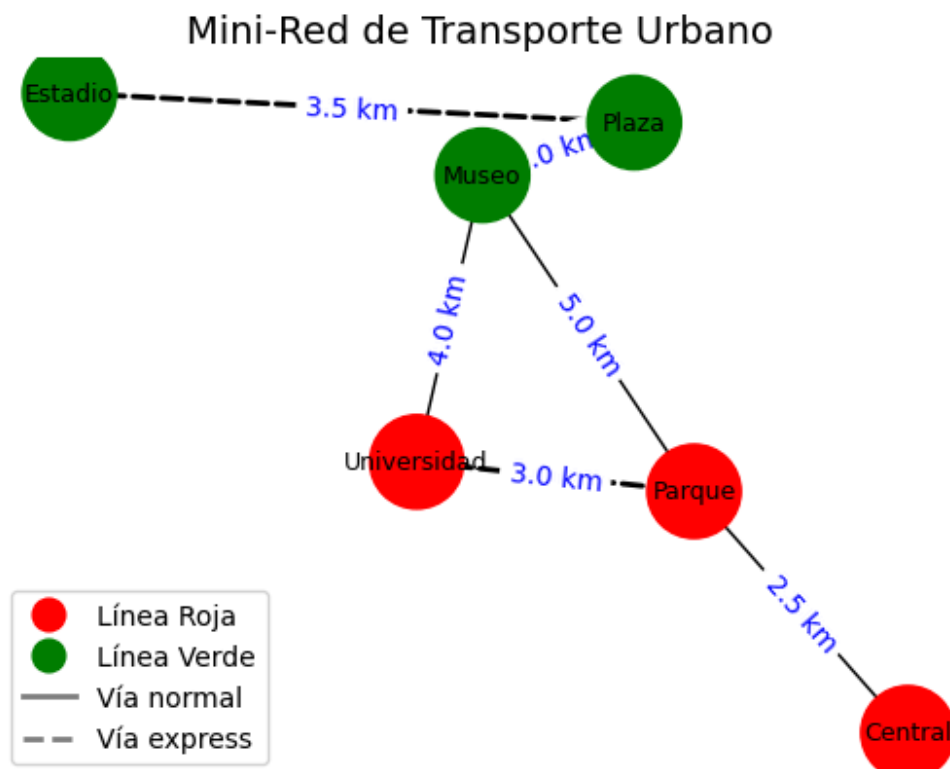
spring_layout Posiciona nodos con un modelo de fuerzas.

circular_layout Posiciona nodos en un círculo.

shell_layout Posiciona nodos en capas concéntricas.

kamada_kawai_layout Posiciona nodos minimizando la energía de un sistema de resortes.

2.1.5.3 Ejemplo: Grafo de una red de transporte



2.1.6 Recursos para profundizar

- Documentación de NetworkX
- Tutorial de Matplotlib
- Introducción a grafos y redes con Python
- NetworkX Tutorial

2.2 Recorridos

En esta sección veremos dos algoritmos para recorrer grafos. Basicamente hay dos formas de recorrer un grafo: a lo ancho y en profundidad.

Recorrer un grafo significa visitar todos sus vértices y aristas de una manera sistemática y ordenada. Los recorridos de grafos son fundamentales en muchas aplicaciones, como la búsqueda de caminos, la detección de ciclos, la planificación de tareas y la optimización de redes.

2.2.1 Recorrido a lo ancho: *Breadth First Search* (BFS)

El recorrido a lo ancho, o BFS, es un algoritmo que explora los nodos de un grafo en capas, visitando primero todos los nodos a una distancia dada antes de pasar a los nodos a una mayor distancia. Esto se logra utilizando una cola para llevar un registro de los nodos que deben ser visitados. Un ejemplo de aplicación de este recorrido lo vimos cuando estudiamos orden topológico.

```
BFS (s: Vertice):
    q <- Cola()

    q.encolar(s)
    visitado[s] = True

    MIENTRAS NO q.esta_vacia()
        v = q.desencolar()

        PARA CADA w EN v.adyacentes:
            SI NO visitado[w]:
                visitado[w] = True
                q.encolar(w)
```

2.2.2 Complejidad del algoritmo BFS

$$O(|V| + |A|) \quad (2.7)$$

2.2.3 Aplicaciones del recorrido BFS

2.2.3.1 Camino Mínimo en grafos sin pesos

```
CAMINO_MINIMO_BFS (s: Vertice)
    q <- Cola()

    distancia[s] = 0
    previo[s] = None

    q.encolar(s)
    visitado[s] = True

    MIENTRAS NO q.esta_vacia()
        v = q.desencolar()

        PARA CADA w EN v.adyacentes:
            SI NO visitado[w]:
                visitado[w] = True
                distancia[w] = distancia[v] + 1
                previo[w] = v
                q.encolar(w)
```

La ventaja de este algoritmo es que encuentra el camino más corto (mínimo número de aristas) desde el vértice s a cualquier otro vértice alcanzable desde s en $O(|V| + |A|)$.

2.2.3.2 Grafo Bipartito

Grafo Bipartito Un grafo **no dirigido** es bipartito si los vértices se pueden dividir en dos grupos, de modo tal que las aristas vayan siempre de un vértice de un grupo a un vértice del otro grupo.

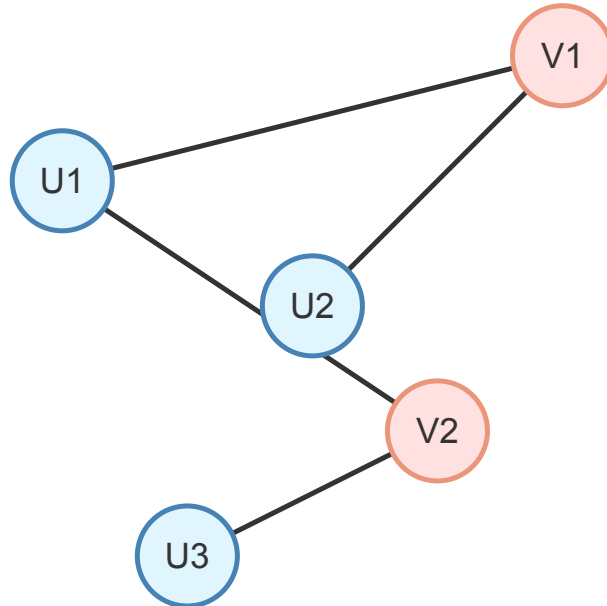


Figure 2.14: Grafo Bipartito

Reordenando los vértices de un grafo bipartito se puede ver claramente la división en dos grupos, donde las aristas van siempre de un grupo a otro y no hay aristas entre vértices del mismo grupo.

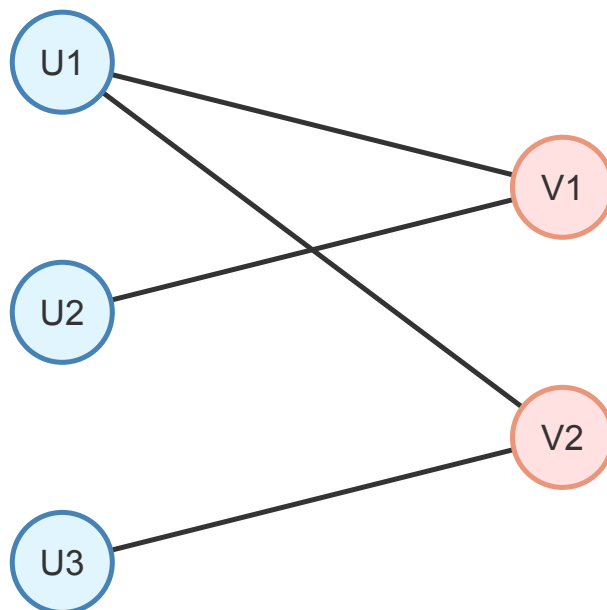


Figure 2.15: Grafo Bipartito Ordenado

```

ES_BIPARTITO (s: Vertice):
    q <- Cola()

    color[s] = True

    q.encolar(s)
    visitado[s] = True

    MIENTRAS NO q.esta_vacia()
        v = q.desencolar()

        PARA CADA w EN v.adyacentes:
            SI NO visitado[w]:
                visitado[w] = True
                color[w] = NOT color[v]
                q.encolar(w)
            SINO:
                SI color[w] == color[v]:
                    DEVOLVER False

    DEVOLVER True

```

2.2.3.3 Otras Aplicaciones

Web Crawler Bot que utilizan los motores de búsqueda para descubrir páginas siguiendo los enlaces que hay en ella.

Sistemas de navegación GPS Para encontrar localizaciones vecinas.

2.2.4 Recorrido en profundidad: *Depth First Search* DFS

El recorrido en profundidad, o DFS, es un algoritmo que explora los nodos de un grafo adentrándose lo más posible en cada rama antes de retroceder. Esto se logra utilizando una pila (o la pila de llamadas del sistema si se usa recursión) para llevar un registro de los nodos que deben ser visitados.

```

DFS (v, visitado = {}, contador = 0):
    visitado[v] = True
    contador += 1

    PARA CADA w EN v.adyacentes:
        SI NO visitado[w]:
            DFS(w, visitado, contador)

    DEVOLVER contador

```

2.2.4.1 Complejidad

$$O(|V| + |A|) \quad (2.8)$$

2.2.4.2 Aplicaciones

2.2.4.2.1 Componentes conexas

Grafo conexo Un grafo no dirigido es conexo si para todo par de vértices u y v de G , hay un camino que los une.

Componentes conexas Dado un grafo no dirigido G , una componentes conexa es un conjunto de vértices tal que empezando en uno de ellos cualquiera podemos acceder al resto recorriendo las aristas.

Componentes fuertemente conexas Dado un grafo dirigido G , una componentes fuertemente conexa es un conjunto de vértices tal que empezando en uno de ellos cualquiera podemos acceder al resto recorriendo las aristas en el sentido que indican.

```

COMPONENTES_CONEXAS (G: Grafo):
    PARA CADA v EN G.vertices:
        visitado[v] = -1

```

```
contador = 0
```

```
PARA CADA v EN G.vertices:
```

```
  SI visitado[v] == -1
```

```
    DFS(v, visitado, contador)
```

```
    contador += 1
```

```
DFS (v, visitado, contador):
```

```
  visitado[v] = contador
```

```
  PARA CADA w EN v.adyacentes:
```

```
    SI visitado[w] == -1:
```

```
      DFS(w, visitado, contador)
```

2.2.4.3 Otras aplicaciones

- Detección de ciclos en un grafo dirigido o no dirigido.
- Encontrar caminos en un laberinto.
- Encontrar componentes fuertemente connexas en un grafo dirigido (usando el algoritmo de Kosaraju o Tarjan).

2.3 Orden Topológico

El ordenamiento topológico de un grafo es un orden lineal de sus vértices tal que, para cada arista dirigida (u, v) , el vértice u aparece antes que el vértice v en el orden. Este concepto es aplicable únicamente a grafos dirigidos acíclicos (DAG).

Se utiliza para resolver problemas de planificación y dependencia de tareas, donde es necesario determinar un orden en el que se deben realizar las tareas.

2.3.1 Algoritmo de Kahn

Una forma de encontrar un ordenamiento topológico es mediante el algoritmo de Kahn (Arthur Kahn), que utiliza un enfoque basado en el grado de entrada de los vértices. Los pasos son los siguientes:

```
ORDEN TOPOLÓGICO (G: DiGrafo)
  q <- Cola vacía

  PARA CADA v EN G.nodos:
    SI v.grado_entrada == 0:
      q.encolar(v)

  MIENTRAS NO q.esta_vacia:
    v = q.desencolar()

    VISITAR v

    PARA CADA w EN v.nodos_adyacentes:
      w.grado_entrada -= 1

      SI w.grado_entrada == 0:
        q.encolar(w)

  SI quedaron nodos sin procesar:
    REPORTAR error: grafo con ciclos
```

Por ejemplo dado el siguiente grafo:

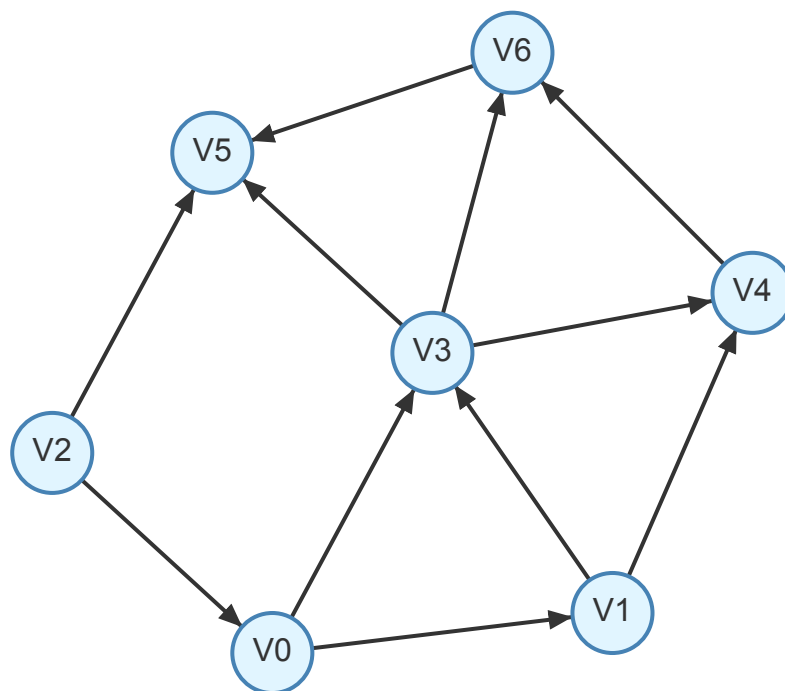


Figure 2.16: Grafo Dirigido Acíclico

Un orden topológico posible es: $V_2, V_0, V_1, V_3, V_4, V_6, V_5$. Como se observa a continuación.

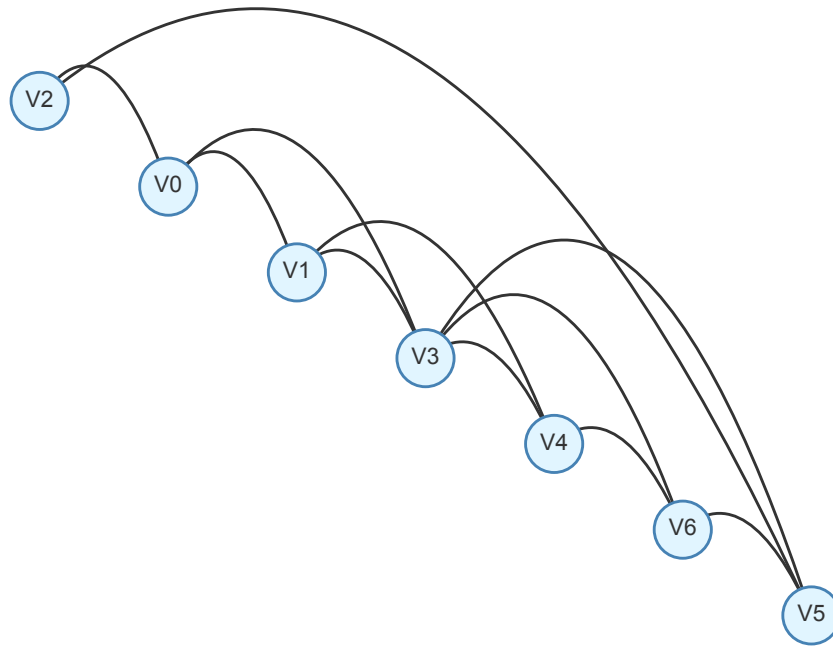


Figure 2.17: Ordenamiento Topológico

El orden topológico no es único

Pueden existir múltiples ordenamientos topológicos para un mismo grafo. El orden topológico depende de la forma en que se procesan los nodos en el algoritmo. Por ejemplo si se usa una pila en lugar de una cola se obtiene un ordenamiento diferente.

2.3.1.1 Complejidad

La complejidad del algoritmo de Kahn es $O(|V| + |E|)$. Esto se debe a que cada vértice y cada arista se procesan una sola vez.

Nota

En los algoritmos de grafos se acostumbra a utilizar tanto la cantidad de vértices $|V|$ como la cantidad de aristas $|E|$ para analizar la complejidad temporal ya que si bien se puede acotar $|E|$ en función de $|V|$, expresarlo en función de ambas variables brinda más información sobre el comportamiento del algoritmo en diferentes tipos de grafos.

2.3.2 Cálculo del orden topológico con NetworkX

NetworkX proporciona el método `topological_sort` para calcular el orden topológico de un grafo dirigido acíclico (DAG).

```
import networkx as nx

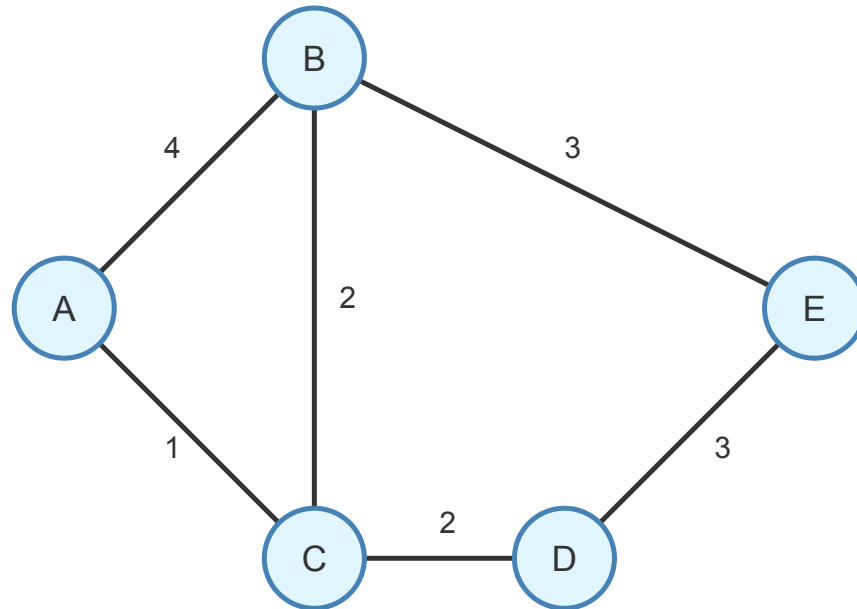
grafo = {
    "V0": ["V1", "V3"],
    "V1": ["V3", "V4"],
    "V2": ["V0", "V5"],
    "V3": ["V4", "V5", "V6"],
    "V4": ["V6"],
```

```
    "V5": [],  
    "V6": ["V5"],  
}  
  
G = nx.DiGraph(grafo)  
  
list(nx.topological_sort(G))
```

2.4 Caminos Mínimos

Un camino mínimo en un grafo es una secuencia de aristas que conecta dos vértices con el menor peso total posible. Este concepto es fundamental en diversas aplicaciones, como la navegación, la planificación de rutas y la optimización de redes.

Por ejemplo dado el siguiente grafo:



En la siguiente tabla encontramos los caminos mínimos desde el vértice *A* a los demás vértices:

Vértice	Camino Mínimo	Costo
<i>A</i>	•	0
<i>B</i>	<i>A – C – B</i>	3
<i>C</i>	<i>A – C</i>	1
<i>D</i>	<i>A – C – D</i>	3
<i>E</i>	<i>A – C – D – E</i>	6

2.4.1 Algoritmos para Encontrar Caminos Mínimos

Existen varios algoritmos para encontrar caminos mínimos en grafos, entre los más conocidos se encuentran:

Dijkstra Este algoritmo es eficiente para grafos ponderados con aristas de peso no negativo. Utiliza una cola de prioridad para explorar los vértices más cercanos al origen.

Bellman-Ford Este algoritmo puede manejar grafos con aristas de peso negativo y es útil para detectar ciclos negativos. Funciona relajando las aristas repetidamente.

Ambos algoritmos funcionan en grafos dirigidos y no dirigidos.

Cuando se trata de algoritmos sin pesos en las aristas, se acostumbra definir el camino mínimo como el que tiene la menor cantidad de aristas, lo que se logra asignando un costo de 1 a cada arista.

2.4.2 Algoritmo de Dijkstra

El **algoritmo de Dijkstra** fue propuesto por Edsger W. Dijkstra en 1956 y publicado en 1959. Es un algoritmo ávido o **greedy** que encuentra todos los caminos mínimos desde un nodo inicial a todos los demás nodos en un grafo ponderado.

Sigue una estrategia de exploración de los nodos más cercanos al origen, actualizando las distancias mínimas a medida que avanza.

La inicialización del algoritmo consiste a marcar a todos los vértices con distancia infinita, excepto el vértice de origen que se marca con distancia 0 y se encola en una **cola de prioridad de mínimos**.

En cada ciclo se extrae el vértice con la distancia más corta desde el origen, se lo marca como visitado y se exploran sus vecinos. Si se encuentra un vecino ya visitado, se ignora, ya que no se puede mejorar su distancia. A cada vecino se le actualiza su distancia si se encuentra un camino más corto y se encola.

El paso **greedy** del algoritmo consiste en seleccionar el vértice no visitado con la distancia más corta y **marcarlo como visitado**. Es decir el algoritmo considera que esa distancia no se podrá mejorar por ningún otro camino alternativo.

```
DIJKSTRA (G: DiGrafo, s: Vertice)
    pq = ColaDePrioridad()

    PARA CADA v EN G.vertices
        distancia[v] = ∞
        previo[v] = None
        visitado[v] = False

    distancia[s] = 0
    pq.encolar(s, 0)

    MIENTRAS NO pq.esta_vacia():
        v = pq.desencolar_minimo()

        visitado[v] = True

        PARA CADA w EN v.adyacentes:
            SI w no está visitado:
                SI distancia[v] + peso(v, w) < distancia[w]:
                    distancia[w] = distancia[v] + peso(v, w)
                    previo[w] = v
                    pq.encolar(w, distancia[w])
```

A continuación se muestra la aplicación del algoritmo al grafo de la figura anterior.

```
import networkx as nx
from grafos import caminos_minimos

# Definición del grafo dirigido (puedes modificarlo y ejecutar para ver el
# paso a paso)
G = nx.DiGraph()
edges = [
    ("A", "B", 4),
    ("A", "C", 1),
    ("B", "E", 3),
    ("C", "B", 2),
    ("C", "D", 2),
    ("D", "E", 3),
]
G.add_weighted_edges_from(edges)
SOURCE = "A"

caminos_minimos.show_dijkstra_step_by_step(G, SOURCE)
```

2.4.2.1 Aristas negativas

Como el **algoritmo de Dijkstra** se basa en extraer de la cola de prioridad el vértice con la menor distancia provisional desde el origen y marcarlo como **visitado** —suponiendo que esa distancia ya no podrá mejorarse—, surge un problema cuando existen aristas con peso negativo ya que podría aparecer más adelante un camino más corto hacia un vértice que ya fue marcado como **visitado**, lo que rompe el supuesto fundamental del algoritmo.

Si hay alguna arista negativa **Dijkstra** puede fallar o no según el vértice que se considere como origen y la topología del grafo, por lo tanto para asegurar que el algoritmo funciona siempre:

Importante

El algoritmo de **Dijkstra** no admite grafos con aristas con pesos negativos para poder asegurar su correcto funcionamiento

A continuación se muestra un ejemplo con una arista negativa donde el algoritmo de **Dijkstra** falla:

```
import networkx as nx
from grafos import caminos_minimos

# Definición del grafo dirigido (puedes modificarlo y ejecutar para ver el
paso a paso)
G = nx.DiGraph()
edges = [
    ("A", "B", 1),
    ("A", "C", 2),
    ("B", "E", 3),
    ("C", "B", -2),
    ("C", "D", 2),
    ("D", "E", 3),
]
G.add_weighted_edges_from(edges)
SOURCE = "A"

caminos_minimos.show_dijkstra_step_by_step(G, SOURCE)
```

El primer vértice que extrae de la cola de prioridad (paso 4) y marca como visitado es el vértice *B*. Luego al procesar *C* se encuentra un camino más corto hasta *B* con un costo total de 0, pero no puede actualizar su distancia ya que ya fue marcado como visitado (paso 7).

Si la arista (*A*, *B*) fuera la única arista negativa del grafo entonces el algoritmo no fallaría y encontraría correctamente los costos mínimos hacia todos los vértices.

2.4.2.2 Complejidad del algoritmo de Dijkstra

Si el grafo se representa mediante **listas de adyacencia**, recorrer todas las aristas del grafo tiene un costo de $O(|V| + |A|)$, que se simplifica a $O(|A|)$ cuando el grafo es conexo y $|A| \geq |V|$.

El algoritmo utiliza además una **cola de prioridad** (implementada típicamente con un **montículo binario**) donde se almacenan los vértices junto con su distancia mínima tentativa. Las operaciones críticas son:

- **extract-min** (extraer el vértice con menor distancia): costo $O(\log |V|)$, ejecutada a lo sumo $|V|$ veces para los vértices cuyo valor mínimo definitivo se extrae.
- Cada **relajación que mejora la distancia** del vértice objetivo simplemente **vuelve a encolarlo** con la nueva distancia, también con un costo de $O(\log |V|)$.

En total, cada arista se procesa a lo sumo una vez por iteración de extracción del vértice origen, y puede generar como máximo una inserción en la cola por relajación efectiva. Por tanto, el costo total del algoritmo es:

$$T(n) = O(|V| \log |V|) + O(|A| \log |V|). \quad (2.9)$$

Como en la mayoría de los grafos de interés se cumple $|A| \geq |V|$, el primer término de la suma queda dominado por el segundo y se obtiene la cota habitual:

$$T(n) = O(|A| \log |V|). \quad (2.10)$$

Otra manera de verlo: en cada iteración del bucle *MIENTRAS*, se procesan únicamente los adyacentes del vértice extraído, y a lo largo de todo el algoritmo cada arista $((u,v))$ se examina una sola vez. Formalmente:

$$\sum_{v \in V} \deg(v) = 2 |A| \Rightarrow O(|A|) \text{ relajaciones.} \quad (2.11)$$

Cada relajación puede provocar la inserción de un nuevo elemento en la cola de prioridad, lo que explica la complejidad final de $O(|A| \log |V|)$.

2.4.3 Algoritmo de Bellman-Ford

El **algoritmo de Bellman-Ford** fue desarrollado de manera independiente por dos investigadores en la década de 1950: Richard Bellman, quien lo publicó en 1958 en el marco de su trabajo sobre programación dinámica, y Lester R. Ford Jr., que lo había presentado unos años antes (1956).

Este algoritmo es una alternativa y complemento al ya conocido algoritmo de **Dijkstra** (1956) que ofrece una solución eficiente al problema de caminos mínimos desde un origen pero con la restricción de no tener aristas con pesos negativos.

En muchos problemas reales pueden aparecer aristas con pesos negativos, por ejemplo, en modelos económicos (pérdidas y ganancias), en análisis de redes de flujo o incluso en ciertas métricas de ingeniería de software o logística. En estos casos, **Bellman-Ford** ofrece una solución más general:

- Permite calcular los caminos mínimos desde un nodo origen a todos los demás, incluso con pesos negativos.
- Informa si existe un **ciclo negativo**, es decir, un ciclo en el que la suma de los pesos es menor que cero, lo que implicaría que no existe una solución bien definida para el problema de caminos mínimos (porque siempre se puede “dar una vuelta más” al ciclo y disminuir la distancia indefinidamente).

Ciclo Negativo Un ciclo negativo en un grafo dirigido y ponderado es una secuencia de aristas que comienza y termina en el mismo vértice, y cuya suma de pesos es negativa.

Formalmente $C = V_0 \rightarrow V_1 \rightarrow V_2 \rightarrow \dots \rightarrow V_k \rightarrow V_0$ es un ciclo negativo si:

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0 \quad (2.12)$$

donde $w(u, v)$ es el peso de la arista $u \rightarrow v$

Bellman-Ford se basa en la técnica de programación dinámica y en el principio de relajación de aristas:

La idea central es que, en un grafo sin ciclos negativos, el camino más corto entre dos vértices tiene a lo sumo $|V| - 1$ aristas.

El algoritmo realiza sucesivas iteraciones en las que intenta mejorar (*relajar*) las distancias conocidas, comparando si pasar por una nueva arista ofrece un camino más corto.

Supuesto clave Después de realizar $|V| - 1$ iteraciones, todas las distancias mínimas posibles estarán correctamente calculadas.

- Cada iteración (relajación) asegura que se encuentra la distancia mínima, al menos, a un vértice del grafo. En cada una de estas iteraciones se revisan todas las aristas del grafo, recalculando el costo mínimo de todos los vértices.
- En $|V| - 1$ iteraciones se habrán explorado todas las posibilidades y por lo tanto se finaliza el cálculo.
- Una iteración adicional a las $|V| - 1$ iteraciones anteriores, permite detectar ciclos negativos, ya que si la distancia de un vértice se mejora es porque hay al menos un ciclo negativo en el grafo.

BELLMAN_FORD (G: DiGrafo, s: Vertice)

```

PARA CADA v EN G.nodos
    distancia[v] = ∞
    previo[v] = None

distancia[s] = 0

REPETIR len(G.nodos) - 1 VECES:
    PARA CADA (v, w, peso) EN G.aristas:
        SI distancia[v] + peso < distancia[w]
            distancia[w] = distancia[v] + peso
            previo[w] = v

PARA CADA (v, w, peso) EN G.aristas:
    SI distancia[v] + peso < distancia[w]
        REPORTAR error: grafo con ciclos negativos

```

A continuación se muestra un ejemplo con una arista negativa donde el algoritmo de *Dijkstra* falla:

```

import networkx as nx
from grafos import caminos_minimos

# Definición del grafo dirigido (puedes modificarlo y ejecutar para ver el
paso a paso)
G = nx.DiGraph()
edges = [
    ("A", "B", 1),
    ("A", "C", 2),
    ("B", "E", 3),
    ("C", "B", -2),
    ("C", "D", 2),
    ("D", "E", 3),
]
G.add_weighted_edges_from(edges)
SOURCE = "A"

caminos_minimos.show_bellman_ford_step_by_step(G, SOURCE)

```

2.4.3.1 Complejidad del algoritmo de Bellman-Ford

La complejidad temporal del *algoritmo de Bellman-Ford* está determinada por los dos ciclos anidados de las líneas 9 y 10. El bucle *REPETIR* se ejecuta $|V| - 1$ veces y el ciclo *PARA* se ejecuta $|A|$ veces y como adentro de los ciclos todas las operaciones son $O(1)$ queda:

$$T(n) = O(|V| \times |A|) \quad (2.13)$$

2.4.4 Recursos para profundizar

- Algoritmo de Dijkstra (Wikipedia)
- Algoritmo de Bellman-Ford (Wikipedia)
- Algoritmo de Dijkstra en DSA
- Algoritmo de Bellman-Ford en DSA

3. Representación y Adquisición de Datos

3.1 Registros de datos

En esta sección vamos a ver distintos formatos para organizar la información en archivos, es decir de la organización lógica de los datos. Estos formatos son independientes del lenguaje de programación que utilicemos, y en muchos casos son independientes del software que utilicemos.

Los registros permiten estructurar la información de una forma que facilita su almacenamiento, recuperación y manipulación. Un registro es un conjunto de datos relacionados que se almacenan juntos y representan una entidad o un objeto específico. Cada registro está compuesto por varios campos, donde cada campo contiene un dato específico.

Supongamos que queremos crear una **agenda** para almacenar datos de contactos: **nombre**, **apellido**, **teléfono** y **email**. Una primera aproximación sería guardar los datos sin ningún tipo de organización, simplemente uno detrás de otro:

```
class Agenda:
    def __init__(self, archivo):
        self.archivo = archivo

    def guardar_contacto(self, nombre, apellido, telefono, email):
        with open(self.archivo, "a") as datos:
            datos.write(nombre)
            datos.write(apellido)
            datos.write(telefono)
            datos.write(email)

agenda = Agenda("agenda.txt")
agenda.guardar_contacto(
    "Ana", "Calle Falsa 123", "555-1234", "ana@example.com"
)
agenda.guardar_contacto(
    "Bart", "Calle Falsa 123", "555-5678", "bart@example.com"
)

with open("agenda.txt") as datos:
    for linea in datos:
        print(linea)
```

Si observamos el contenido del archivo `agenda.txt`, vemos que los datos están todos juntos, sin ningún tipo de organización. Se ha perdido **la integridad de los datos**, ya que no sabemos dónde termina un dato y empieza otro. Además, si queremos buscar un contacto, tenemos que leer todo el archivo y buscar el nombre, lo cual es muy ineficiente.

Existen varios formatos para organizar los registros en un archivo, veamos algunos de ellos.

3.1.1 Registros de longitud fija con campos de longitud fija

Una forma de organizar los registros es asignar una longitud fija a cada campo. Por ejemplo, podemos decidir que el campo **nombre** tendrá 20 caracteres, el campo **apellido** 30 caracteres, el campo **teléfono** 15 caracteres y el campo **email** 40 caracteres. Si un dato es más corto que la longitud asignada, se rellena con espacios en blanco o nulos. Si un dato es más largo, se trunca.

```
import os
import struct

class Agenda:
    def __init__(
        self,
```

```

archivo,
len_nombre=20,
len_apellido=30,
len_telefono=15,
len_email=40,
):
    self._archivo = archivo
    # Formato del registro: cada campo tiene longitud fija en bytes.
    # Ejemplo: "20s30s15s40s" para nombre, apellido, teléfono y email.
    # En total 105 bytes por registro.
    self._formato = "%ds%ds%ds%ds" % (
        len_nombre,
        len_apellido,
        len_telefono,
        len_email,
    )
    # Calcula la longitud total del registro en bytes.
    # struct.calcsize calcula el tamaño en bytes del formato especificado.
    self._len_registro = struct.calcsize(self._formato)

    # Calcula la cantidad de registros presentes en la agenda.
    try:
        tam_archivo = os.path.getsize(archivo)
        # Divide el tamaño del archivo por la longitud de cada registro.
        self._cant_registros = tam_archivo // self._len_registro
    except FileNotFoundError:
        self._cant_registros = 0

def guardar_contacto(
    self, nombre, apellido, telefono="", email=""
):
    """
    Guarda un registro en el archivo.
    Nombre y apellido son obligatorios.
    """
    if not nombre or not apellido:
        raise ValueError("Nombre y apellido son obligatorios")

    # Abre el archivo en modo append binario. Lo crea si no existe.
    with open(self._archivo, "ab") as registros:
        # struct.pack convierte los datos en una secuencia de bytes
        # según el formato definido. Cada campo se codifica y se
        # rellena o trunca para ocupar la cantidad de bytes
        # especificada.
        registro = struct.pack(
            self._formato,
            nombre.encode(),
            apellido.encode(),
            telefono.encode(),
            email.encode(),
        )
        # Escribe el registro al final del archivo.
        registros.write(registro)

    self._cant_registros += 1

def cantidad_registros(self):
    """Devuelve la cantidad de registros que hay en la agenda."""
    return self._cant_registros

def __iter__(self):
    """Devuelve un iterador para la agenda."""
    return AgendaIterator(self)

```

Los métodos `encode()` y `decode()` convierten entre cadenas de texto y secuencias de bytes. Por defecto utilizan la codificación UTF-8, que es capaz de representar todos los caracteres Unicode. Si se utilizan caracteres especiales (como tildes o ñ), es importante asegurarse de que la codificación sea la misma al guardar y al leer los datos.

A continuación definimos el iterador para la agenda:

```
class AgendaIterator:
    """Iterador para la agenda de registros de longitud fija"""

    def __init__(self, agenda):
        self._agenda = agenda
        self._index = 0 # Índice del registro actual

    def __iter__(self):
        return self

    def __next__(self):
        """
        Devuelve:
            dict: Un diccionario con el siguiente conjunto de datos o
                  valores en la iteración.
        """
        # Si no quedan más registros finalizamos la iteración
        if self._index >= self._agenda._cant_registros:
            raise StopIteration()

        with open(self._agenda._archivo, "rb") as registros:
            # Calcula la posición en bytes del registro actual
            posicion = self._index * self._agenda._len_registro
            registros.seek(posicion)
            # Lee el bloque de bytes correspondiente al registro
            registro = registros.read(self._agenda._len_registro)

        self._index += 1

        # Verifica que se haya leído un registro completo
        if len(registro) != self._agenda._len_registro:
            # Si el registro está corrupto, devuelve campos vacíos
            return "", "", "", ""

        # struct.unpack convierte los bytes en tuplas de campos
        # según el formato definido en la agenda
        b_nombre, b_apellido, b_telefono, b_email = struct.unpack(
            self._agenda._formato, registro
        )

        # Decodifica los bytes y elimina espacios/nulos extra
        return (
            b_nombre.decode().strip(),
            b_apellido.decode().strip(),
            b_telefono.decode().strip(),
            b_email.decode().strip(),
        )
```

Nota

El método `strip()` elimina espacios en blanco y caracteres de nueva línea al inicio y al final de una cadena. En este caso, se utiliza para eliminar los caracteres nulos (`\x00`) que se utilizan para rellenar los campos cuando son más cortos que la longitud asignada.

Ejemplo de uso con el iterador

```

agenda = Agenda("agenda_fixed.dat")
agenda.guardar_contacto("March", "Simpson", "555-1234", "march@example.com")
agenda.guardar_contacto("Bart", "Simpson", "555-5678", "bart@example.com")
agenda.guardar_contacto("Homer", "Simpson", "555-8765", "homer@example.com")
agenda.guardar_contacto("Lisa", "Simpson") # sin teléfono ni email

print(f"Cantidad de registros: {agenda.cantidad_registros()}")
for nombre, apellido, telefono, email in agenda:
    print(f"{nombre}, {apellido}\n")
    print(f"Teléfono: {telefono}\n")
    print(f"Email: {email}\n")
    print("-----")

```

Si observamos el contenido del archivo `agenda_fixed.dat` con un editor hexadecimal, vemos que los datos están organizados en bloques de longitud fija, y cada campo ocupa la cantidad de bytes asignada, rellenando con nulos si es necesario.

```

with open("agenda_fixed.dat", "rb") as f:
    contenido = f.read()
    print(contenido)

print(f"Longitud del registro: {agenda._len_registro} bytes")
print(f"Formato del registro: {agenda._formato} (105 bytes)")
print(f"Cantidad de registros: {agenda.cantidad_registros()}")
print(
    f"Cantidad de bytes en el archivo: {len(contenido)} "
    f"(105 * {agenda.cantidad_registros()})"
)

```

Tampoco permite almacenar datos que superen la longitud asignada, ya que se truncan. Por ejemplo, si intentamos guardar un nombre con más de 20 caracteres, se perderán los caracteres adicionales.

3.1.2 Registros de longitud fija y campos de longitud variable

Otra forma de organizar los registros es asignar una longitud fija a cada registro, pero permitir que los campos tengan longitud variable. Para ello, se puede utilizar un delimitador para separar los campos dentro del registro. Por ejemplo, podemos utilizar el carácter `|` como delimitador. Este tipo de organización es más eficiente en términos de espacio, ya que no se desperdicia espacio si los campos son más cortos que la longitud asignada. Sin embargo, tiene la desventaja de que no se pueden almacenar registros que superen la longitud asignada, ya que se truncan.

Nota

El carácter delimitador no puede aparecer en los datos, ya que se interpretaría como el final de un campo. Si es necesario que este carácter sea parte de los datos, se puede utilizar un mecanismo de escape, como por ejemplo, duplicar el carácter (`||` se interpreta como un solo `|` en los datos).

```

class Agenda:
    def __init__(self, archivo, campos, len_registro=100):
        self._archivo = archivo
        self._len_registro = len_registro
        self._campos = campos # lista de nombres de campos
        try:
            tam_archivo = os.path.getsize(archivo)
            self._cant_registros = tam_archivo // self._len_registro
        except FileNotFoundError:
            self._cant_registros = 0

    def guardar_contacto(self, *valores):
        """
        Guarda un registro en el archivo.

```

```

La cantidad de valores debe coincidir con la cantidad de campos.
"""
if len(valores) != len(self._campos):
    raise ValueError("Cantidad de valores incorrecta")

if not valores[0] or not valores[1]:
    raise ValueError("Nombre y apellido son obligatorios")

# Une los valores usando '|' como delimitador
registro = "|".join(str(valor) for valor in valores)

# Verifica que el registro no supere la longitud máxima
if len(registro.encode()) > self._len_registro:
    raise ValueError("El registro es demasiado largo")

# Convierte el registro a bytes y lo rellena con nulos si es necesario
registro = registro.encode()
registro = registro.ljust(self._len_registro, b"\x00")

# Escribe el registro al final del archivo
with open(self._archivo, "ab") as registros:
    registros.write(registro)

self._cant_registros += 1

def cantidad_registros(self):
    """Devuelve la cantidad de registros que hay en la agenda"""
    return self._cant_registros

def __iter__(self):
    """Devuelve un iterador para la agenda"""
    return AgendaIterator(self)

```

Nota

registro.encode() convierte la cadena a bytes, y ljust rellena con nulos a la derecha, hasta alcanzar la longitud fija del registro. Algunos caracteres como vocales con tildes o la letra ñ pueden ocupar más de un byte (por ejemplo é se codifica como b'\xc3\xa9'), por lo que es importante medir la longitud en bytes y no en caracteres (línea 27).

A continuación definimos el iterador para la agenda:

```

class AgendaIterator:
    """Iterador para la agenda de registros de longitud fija y campos de
    longitud variable
    """

    def __init__(self, agenda):
        self._agenda = agenda
        self._index = 0 # Índice del registro actual

    def __iter__(self):
        return self

    def __next__(self):
        """
        Devuelve:
            dict: Un diccionario con el siguiente conjunto de datos o valores
                en la iteración.
        """
        # Si no quedan más registros finalizamos la iteración
        if self._index >= self._agenda._cant_registros:
            raise StopIteration()

        with open(self._agenda._archivo, "rb") as registros:

```

```

# Calcula la posición en bytes del registro actual.
# Cada registro ocupa self._agenda._len_registro bytes,
# por lo tanto la posición es índice * longitud_registro.
posicion = int(self._index * self._agenda._len_registro)
registros.seek(posicion)
registro = registros.read(self._agenda._len_registro)

self._index += 1

if len(registro) != self._agenda._len_registro:
    # Registro corrupto
    return dict((campo, "") for campo in self._agenda._campos)

registro = registro.rstrip(b"\x00").decode(errors="replace")
campos = registro.split("|")

# Rellenar campos faltantes con cadenas vacías
while len(campos) < len(self._agenda._campos):
    campos.append("")

# Devuelve un diccionario con claves nombre de los campos
return dict(zip(self._agenda._campos,
                campos[: len(self._agenda._campos)]))

```

Nota

El método `rstrip(b'\x00')` elimina los caracteres nulos (`\x00`) al final de la cadena de bytes, que se utilizan para rellenar el registro cuando es más corto que la longitud asignada. Luego, `decode(errors="replace")` convierte los bytes a una cadena de texto, reemplazando cualquier byte inválido con el carácter de reemplazo (`?`).

La función `zip` combina dos listas en una lista de tuplas, donde cada tupla contiene un elemento de cada lista. En este caso, se utiliza para combinar la lista de nombres de campos con la lista de valores correspondientes, y luego se convierte en un diccionario.

Ejemplo de uso con el iterador

```

agenda = Agenda("agenda.dat", ["nombre", "apellido", "telefono", "email"])
agenda.guardar_contacto("Juan", "Pérez", "123456789", "juan@example.com")
agenda.guardar_contacto("Ana", "Gómez", "987654321", "ana@example.com")

```

```

for contacto in agenda:
    print(contacto)
    print("-----")

```

Si observamos el contenido del archivo `agenda.dat` con un editor hexadecimal, vemos que los datos están organizados en bloques de longitud fija, y cada campo está separado por el carácter `|`, rellenando con nulos si es necesario.

```

with open("agenda.dat", "rb") as f:
    contenido = f.read()
    print(contenido)

print(f"Longitud del registro: {agenda._len_registro} bytes")
print(f"Cantidad de registros: {agenda.cantidad_registros()}")
print(
    f"Cantidad de bytes en el archivo: {len(contenido)} "
    f"({agenda._len_registro} * {agenda.cantidad_registros()})"
)

```

3.1.3 Registros de longitud variable y campos de longitud variable

Para implementar este tipo de registro se puede preceder cada registro con un entero que indique la longitud del registro en bytes. De esta forma, al leer el archivo, se lee primero la

longitud del registro y luego se lee el registro completo. Análogamente, se puede preceder cada campo con un entero que indique la longitud del campo en bytes.

```
import struct
import os

class Agenda:
    def __init__(self, archivo, campos):
        self._archivo = archivo
        self._campos = campos # lista de nombres de campos

    try:
        tam_archivo = os.path.getsize(archivo)
        self._cant_registros = 0

        with open(archivo, "rb") as f:
            pos = 0
            # Recorre el archivo desde el inicio hasta el final
            while pos < tam_archivo:
                f.seek(pos)
                # Los primeros 4 bytes indican la longitud del registro
                # 4 bytes para un entero sin signo (unsigned int)
                len_bytes = f.read(4)

                if len(len_bytes) < 4:
                    break # fin de archivo o registro corrupto

                # Convierte los 4 bytes en un int (longitud del registro)
                (len_registro,) = struct.unpack("I", len_bytes)
                # Avanza la pos: 4 bytes + longitud del registro
                pos += 4 + len_registro
                # Incrementa el contador de registros
                self._cant_registros += 1
    except FileNotFoundError:
        # Si el archivo no existe, no hay registros
        self._cant_registros = 0

    def guardar_contacto(self, *campos):
        """
        Guarda un registro en el archivo.
        La cantidad de campos debe coincidir con la definición.
        """
        if len(campos) != len(self._campos):
            raise ValueError(
                "La cantidad de campos no coincide con la " "definición"
            )

        registro = b""
        for campo in campos:
            campo_bytes = campo.encode()
            len_campo = len(campo_bytes)
            # struct.pack("I", len_campo) convierte el entero en 4 bytes
            # Luego se concatenan los 4 bytes de longitud y los bytes del
            registro += struct.pack("I", len_campo) + campo_bytes
        len_registro = len(registro)

        with open(self._archivo, "ab") as registros:
            # Se concatenan los 4 bytes de longitud y los bytes del registro
            registros.write(struct.pack("I", len_registro))
            registros.write(registro)

        self._cant_registros += 1

    def cantidad_registros(self):
```

```

        """Devuelve la cantidad de registros que hay en la agenda"""
        return self._cant_registros

    def __iter__(self):
        """Devuelve un iterador para la agenda"""
        return AgendaIterator(self)

```

A continuación definimos el iterador para la agenda:

```

class AgendaIterator:
    """Iterador para la agenda de registros de longitud variable"""

    def __init__(self, agenda):
        self._agenda = agenda
        self._pos = 0
        self._tam_archivo = os.path.getsize(self._agenda._archivo)

    def __iter__(self):
        return self

    def __next__(self):
        """
        Devuelve:
            dict: Un diccionario con el siguiente conjunto de datos o
            valores en la iteración.
        """
        # Si quedan registros por leer
        if self._pos >= self._tam_archivo:
            raise StopIteration()

        with open(self._agenda._archivo, "rb") as f:
            f.seek(self._pos)
            len_bytes = f.read(4)

            if len(len_bytes) < 4:
                raise StopIteration()

            (len_registro,) = struct.unpack("I", len_bytes)
            registro_bytes = f.read(len_registro)

            if len(registro_bytes) < len_registro:
                raise StopIteration()

            self._pos += 4 + len_registro
            campos = []

            # offset es la posición dentro del registro
            offset = 0
            while (offset < len_registro) and \
                (len(campos) < len(self._agenda._campos)):
                len_campo_bytes = registro_bytes[offset : offset + 4]

                if len(len_campo_bytes) < 4:
                    break

                (len_campo,) = struct.unpack("I", len_campo_bytes)
                offset += 4
                campo_bytes = registro_bytes[offset : offset + len_campo]
                campo = campo_bytes.decode()
                campos.append(campo)
                offset += len_campo

            # Rellenar campos faltantes con cadenas vacías
            while len(campos) < len(self._agenda._campos):
                campos.append("")

```

```
return dict(zip(self._agenda._campos, campos))
```

Ejemplo de uso con el iterador

```
campos = ["nombre", "apellido", "telefono", "email"]
agenda = Agenda("agenda_var.dat", campos)
agenda.guardar_contacto("Juan", "Pérez", "123456789",
                        "juan.perez@example.com")
agenda.guardar_contacto("Ana", "Gómez", "987654321", "ana.gomez@example.com")
agenda.guardar_contacto("Homero", "Simpson", "555-8765", "") # sin email
agenda.guardar_contacto(
    "Lisa", "Simpson", "", "lisa.simpson@example.com"
) # sin teléfono

for contacto in agenda:
    print(contacto)
    print("-----")
```

Si observamos el contenido del archivo `agenda_var.dat` con un editor hexadecimal, vemos que los datos están organizados en bloques de longitud variable, y cada campo está precedido por un entero que indica la longitud del campo en bytes.

```
with open("agenda_var.dat", "rb") as f:
    contenido = f.read()
    print(contenido)

print(f"Cantidad de bytes en el archivo: {len(contenido)}")
print(f"Cantidad de registros: {agenda.cantidad_registros()}")
```

Esta forma de organizar los registros es la más flexible, ya que permite almacenar datos de cualquier longitud sin desperdiciar espacio. Sin embargo, tiene la desventaja de que el acceso a los registros es secuencial, ya que no se puede calcular la posición de un registro en función de su índice. Además, la implementación es más compleja, ya que se deben manejar las longitudes de los registros y campos.

3.1.4 Otras formas de organizar registros

Registros de longitud variable y campos de longitud fija Cada campo tiene una longitud fija, pero el registro puede tener una longitud variable. Se puede utilizar un delimitador para separar los campos dentro del registro, y preceder el registro con un entero que indique la longitud del registro en bytes.

Usar índices para acceder rápidamente a los registros Se puede crear un archivo de índices que contenga la posición de cada registro en el archivo de datos. De esta forma, se puede acceder rápidamente a un registro específico sin tener que leer todo el archivo.

Cada una de estas formas tiene sus ventajas y desventajas, y la elección depende de las necesidades específicas de la aplicación. En general cuanto mayor flexibilidad se requiere en función del espacio ocupado, mayor es la complejidad de la implementación.

3.1.5 Archivos CSV

Otra forma común de organizar los registros es utilizar el formato CSV (*Comma-Separated Values*). En este formato, cada registro se almacena en una línea del archivo, y los campos dentro del registro están separados por comas. Si un campo contiene una coma, se encierra entre comillas dobles. Si un campo contiene comillas dobles, se escapan con otra comilla doble.

Python cuenta con un módulo estándar llamado `csv` que facilita la lectura y escritura de archivos en formato CSV. Veamos cómo implementar la clase `Agenda` utilizando este formato.

Nota

La primera línea del archivo puede contener los nombres de los campos, lo cual facilita la interpretación de los datos. El módulo csv puede manejar esto automáticamente si se utiliza la clase DictReader para leer y DictWriter para escribir.

```
import csv
import os

class Agenda:
    def __init__(self, archivo, campos, separador=","):
        self._archivo = archivo
        self._campos = campos # lista de nombres de campos
        self._cant_registros = 0
        self._separador = separador

        # Escribir cabecera si el archivo no existe o está vacío
        if not os.path.exists(archivo) or os.path.getsize(archivo) == 0:
            with open(archivo, "w", newline="") as f:
                # Escribir el encabezado usando los nombres de los campos
                writer = csv.DictWriter(f, fieldnames=campos, \
                                       delimiter=separador)
                writer.writeheader() # escribe la cabecera en la primera
línea

        # Contar registros (excluyendo la cabecera)
        try:
            with open(archivo, "r", newline="") as f:
                reader = csv.DictReader(f, fieldnames=campos, \
                                       delimiter=separador)

                next(reader, None) # saltar cabecera

                for _ in reader:
                    self._cant_registros += 1
        except FileNotFoundError:
            self._cant_registros = 0

    def guardar_contacto(self, *valores):
        """
        Guarda un registro. La cantidad de valores debe coincidir con
        la cantidad de campos.
        Se arma el registro como un diccionario: {campo: valor}
        """
        if len(valores) != len(self._campos):
            raise ValueError("Cantidad de valores incorrecta")

        if not valores[0] or not valores[1]:
            raise ValueError("Nombre y apellido son obligatorios")

        registro = dict(zip(self._campos, valores)) # registro como dict

        with open(self._archivo, "a", newline="") as f:
            writer = csv.DictWriter(
                f, fieldnames=self._campos, delimiter=self._separador
            )
            writer.writerow(registro) # escribe el registro como fila dict

        self._cant_registros += 1

    def cantidad_registros(self):
        """Devuelve la cantidad de registros que hay en la agenda"""
        return self._cant_registros

    def __iter__(self):
```

```

    """Devuelve un iterador para la agenda"""
    return AgendaIterator(self)

```

A continuación definimos el iterador para la agenda:

```

class AgendaIterator:
    """Iterador para la agenda de registros en formato CSV con cabecera"""

    def __init__(self, agenda):
        self._agenda = agenda
        # Abrir el archivo en modo lectura
        self._file = open(self._agenda._archivo, "r", newline="")
        # Usar DictReader para leer registros como diccionarios
        self._reader = csv.DictReader(
            self._file,
            fieldnames=self._agenda._campos,
            delimiter=self._agenda._separador,
        )
        next(self._reader, None) # Saltar la cabecera

    def __iter__(self):
        return self

    def __next__(self):
        try:
            # Cada registro se obtiene como un diccionario {campo: valor}
            registro = next(self._reader)
            # Si faltan campos, se rellenan con cadenas vacías
            for campo in self._agenda._campos:
                if campo not in registro or registro[campo] is None:
                    registro[campo] = ""
            # Devuelve el registro parseado en campos
            return registro
        except StopIteration:
            self._file.close()
            raise StopIteration

```

Ejemplo de uso con el iterador

```

campos = ["nombre", "apellido", "telefono", "email"]
agenda = Agenda("agenda.csv", campos)
agenda.guardar_contacto("Juan", "Pérez", "123456789",
    "juan.perez@example.com")
agenda.guardar_contacto("Ana", "Gómez", "987654321", "ana.gomez@example.com")
agenda.guardar_contacto("Homer", "Simpson", "555-8765", "") # sin email
agenda.guardar_contacto("Lisa", "Simpson", "", "lisa.simpson@example.com")

for contacto in agenda:
    print(contacto)
    print("-----")

```

Si observamos el contenido del archivo `agenda.csv`, vemos que los datos están organizados en líneas, y cada campo está separado por comas. La primera línea contiene los nombres de los campos.

```

with open("agenda.csv", "r") as f:
    contenido = f.read()
    print(contenido)

print(f"Cantidad de registros: {agenda.cantidad_registros()}")

```

3.1.6 Comparación de formatos de registros

Tipo de registro	Descripción	Ventajas	Desventajas
------------------	-------------	----------	-------------

Tipo de registro	Descripción	Ventajas	Desventajas
Longitud fija, campos de longitud fija	Cada campo ocupa una cantidad fija de bytes.	Acceso rápido por posición. Implementación simple. Uso eficiente de memoria.	Desperdicio de espacio si los datos son cortos. Trunca datos largos. Poco flexible.
Longitud fija, campos de longitud variable	El registro tiene longitud fija, los campos se separan por delimitador.	Mejor aprovechamiento de espacio. Acceso rápido por posición. Implementación simple.	No permite registros largos. El delimitador no puede estar en los datos. Menos flexible.
Longitud variable, campos de longitud variable	Cada registro y campo precedido por su longitud en bytes.	Máxima flexibilidad. No desperdicia espacio. Permite cualquier longitud de datos.	Acceso secuencial. Implementación más compleja. Mayor sobrecarga por longitud extra.
CSV (Comma-Separated Values)	Cada registro es una línea, campos separados por comas.	Muy utilizado. Fácil de leer y editar. Compatible con muchas herramientas.	No apto para acceso aleatorio. Problemas con comas en los datos. Menos eficiente.

3.2 JavaScript Object Notation (JSON)

JSON (*JavaScript Object Notation*) es un formato ligero de intercambio de datos que es fácil de leer y escribir para los humanos, y fácil de parsear y generar para las máquinas. Es un formato de texto que utiliza una sintaxis basada en objetos y arrays de JavaScript.

3.2.1 Estructura de JSON

Un archivo JSON está compuesto por una serie de pares clave-valor, donde cada clave es una cadena de texto y cada valor puede ser un número, una cadena de texto, un booleano, un array o un objeto. Los objetos se representan mediante llaves {} y los arrays mediante corchetes [].

```
{
  "nombre": "Juan",
  "edad": 30,
  "es_estudiante": false,
  "cursos": ["Matemáticas", "Física", "Química"],
  "direccion": {
    "calle": "Calle Falsa 123",
    "ciudad": "Springfield",
    "pais": "USA"
  }
}
```

Este formato es muy utilizado para el intercambio de datos entre aplicaciones web y servidores, ya que es fácil de parsear y generar en la mayoría de los lenguajes de programación.

En la sección de Persistencia de Datos vimos cómo trabajar con archivos JSON en Python utilizando la librería estándar `json`. Aquí veremos cómo utilizar JSON para organizar registros en un archivo.

3.2.2 Organizando registros con JSON

Una forma común de organizar registros en un archivo JSON es utilizar una lista de objetos, donde cada objeto representa un registro. Por ejemplo, si queremos almacenar una agenda de contactos, podemos utilizar la siguiente estructura:

```
[
  {
    "nombre": "Juan",
    "telefono": "123456789",
    "email": "juan@example.com"
  },
  {
    "nombre": "Ana",
    "telefono": "987654321",
    "email": "ana@example.com"
  }
]
```

Cada objeto en la lista representa un contacto, con campos para el nombre, teléfono y correo electrónico. Podemos agregar, eliminar o modificar contactos simplemente manipulando la lista de objetos.

En Python, podemos trabajar con este archivo JSON utilizando la librería `json` de la siguiente manera

3.2.3 Agenda en formato JSON

A continuación se muestra cómo definir una clase `Agenda` que almacena los contactos en un archivo `.json`, junto con su iterador y ejemplos de uso.

```
import json
import os
```

```

class Agenda:
    def __init__(self, archivo):
        self._archivo = archivo

        # Si el archivo no existe, lo crea con una lista vacía
        if not os.path.exists(archivo):
            with open(archivo, "w") as f:
                json.dump([], f)

        # Carga los contactos existentes
        with open(archivo, "r") as f:
            self._contactos = json.load(f)

    def guardar_contacto(self, nombre, telefono="", email=""):
        if not nombre:
            raise ValueError("El nombre es obligatorio")

        contacto = {"nombre": nombre, "telefono": telefono, "email": email}
        self._contactos.append(contacto)

        with open(self._archivo, "w") as f:
            json.dump(self._contactos, f, ensure_ascii=False, indent=2)

    def cantidad_registros(self):
        return len(self._contactos)

    def __iter__(self):
        return AgendaIterator(self)

```

Definimos el iterador para la agenda:

```

class AgendaIterator:
    """Iterador para la agenda de contactos en formato JSON"""

    def __init__(self, agenda):
        self._agenda = agenda
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index >= len(self._agenda._contactos):
            raise StopIteration()

        contacto = self._agenda._contactos[self._index]
        self._index += 1
        return contacto

```

Ejemplo de uso:

```

agenda = Agenda("agenda.json")
agenda.guardar_contacto("Juan", "123456789", "juan@example.com")
agenda.guardar_contacto("Ana", "987654321", "ana@example.com")
agenda.guardar_contacto("Homero", "555-8765", "")
agenda.guardar_contacto("Lisa", "", "lisa.simpson@example.com")

print(f"Cantidad de registros: {agenda.cantidad_registros()}")
for contacto in agenda:
    print(contacto)
    print("-----")

```

Si vemos el contenido del archivo `agenda.json`, se observa que los datos están guardados en formato de texto legible, siguiendo el estándar JSON

```
with open("agenda.json", "r") as f:
    contenido = f.read()
    print(contenido)
```

La principal ventaja de utilizar JSON para organizar registros es que es un formato ampliamente soportado y fácil de leer y escribir. Además, permite almacenar datos estructurados de manera flexible, ya que los objetos pueden tener diferentes campos y tipos de datos.

A continuación se define una agenda general donde solo los campos nombres y apellidos son obligatorios, y donde cada registro puede tener incluso campos diferentes.

```
class AgendaGeneral:
    def __init__(self, archivo):
        self._archivo = archivo

        if not os.path.exists(archivo):
            with open(archivo, "w") as f:
                json.dump([], f)

        with open(archivo, "r") as f:
            self._contactos = json.load(f)

    def guardar_contacto(self, **kwargs):
        if "nombre" not in kwargs or "apellido" not in kwargs:
            raise ValueError(
                "Los campos 'nombre' y 'apellido' son obligatorios"
            )

        self._contactos.append(kwargs)

        with open(self._archivo, "w") as f:
            json.dump(self._contactos, f, ensure_ascii=False, indent=2)

    def cantidad_registros(self):
        return len(self._contactos)

    def __iter__(self):
        return AgendaIterator(self)
```

Ejemplo de uso:

```
agenda = AgendaGeneral("agenda_general.json")
agenda.guardar_contacto(
    nombre="Juan",
    apellido="Pérez",
    telefono="123456789",
    email="juan.perez@example.com",
)
agenda.guardar_contacto(
    nombre="Ana",
    apellido="García",
    telefono="987654321",
    cumpleaños="1990-01-01"
)
agenda.guardar_contacto(
    nombre="Homer",
    apellido="Simpson",
    direccion={"calle": "742 Evergreen Terrace", "ciudad": "Springfield"},
    telefono="555-8765",
)
agenda.guardar_contacto(
    nombre="Lisa",
    apellido="Simpson",
    email="lisa.simpson@example.com",
    hobbies=["saxofón", "política"],
)
```

```

)
agenda.guardar_contacto(
    nombre="Bart",
    apellido="Simpson",
    telefono="555-1234",
    email="bart.simpson@example.com",
)
for contacto in agenda:
    # Imprime nombre y apellido en la primera línea
    nombre = contacto.get("nombre", "")
    apellido = contacto.get("apellido", "")
    print(f"{nombre} {apellido}")

    # Función recursiva para imprimir campos
    def imprimir_campos(d, indent=2):
        for clave, valor in d.items():
            if clave in ("nombre", "apellido"):
                continue

            if isinstance(valor, dict):
                print(" " * indent + f"{clave}:")
                imprimir_campos(valor, indent + 2)
            elif isinstance(valor, list):
                print(" " * indent + f"{clave}: [")
                for item in valor:
                    if isinstance(item, dict):
                        imprimir_campos(item, indent + 4)
                    else:
                        print(" " * (indent + 2) + f"- {item}")
                print(" " * indent + "]")
            else:
                print(" " * indent + f"{clave}: {valor}")

    imprimir_campos(contacto)
    print("-----")

```

Archivo agenda_general.json:

```

with open("agenda_general.json", "r") as f:
    contenido = f.read()
    print(contenido)

print(f"Cantidad de bytes en el archivo: {len(contenido)}")
print(f"Cantidad de registros: {agenda.cantidad_registros()}")

```

3.3 EXtensible Markup Language (XML)

3.3.1 XML

XML (*eXtensible Markup Language*) es un lenguaje de marcado que define un conjunto de reglas para la codificación de documentos en un formato que es tanto legible por humanos como por máquinas. Fue diseñado para almacenar y transportar datos, y es ampliamente utilizado en aplicaciones web, servicios web y sistemas de intercambio de datos.

XML permite a los usuarios definir sus propias etiquetas y estructuras de datos, lo que lo hace muy flexible y adaptable a diferentes necesidades. A diferencia de HTML, que tiene un conjunto fijo de etiquetas predefinidas, XML permite crear etiquetas personalizadas que describen el contenido de manera más precisa.

XML es un estándar abierto mantenido por el World Wide Web Consortium (W3C) y es ampliamente utilizado en diversas aplicaciones, incluyendo:

- Intercambio de datos entre sistemas diferentes.
- Configuración de aplicaciones.
- Almacenamiento de datos estructurados.
- Representación de documentos complejos (como docx, xlsx, etc.)
- Difusión de Noticias (RSS, Atom).

3.3.1.1 Estructura de un documento XML

Un documento XML está compuesto por varios elementos clave:

Prólogo Es la parte inicial del documento que puede incluir una declaración XML y otras instrucciones, como la definición de la codificación de caracteres. La declaración XML es opcional pero recomendada, y suele verse así: `<?xml version="1.0" encoding="UTF-8"?>`.

Elementos Son las unidades básicas de un documento XML y están delimitados por etiquetas de apertura y cierre. Por ejemplo, `<nombre>Juan</nombre>` es un elemento que contiene el texto "Juan".

Atributos Son pares clave-valor que proporcionan información adicional sobre un elemento. Se incluyen dentro de la etiqueta de apertura. Por ejemplo, `<persona id="123">` tiene un atributo `id` con el valor "123".

Contenido Es el texto o los datos que se encuentran entre las etiquetas de apertura y cierre de un elemento.

Comentarios Se pueden incluir comentarios en un documento XML utilizando la sintaxis `<!-- Comentario -->`.

El siguiente fragmento muestra un poema estructurado en XML

```
<?xml version="1.0" encoding="UTF-8"?>

<poema fecha="Abril de 1915" lugar="Granada">
  <titulo>Alba</titulo>
  <verso>Mi corazón oprimido</verso>
  <verso>siente junto a la alborada</verso>
  <verso>el dolor de sus amores</verso>
  <verso>y el sueño de las distancias. </verso>
</poema>
```

donde se pueden distinguir claramente los diferentes elementos:

- poema
- titulo
- verso

y los atributos de poema

- @fecha
- @lugar

Como se puede observar XML permite estructurar la información en forma jerárquica, como si fuera un árbol, donde los elementos pueden contener otros elementos anidados. Un archivo XML bien formado debe cumplir con ciertas reglas, como tener un único elemento raíz que contenga todos los demás elementos, y todas las etiquetas deben estar correctamente cerradas y anidadas.

Los atributos en XML permiten agregar metadatos o información adicional a los elementos, lo que puede ser útil para describir propiedades o características específicas de los datos. Sin embargo, es importante usarlos con moderación y de manera coherente para evitar complicaciones en la interpretación del documento.

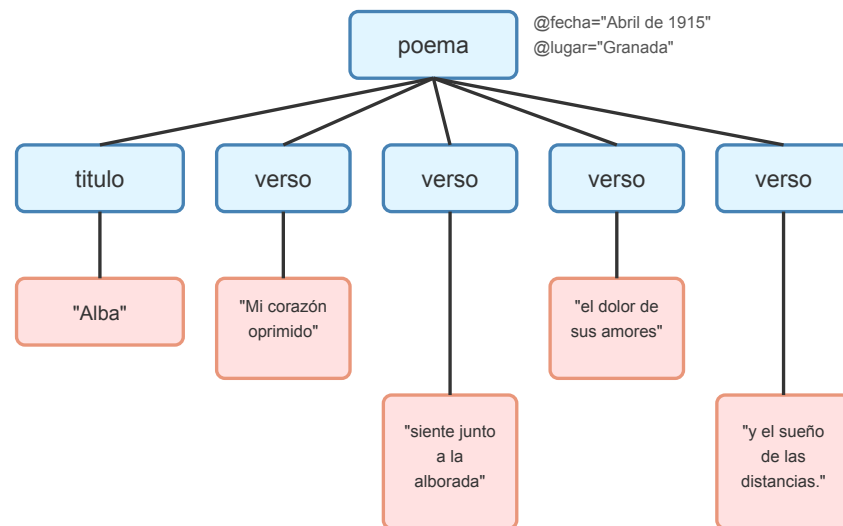


Figure 3.1: Árbol que representa la estructura del poema en XML.

Existen varios sitios en internet que permiten visualizar el árbol asociado. Por ejemplo <https://codebeautify.org/xmlviewer>.

En el ejemplo anterior, el elemento raíz es poema, que contiene como elementos hijos: titulo y varios elementos verso. El elemento poema también tiene dos atributos: fecha y lugar, que proporcionan información adicional sobre el poema.

Que un documento XML se pueda representar como un árbol simplifica la consulta y manipulación de los datos, ya que se pueden utilizar técnicas de recorrido de árboles para acceder a elementos específicos o extraer información relevante.

3.3.2 XPath: XML Path Language

XPath (*XML Path Language*) es un lenguaje de consulta utilizado para navegar y seleccionar nodos en documentos XML. Proporciona una sintaxis para definir rutas que permiten localizar elementos, atributos y otros nodos dentro de la estructura jerárquica de un documento XML.

Permite describir caminos a través del árbol XML utilizando una notación similar a la de los sistemas de archivos. Por ejemplo, la expresión `/poema/titulo` selecciona el elemento **titulo** que es hijo directo del elemento raíz **poema**. Evaluar una expresión XPath es buscar **elementos** o **atributos** en un documento XML que coincidan con los criterios especificados en la expresión. El resultado son todos los nodos que cumplen con esos criterios.

Por ejemplo, la expresión `//verso` selecciona todos los elementos **verso** en el documento, independientemente de su posición en la jerarquía. La expresión `//verso[text()='Mi corazón oprimido']` selecciona el elemento **verso** que contiene el texto exacto "Mi corazón oprimido".

En los ejemplos a continuación usaremos el siguiente documento XML que representa una biblioteca con varios libros y autores.

```

<?xml version="1.0" encoding="UTF-8"?>
<biblioteca>
  <libro>
    <titulo>La vida está en otra parte</titulo>
    <autor>Milan Kundera</autor>
    <fechaPublicacion año="1973" />
    <precio>305.50</precio>
  </libro>
  <revista>
    <titulo>Computer Gaming World</titulo>
    <editorial>Golden Empire Publications</editorial>
    <fechaPublicacion año="1981" />
    <precio>669.99</precio>
  </revista>
  <libro>
    <titulo>Pantaleón y las visitadoras</titulo>
    <autor fechaNacimiento="28/03/1936">Mario Vargas Llosa</autor>
    <fechaPublicacion año="1973" />
    <precio>214.48</precio>
  </libro>
  <libro>
    <titulo>Conversación en la catedral</titulo>
    <autor fechaNacimiento="28/03/1936">Mario Vargas Llosa</autor>
    <fechaPublicacion año="1969" />
    <precio>541.78</precio>
  </libro>
  <revista>
    <titulo>PC Users</titulo>
    <editorial>RedUsers</editorial>
    <fechaPublicacion año="2000" />
    <precio>220.50</precio>
  </revista>
</biblioteca>

```

3.3.2.1 Expresiones XPath para seleccionar nodos

Expresión	Descripción
/	Si está al principio de la expresión, indica el nodo raíz, si no, indica "hijo"
//	Camino. Permite seleccionar nodos en un camino descendiente a partir de la posición actual
.	Nodo actual
..	Padre del nodo actual
@nombre_atributo	Atributo

Cada consulta XPath devuelve un conjunto de nodos que cumplen con los criterios especificados en la expresión. Se recomienda realizar pruebas en XPath Tester.

/biblioteca Selecciona el nodo raíz biblioteca.

/biblioteca/libro Selecciona todos los nodos libro que son hijos directos de biblioteca.

//autor Selecciona todos los nodos autor en el documento, independientemente de su posición en la jerarquía.

/biblioteca//titulo Selecciona todos los nodos titulo que son descendientes de biblioteca, sin importar cuántos niveles haya entre ellos. En este caso, selecciona los títulos de libros y revistas.

//libro/precio Selecciona todos los nodos precio que son hijos directos de cualquier nodo libro.

//editorial/.. Selecciona el nodo padre de todos los nodos editorial, que en este caso son nodos revista.

//autor[@fechaNacimiento] Selecciona todos los nodos autor que tengan un atributo fechaNacimiento.

3.3.2.2 Predicados

Los predicados pueden ser usados para filtrar un conjunto de nodos en base a una condición dada.

Los predicados se escriben entre corchetes ([,]). **/biblioteca/libro[position() = 1]** : Selecciona el primer nodo libro hijo de biblioteca.

/biblioteca/libro[1] Equivalente a la expresión anterior, selecciona el primer nodo libro hijo de biblioteca.

/biblioteca/libro[last()] Selecciona el último nodo libro hijo de biblioteca.

/biblioteca/libro[last() - 1] Selecciona el penúltimo nodo libro hijo de biblioteca.

/biblioteca/libro[position() < 3] Selecciona los dos primeros nodos libro hijos de biblioteca.

//autor[not(@fechaNacimiento)] Selecciona todos los nodos autor que **no** tengan un atributo fechaNacimiento.

//autor[@fechaNacimiento="28/03/1936"] Selecciona todos los nodos autor cuyo atributo fechaNacimiento tenga el valor "28/03/1936".

3.3.2.3 Selectores y comodines

Expresión	Resultado
*	Todos los elementos en el nivel actual
@*	Todos los atributos del nodo actual
text()	El contenido de texto de un nodo

/biblioteca/* Selecciona todos los elementos hijos directos de biblioteca.

/biblioteca/** Selecciona todos los elementos descendientes de biblioteca.

//autor[@*] Selecciona todos los elementos autor que tengan algún atributo.

node() Selecciona todos los nodos del documento.

//titulo/text() Selecciona el texto (no el nodo completo) de los títulos.

3.3.2.4 Selección de varios caminos

El operador | es el operador de unión permite seleccionar distintos caminos en el documento.

//libro/titulo | //libro/precio Selecciona todos los nodos titulo y todos los nodos precio hijos de libro |

3.3.2.5 Comparaciones

Operador	Descripción	Ejemplo
=	Igualdad	precio = 541.78
!=	Distinto	precio != 541.78
<	Menor estricto	precio < 500
<=	Menor o igual	precio <= 541.78
>	Mayor estricto	precio > 500
>=	Mayor o igual	precio >= 541.78

/biblioteca//libro[precio < 350] Selecciona todos los nodos libro que tengan como hijo directo un elemento precio con valor menor a 350.

/biblioteca//libro[precio < 350]/titulo Selecciona todos los nodos titulo hijos de nodos libro que tengan como hijo directo un elemento precio con valor menor a 350.

3.3.2.6 Operadores

Operador	Descripción	Ejemplo
+	Suma	6 + 4
-	Sustracción	6 - 4
*	Multiplicación	6 * 4
div	División	8 div 4
or	Disyunción	precio = 541.78 or precio = 214.48
and	Conjunción	precio > 300 and precio <= 541.78
mod	Módulo (resto de la división entera)	5 mod 2

/biblioteca//libro[precio > 300 and precio <= 541.78] Selecciona todos los nodos libro que tengan como hijo directo un elemento precio con valor mayor a 300 y menor o igual a 541.78.

/biblioteca//libro[precio = 541.78 or precio = 214.48] Selecciona todos los nodos libro que tengan como hijo directo un elemento precio con valor igual a 541.78 o 214.48.

/biblioteca//libro[precio + 100 > 600] Selecciona los nodos libro donde el valor del elemento precio sumado a 100 es mayor que 600.

3.3.3 XML y Python

Python ofrece varias bibliotecas para trabajar con XML, siendo las más comunes `xml.etree.ElementTree`, `lxml` y `xml.dom.minidom`. En los ejemplos usaremos `lxml` que es muy potente y flexible.

Para instalar `lxml`, se puede usar `pip`:

```
pip install lxml
from lxml import etree

# Cargar el documento XML
tree = etree.parse("../_static/code/xml/biblioteca.xml")
root = tree.getroot()

# Realizar una consulta XPath
# Seleccionar todos los títulos de libros
titulos = root.xpath("/biblioteca/libro/titulo/text()")
print("Títulos de libros:")
for titulo in titulos:
    print(titulo)

# Seleccionar todos los autores con fecha de nacimiento
autores_con_fecha = root.xpath("//autor[@fechaNacimiento]/text()")
print("\nAutores con fecha de nacimiento:")
for autor in autores_con_fecha:
    print(autor)

# Seleccionar libros con precio menor a 300
libros_baratos = root.xpath("/biblioteca/libro[precio < 300]/titulo/text()")
print("\nLibros con precio menor a 300:")
for libro in libros_baratos:
    print(libro)
```

Otro ejemplo: Calcular el precio total de los libros

```
from lxml import etree

# Cargar el documento XML
tree = etree.parse("../_static/code/xml/biblioteca.xml")
root = tree.getroot()
```

```
# Calcular el precio total de todos los libros
precios = root.xpath("/biblioteca/libro/precio/text()")
total_precio = sum(float(precio) for precio in precios)

print(f"\nPrecio total de todos los libros: {total_precio:.2f}")
```

3.3.3.1 Noticias con RSS y Atom

RSS (Really Simple Syndication o Rich Site Summary) es un formato basado en XML que se utiliza para distribuir y compartir contenido web, como noticias, blogs, podcasts y otros tipos de información actualizada regularmente. Los archivos RSS permiten a los usuarios suscribirse a fuentes de contenido y recibir actualizaciones automáticas cuando se publica nuevo contenido. A continuación se muestra un ejemplo de lectura del feed RSS con las últimas noticias del diario Clarin:

```
import feedparser

# URL del feed RSS de Clarin
rss_url = "https://www.clarin.com/rss/lo-ultimo/"

# Parsear el feed RSS
feed = feedparser.parse(rss_url)

# Mostrar los títulos y enlaces de las últimas noticias
print("Últimas noticias de Clarin:\n")
for entry in feed.entries[:5]: # Mostrar solo las primeras 5 noticias
    print(f"Título: {entry.title}")
    print(f"Fecha de publicación: {entry.published}")
    print(f"Enlace: {entry.link}\n")
```

3.4 Expresiones Regulares (Regex)

Las **expresiones regulares** (también conocidas como *regex* o *regexp*) son secuencias de caracteres que forman un patrón de búsqueda. Son una herramienta fundamental para el procesamiento de texto y la recuperación de información, permitiendo buscar, validar, extraer y manipular cadenas de texto de forma eficiente y flexible.

En el contexto de las estructuras de datos y la recuperación de información, las expresiones regulares son esenciales para:

- **Búsqueda y extracción de información** en grandes volúmenes de texto
- **Validación de datos** (emails, teléfonos, URLs, etc.)
- **Procesamiento de logs** y archivos de texto
- **Web scraping** y extracción de datos de páginas web
- **Limpieza y transformación de datos** antes de almacenarlos en estructuras de datos

3.4.1 ¿Qué son las expresiones regulares?

Una expresión regular es un patrón que describe un conjunto de cadenas de texto. Por ejemplo, el patrón `r"\d{3}-\d{4}"` describe cualquier cadena que tenga tres dígitos, seguidos de un guion, seguidos de cuatro dígitos (como "123-4567").

Las expresiones regulares utilizan una sintaxis especial con **metacaracteres** que tienen significados especiales. Estos metacaracteres permiten definir patrones complejos de forma concisa.

3.4.1.1 Ejemplo simple

```
import re

texto = "Mi teléfono es 555-1234"
patron = r"\d{3}-\d{4}"

resultado = re.search(patron, texto)
if resultado:
    print(f"Encontrado: {resultado.group()}")
else:
    print("No encontrado")
```

3.4.2 Sintaxis básica y metacaracteres

Las expresiones regulares utilizan caracteres especiales (metacaracteres) que tienen significados específicos. Veamos los más importantes:

3.4.2.1 Caracteres literales

Los caracteres normales (letras, números) se buscan literalmente:

```
import re

texto = "Python es un lenguaje de programación"
patron = r"Python"

if re.search(patron, texto):
    print("Se encontró 'Python'")
```

3.4.2.2 Metacaracteres básicos

- . (**punto**) Coincide con cualquier carácter excepto salto de línea
- ^ (**circunflejo**) Coincide con el inicio de la cadena
- \$ (**signo de dólar**) Coincide con el final de la cadena
- * (**asterisco**) Coincide con 0 o más repeticiones del patrón anterior
- + (**más**) Coincide con 1 o más repeticiones del patrón anterior
- ? (**interrogación**) Coincide con 0 o 1 repetición del patrón anterior
- [] (**corchetes**) Define un conjunto de caracteres
- | (**barra vertical**) Operador OR (alternativa)

- () (paréntesis)** Agrupa expresiones y captura coincidencias
- \ (barra invertida)** Escapa metacaracteres o define secuencias especiales

3.4.2.3 Ejemplos de metacaracteres

```
import re

# Punto: cualquier carácter
print(re.findall(r"c.sa", "casa cosa cesa")) # ['casa', 'cosa', 'cesa']

# Inicio y fin de cadena
print(re.search(r"^Hola", "Hola mundo")) # Encuentra 'Hola'
print(re.search(r"mundo$", "Hola mundo")) # Encuentra 'mundo'

# Asterisco: 0 o más
print(re.findall(r"lo*", "l lo loo looo")) # ['l', 'lo', 'loo', 'looo']

# Más: 1 o más
print(re.findall(r"lo+", "l lo loo looo")) # ['lo', 'loo', 'looo']

# Interrogación: 0 o 1
print(re.findall(r"lo?", "l lo loo")) # ['l', 'lo', 'lo']
```

3.4.2.4 Cuantificadores

Los cuantificadores especifican cuántas veces debe aparecer el patrón anterior:

- {n}** Exactamente n repeticiones
- {n,}** Al menos n repeticiones
- {n,m}** Entre n y m repeticiones

```
import re

texto = "El código es 12, 123, 1234 y 12345"

# Exactamente 3 dígitos
print(re.findall(r"\d{3}", texto)) # ['123', '123']

# Al menos 3 dígitos
print(re.findall(r"\d{3,}", texto)) # ['123', '1234', '12345']

# Entre 2 y 4 dígitos
print(re.findall(r"\d{2,4}", texto)) # ['12', '123', '1234', '1234']
```

3.4.2.5 Clases de caracteres

Las clases de caracteres permiten definir conjuntos de caracteres válidos:

- [abc]** Cualquiera de los caracteres a, b o c
- [a-z]** Cualquier letra minúscula
- [A-Z]** Cualquier letra mayúscula
- [0-9]** Cualquier dígito
- [^abc]** Cualquier carácter excepto a, b o c

```
import re

texto = "El código postal es A1234BCZ"

# Letras mayúsculas
print(re.findall(r"[A-Z]", texto))

# Dígitos
print(re.findall(r"[0-9]+", texto))

# Letras y números
print(re.findall(r"[A-Z0-9]+", texto))
```

```
# Todo excepto espacios
print(re.findall(r"[^ ]+", texto))
```

3.4.2.6 Secuencias especiales

Python proporciona atajos para clases de caracteres comunes:

```
\d Cualquier dígito (equivalente a [0-9])
\D Cualquier no-dígito
\w Cualquier carácter de palabra: letra, dígito o guion bajo (equivalente a [a-zA-Z0-9_])
\W Cualquier no-carácter de palabra
\s Cualquier espacio en blanco (espacio, tab, salto de línea)
\S Cualquier no-espacio en blanco
\b Límite de palabra
\B No-límite de palabra
```

```
import re

texto = "Usuario: juan_123, Email: juan@email.com"

# Dígitos
print(re.findall(r"\d+", texto)) # ['123']

# Caracteres de palabra
print(
    re.findall(r"\w+", texto)
) # ['Usuario', 'juan_123', 'Email', 'juan', 'email', 'com']

# Límites de palabra
print(re.findall(r"\bjuan\b", texto)) # ['juan']
print(re.findall(r"\bjuan", "juanito juan")) # ['juan', 'juan']
```

3.4.3 El módulo re en Python

Python proporciona el módulo re para trabajar con expresiones regulares. Este módulo incluye varias funciones útiles:

3.4.3.1 Funciones principales

```
re.search(patron, texto) Busca la primera ocurrencia del patrón en el texto
re.match(patron, texto) Busca el patrón solo al inicio del texto
re.findall(patron, texto) Devuelve todas las ocurrencias del patrón como una lista
re.finditer(patron, texto) Devuelve un iterador con todas las ocurrencias
re.sub(patron, reemplazo, texto) Reemplaza las ocurrencias del patrón
re.split(patron, texto) Divide el texto usando el patrón como separador
re.compile(patron) Compila el patrón para reutilizarlo eficientemente
```

3.4.3.2 Ejemplos de uso

```
import re

texto = "Los emails son: juan@email.com, ana@empresa.com.ar y pedro@sitio.org"

# search: encuentra la primera coincidencia
resultado = re.search(r"\w+@\w+\. \w+", texto)
if resultado:
    print(f"Primer email encontrado: {resultado.group()}")

# findall: encuentra todas las coincidencias
emails = re.findall(r"\w+@[ \w. ]+", texto)
print(f"Todos los emails: {emails}")

# finditer: iterador sobre las coincidencias
for match in re.finditer(r"\w+@[ \w. ]+", texto):
    print(f"Email en posición {match.start()}-{match.end()}: {match.group()}")
```

3.4.3.3 Compilación de patrones

Cuando se usa un patrón múltiples veces, es más eficiente compilarlo:

```
import re

# Compilar el patrón una sola vez
patron_email = re.compile(r"\w+@[\.w.]+")

textos = [
    "Contacto: juan@email.com",
    "Soporte: ayuda@empresa.com",
    "Ventas: ventas@sitio.org",
]

for texto in textos:
    match = patron_email.search(texto)
    if match:
        print(f"{texto} -> {match.group()}")
```

3.4.3.4 Grupos de captura

Los paréntesis () crean grupos de captura que permiten extraer partes específicas del patrón:

```
import re

texto = "Fecha: 15/03/2024"
patron = r"(\d{2})/(\d{2})/(\d{4})"

match = re.search(patron, texto)
if match:
    print(f"Fecha completa: {match.group(0)}") # Toda la coincidencia
    print(f"Día: {match.group(1)}") # Primer grupo
    print(f"Mes: {match.group(2)}") # Segundo grupo
    print(f"Año: {match.group(3)}") # Tercer grupo
    print(f"Todos los grupos: {match.groups()}") # Tupla con todos los grupos
```

3.4.3.5 Grupos nombrados

Se pueden asignar nombres a los grupos para mayor claridad:

```
import re

texto = "Producto: ABC-123 Precio: $450.50"
patron = r"(?P<codigo>[A-Z]+\d+).*?(?P<precio>\d+\.\d+)"

match = re.search(patron, texto)
if match:
    print(f"Código: {match.group('codigo')}")
    print(f"Precio: {match.group('precio')}")
    print(f"Diccionario: {match.groupdict()}")
```

3.4.3.6 Miradas alrededor

Las miradas alrededor permiten hacer coincidir un patrón solo si está precedido o seguido por otro patrón, sin incluirlo en la coincidencia:

(?=...) Mirada hacia adelante positiva (*lookahead*)
(?!...) Mirada hacia adelante negativa
(?<=...) Mirada hacia atrás positiva (*lookbehind*)
(?<?!...) Mirada hacia atrás negativa

```
import re

texto = "foo1 bar2 foo3 baz4"
# Mirada hacia adelante: foo seguido de un dígito
print(re.findall(r"foo(=?\d)", texto)) # ['foo', 'foo']
```

```
# Mirada hacia atrás: dígito precedido de foo
print(re.findall(r"(?<=foo)\d", texto)) # ['1', '3']
```

3.4.4 Casos de uso en recuperación de información

Las expresiones regulares son fundamentales en la recuperación y procesamiento de información. Veamos casos prácticos:

3.4.4.1 Validación de datos

```
import re

def validar_email(email):
    """Valida un email con expresión regular"""
    patron = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    return re.match(patron, email) is not None

def validar_telefono_argentina(telefono):
    """Valida teléfono argentino: +54 11 1234-5678"""
    patron = r"^\+54\s?\d{2}\s?\d{4}-?\d{4}$"
    return re.match(patron, telefono) is not None

def validar_url(url):
    """Valida una URL"""
    patron = r"^https?://(www\.)?[\w.-]+\.[a-z]{2,}(/[\w.-]*)*/?$"
    return re.match(patron, url) is not None

# Pruebas
print(f"Email válido: {validar_email('juan@email.com')}")
print(f"Email inválido: {validar_email('juan@email')}")
print(f"Teléfono válido: {validar_telefono_argentina('+54 11 1234-5678')}")
print(f"URL válida: {validar_url('https://www.ejemplo.com/path')}")
```

3.4.4.2 Extracción de información

```
import re

# Texto de ejemplo: log de servidor web
log = """
192.168.1.1- -[15/Mar/2024:10:30:45 +0000] "GET /index.html HTTP/1.1" 200 1234
10.0.0.5- -[15/Mar/2024:10:31:12 +0000] "POST /api/data HTTP/1.1" 201 567
192.168.1.1- -[15/Mar/2024:10:32:33 +0000] "GET /styles.css HTTP/1.1" 200 8910
"""

# Patrón para extraer información del log
patron = (
    r'(\d+\.\d+\.\d+\.\d+).*?\[([^\]]+)\]' # IP y fecha
    r'\s+(\w+)\s+([\s]+).*?"\s+(\d+)\s+(\d+)' # Método, ruta, código y tamaño
)

for linea in log.strip().split("\n"):
    match = re.search(patron, linea)
    if match:
        ip, fecha, metodo, ruta, codigo, tamaño = match.groups()
        print(f"IP: {ip}")
        print(f"Fecha: {fecha}")
        print(f"Método: {metodo}")
        print(f"Ruta: {ruta}")
        print(f"Código: {codigo}")
        print(f"Tamaño: {tamaño} bytes")
        print("---")
```

3.4.4.3 Limpieza y normalización de texto

```
import re

def limpiar_texto(texto):
    """Limpia un texto para procesamiento"""
    # Eliminar URLs
    texto = re.sub(r"https?://\S+", "", texto)

    # Eliminar emails
    texto = re.sub(r"\S+@\S+", "", texto)

    # Eliminar múltiples espacios
    texto = re.sub(r"\s+", " ", texto)

    # Eliminar caracteres especiales (mantener letras, números y espacios)
    texto = re.sub(r"^\w\s", "", texto)

    return texto.strip()

texto_sucio = """
Visita https://ejemplo.com para más info!
Contacto: info@ejemplo.com
    Múltiples espacios aquí...
¿Caracteres especiales? @$%
"""

print("Texto original:")
print(texto_sucio)
print("\nTexto limpio:")
print(limpiar_texto(texto_sucio))
```

3.4.4.4 Tokenización de texto

```
import re

def tokenizar(texto):
    """Divide un texto en palabras (tokens)"""
    # Encontrar todas las secuencias de caracteres de palabra
    tokens = re.findall(r"\b\w+\b", texto.lower())
    return tokens

# Contar frecuencia de palabras
from collections import Counter

texto = "Python es un lenguaje de programación. ¡Es genial!"
tokens = tokenizar(texto)
print(f"Tokens: {tokens}")
print(f"Total de tokens: {len(tokens)}")

frecuencias = Counter(tokens)
print(f"Palabras más comunes: {frecuencias.most_common(3)}")
```

3.4.5 Aplicaciones en procesamiento de texto

3.4.5.1 Búsqueda de patrones en archivos

```
import re

# Crear un archivo de ejemplo
contenido_archivo = """
Estructuras de datos
Python es un lenguaje de programación versátil
Expresiones regulares son herramientas poderosas
Python se usa en ciencia de datos
Las estructuras de datos son fundamentales
"""
```

```

"""

with open("ejemplo.txt", "w") as f:
    f.write(contenido_archivo)

def buscar_en_archivo(archivo, patron):
    """Busca un patrón en un archivo y devuelve las líneas que coinciden"""
    resultados = []
    with open(archivo, "r") as f:
        for num_linea, linea in enumerate(f, 1):
            if re.search(patron, linea, re.IGNORECASE):
                resultados.append((num_linea, linea.strip()))
    return resultados

# Buscar líneas que contengan "Python" o "datos"
patron = r"Python|datos"
resultados = buscar_en_archivo("ejemplo.txt", patron)

print("Líneas encontradas:")
for num, linea in resultados:
    print(f"Línea {num}: {linea}")

```

3.4.5.2 Extracción de datos estructurados

```

import re

# HTML de ejemplo
html = """
<div class="producto">
    <h2>Laptop HP</h2>
    <p class="precio">$899.99</p>
    <p>Procesador Intel i7</p>
</div>
<div class="producto">
    <h2>Mouse Logitech</h2>
    <p class="precio">$25.50</p>
    <p>Inalámbrico</p>
</div>
"""

def extraer_productos(html):
    """Extrae información de productos del HTML"""
    # Patrón para encontrar bloques de productos
    patron_producto = (
        r'<div class="producto">.*?<h2>(.*?)</h2>.*?'
        r'<p class="precio">\$([\d.]+)</p>'
    )

    productos = re.findall(patron_producto, html, re.DOTALL)

    return [
        {"nombre": nombre, "precio": float(precio)}
        for nombre, precio in productos
    ]

productos = extraer_productos(html)
for i, prod in enumerate(productos, 1):
    print(f"Producto {i}: {prod['nombre']} - ${prod['precio']}")

```

3.4.5.3 Reemplazo y transformación de texto

```

import re

```

```

def anonimizar_datos(texto):
    """Anonimiza datos sensibles en un texto"""
    # Anonimizar emails
    texto = re.sub(
        r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b",
        "[EMAIL]",
        texto
    )

    # Anonimizar teléfonos (formato: xxx-xxxx o (xxx) xxx-xxxx)
    texto = re.sub(r"\((?\d{3})\)?[-.\s]?(\d{3})[-.\s]?(\d{4})", "[TELÉFONO]",
    texto)

    # Anonimizar números de tarjeta (grupos de 4 dígitos)
    texto = re.sub(r"\b\d{4}[-\s]?(\d{4})[-\s]?(\d{4})[-\s]?(\d{4})\b", "[TARJETA]",
    texto)

    return texto

texto_sensible = """
Contacto: juan.perez@email.com
Teléfono: 555-1234 o (555) 555-5678
Tarjeta: 1234 5678 9012 3456
"""

print("Texto original:")
print(texto_sensible)
print("\nTexto anonimizado:")
print(anonimizar_datos(texto_sensible))

```

3.4.5.4 Análisis de texto y extracción de métricas

```

import re
from collections import Counter

def analizar_texto(texto):
    """Analiza un texto y extrae métricas"""
    # Contar oraciones (terminan en . ! ?)
    oraciones = re.split(r"[.!?]+", texto)
    num_oraciones = len([s for s in oraciones if s.strip()])

    # Contar palabras
    palabras = re.findall(r"\b\w+\b", texto.lower())
    num_palabras = len(palabras)

    # Palabras más comunes
    frecuencias = Counter(palabras)
    palabras_comunes = frecuencias.most_common(5)

    # Contar números
    numeros = re.findall(r"\b\d+\b", texto)

    # Detectar emails
    emails = re.findall(r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b", texto)

    # Detectar URLs
    urls = re.findall(r"https?://[^\s]+", texto)

    return {
        "oraciones": num_oraciones,
        "palabras": num_palabras,
        "palabras_comunes": palabras_comunes,

```

```

        "numeros": numeros,
        "emails": emails,
        "urls": urls,
    }

```

```

texto_ejemplo = """
Python es un lenguaje de programación versátil.
Se utiliza en ciencia de datos, desarrollo web y automatización.
El curso tiene 50 estudiantes y 5 profesores!
Más información en https://ejemplo.com
Contacto: info@ejemplo.com
Python es muy popular. Python es fácil de aprender.
"""

```

```

metricas = analizar_texto(texto_ejemplo)
print("Análisis del texto:")
print(f"Oraciones: {metricas['oraciones']}")
print(f"Palabras: {metricas['palabras']}")
print(f"Palabras más comunes: {metricas['palabras_comunes']}")
print(f"Números encontrados: {metricas['numeros']}")
print(f"Emails encontrados: {metricas['emails']}")
print(f"URLs encontradas: {metricas['urls']}")

```

3.4.6 Flags (modificadores)

Las expresiones regulares en Python admiten varios flags que modifican su comportamiento:

re.IGNORECASE o **re.I** Ignora mayúsculas/minúsculas
re.MULTILINE o **re.M** ^ y \$ coinciden con inicio/fin de cada línea
re.DOTALL o **re.S** . coincide con cualquier carácter, incluyendo saltos de línea
re.VERBOSE o **re.X** Permite escribir patrones más legibles con espacios y comentarios

```

import re

# IGNORECASE
texto = "Python python PYTHON"
print(re.findall(r"python", texto, re.IGNORECASE))

# MULTILINE
texto_multilinea = """Primera línea
Segunda línea
Tercera línea"""
print(re.findall(r"^.*línea", texto_multilinea, re.MULTILINE))

# VERBOSE: patrón más legible
patron_email = re.compile(
    r"""
    ^                # Inicio de la cadena
    [a-zA-Z0-9._%+~]+  # Usuario
    @                # @
    [a-zA-Z0-9.-]+    # Dominio
    \.                # .
    [a-zA-Z]{2,}      # Extensión
    $                # Fin de la cadena
    """,
    re.VERBOSE,
)

print(patron_email.match("usuario@ejemplo.com"))

```

3.4.7 Ejercicios prácticos

3.4.7.1 Ejercicio 1: Validador de contraseñas

```

import re

```

```
def validar_contraseña(password):
    """
    Valida que una contraseña cumpla con:
    - Al menos 8 caracteres
    - Al menos una letra mayúscula
    - Al menos una letra minúscula
    - Al menos un dígito
    - Al menos un carácter especial
    """
    if len(password) < 8:
        return False, "Debe tener al menos 8 caracteres"

    if not re.search(r"[A-Z]", password):
        return False, "Debe tener al menos una mayúscula"

    if not re.search(r"[a-z]", password):
        return False, "Debe tener al menos una minúscula"

    if not re.search(r"\d", password):
        return False, "Debe tener al menos un dígito"

    if not re.search(r"[!@#$%^&*(),.?\"':{}|<>]", password):
        return False, "Debe tener al menos un carácter especial"

    return True, "Contraseña válida"

# Pruebas
contraseñas = ["123456", "abcdefgh", "Abcdefgh", "Abcdefgh1", "Abcdefgh1!"]

for pwd in contraseñas:
    valida, mensaje = validar_contraseña(pwd)
    print(f"{pwd}: {mensaje}")
```

3.4.7.2 Ejercicio 2: Extractor de información de texto

```
import re

def extraer_informacion_contacto(texto):
    """Extrae emails, teléfonos y URLs de un texto"""

    # Patrón para emails
    email_pattern = (
        r"\b[A-Za-z0-9._%+-]+"
        r"@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b"
    )
    emails = re.findall(email_pattern, texto)

    # Patrón para teléfonos (varios formatos)
    telefonos = re.findall(
        r"\+?\d{1,3}?[-.\s]?(\d{1,4})?[-.\s]?d{1,4}[-.\s]?d{1,9}", texto
    )

    # Patrón para URLs
    urls = re.findall(r"https?://[^\s]+", texto)

    return {"emails": emails, "telefonos": telefonos, "urls": urls}

texto_contacto = """
Para más información:
- Email: contacto@empresa.com o soporte@empresa.com.ar
- Teléfono: +54 11 4567-8900 o 555-1234
- Web: https://www.empresa.com y https://soporte.empresa.com/ayuda
"""
```

```

"""

info = extraer_informacion_contacto(texto_contacto)
print("Información extraída:")
print(f"Emails: {info['emails']}")
print(f"Teléfonos: {info['telefonos']}")
print(f"URLs: {info['urls']}")

```

3.4.7.3 Ejercicio 3: Procesador de menciones y hashtags

```

import re

def procesar_tweet(texto):
    """Extrae menciones y hashtags de un texto estilo Twitter"""

    # Menciones: @usuario
    menciones = re.findall(r"@(\w+)", texto)

    # Hashtags: #etiqueta
    hashtags = re.findall(r"#(\w+)", texto)

    # URLs
    urls = re.findall(r"https?://\S+", texto)

    # Texto limpio (sin menciones, hashtags ni URLs)
    texto_limpio = re.sub(r"@(\w+)|#(\w+)|https?://\S+", "", texto)
    texto_limpio = re.sub(r"\s+", " ", texto_limpio).strip()

    return {
        "menciones": menciones,
        "hashtags": hashtags,
        "urls": urls,
        "texto_limpio": texto_limpio,
    }

tweet = """
Gran clase de @profesorEDD sobre #ExpresionesRegulares!
#Python es genial para #DataScience
Más info: https://ejemplo.com/curso
cc: @estudiante1 @estudiante2
"""

resultado = procesar_tweet(tweet)
print("Análisis del tweet:")
print(f"Menciones: {resultado['menciones']}")
print(f"Hashtags: {resultado['hashtags']}")
print(f"URLs: {resultado['urls']}")
print(f"Texto limpio: {resultado['texto_limpio']}")

```

3.4.8 Rendimiento y buenas prácticas

3.4.8.1 Compilar patrones reutilizables

Cuando se usa un patrón múltiples veces, es más eficiente compilarlo:

```

import re
import time

texto = "Python es genial. Python es versátil. Python es popular." * 1000

# Sin compilar
start = time.time()
for _ in range(100):
    re.findall(r"Python", texto)
tiempo_sin_compilar = time.time() - start

```

```
# Con compilación
patron = re.compile(r"Python")
start = time.time()
for _ in range(100):
    patron.findall(texto)
tiempo_con_compilar = time.time() - start

print(f"Sin compilar: {tiempo_sin_compilar:.4f} segundos")
print(f"Con compilar: {tiempo_con_compilar:.4f} segundos")
print(f"Mejora: {tiempo_sin_compilar / tiempo_con_compilar:.2f}x más rápido")
```

3.4.8.2 Evitar backtracking excesivo

Algunas expresiones pueden causar backtracking excesivo y ser muy lentas:

```
import re

# Patrón ineficiente con backtracking
# patron_malo = r"(a)+" # Puede ser muy lento

# Mejor: usar cuantificadores específicos
patron_bueno = r"a+"

texto = "a" * 20 + "b"
print(re.search(patron_bueno, texto))
```

3.4.8.3 Usar raw strings

Siempre usar raw strings (r"...") para evitar problemas con caracteres de escape:

```
import re

# Sin raw string (necesita doble escape)
patron1 = "\\d+\\s+\\w+"

# Con raw string (más legible)
patron2 = r"\d+\s+\w+"

texto = "123 palabras"
print(re.search(patron1, texto).group())
print(re.search(patron2, texto).group())
```

3.4.9 Integración con estructuras de datos

Las expresiones regulares se integran naturalmente con las estructuras de datos de Python:

3.4.9.1 Índice invertido con regex

```
import re
from collections import defaultdict

class IndiceInvertido:
    """Índice invertido que usa regex para procesar documentos"""

    def __init__(self):
        self.indice = defaultdict(set)

    def agregar_documento(self, doc_id, texto):
        """Agrega un documento al índice"""
        # Tokenizar usando regex
        palabras = re.findall(r"\b\w+\b", texto.lower())

        for palabra in palabras:
            self.indice[palabra].add(doc_id)

    def buscar(self, patron):
```

```

        """Busca documentos que contengan palabras que coincidan
        con el patrón"""
        patron_compilado = re.compile(patron, re.IGNORECASE)
        documentos = set()

        for palabra in self.indice.keys():
            if patron_compilado.search(palabra):
                documentos.update(self.indice[palabra])

        return documentos

# Crear índice
indice = IndiceInvertido()
indice.agregar_documento(1, "Python es un lenguaje de programación")
indice.agregar_documento(2, "Programar en Python es divertido")
indice.agregar_documento(3, "Java y JavaScript son diferentes")

# Buscar documentos que contengan palabras que empiecen con "prog"
docs = indice.buscar(r"^prog")
print(f"Documentos con palabras que empiezan con 'prog': {docs}")

# Buscar documentos con palabras que contengan "python"
docs = indice.buscar(r"python")
print(f"Documentos con 'python': {docs}")

```

3.4.9.2 Filtrado de listas con regex

```

import re

# Lista de archivos
archivos = [
    "documento1.txt",
    "imagen.png",
    "datos.csv",
    "reporte_2024.pdf",
    "script.py",
    "backup_2024_03_15.zip",
]

def filtrar_archivos(archivos, patron):
    """Filtra archivos que coincidan con el patrón"""
    patron_compilado = re.compile(patron)
    return [
        archivo for archivo in archivos if patron_compilado.search(archivo)
    ]

# Filtrar archivos de texto
print("Archivos .txt:", filtrar_archivos(archivos, r"\.txt$"))

# Filtrar archivos de 2024
print("Archivos de 2024:", filtrar_archivos(archivos, r"2024"))

# Filtrar archivos Python
print("Archivos Python:", filtrar_archivos(archivos, r"\.py$"))

```

3.4.10 Tablas de referencia rápida

Descargar la versión imprimible

3.4.10.1 Caracteres

Expresión	Significado	Ejemplo	Match
-----------	-------------	---------	-------

Expresión	Significado	Ejemplo	Match
\d	En la mayoría de los lenguajes un dígito 0..9	file_\d\d	file_25
	En Python 3 y .Net un dígito Unicode	file_\d\d	file_2ᳵ
\w	En la mayoría de los lenguajes, un carácter de palabra: letra, dígito o ‘_’	\w-\w\w\w	A-f_3
	En Python 3, un símbolo Unicode de palabra, incluye ‘_’	\w-\w\w\w	字-ま_\𐀀
	En .NET, un símbolo Unicode de palabra, incluye conector ‘_’	\w-\w\w\w	字-ま_𐀀
\s	En la mayoría de los lenguajes caracteres de blanco estándar	a\s b\s c	a b c
	En la .NET, Python 3, Javascript, caracteres de blanco Unicode	a\s b\s c	a b c
\D	Un caracter que no es un dígito \d del lenguaje	\D\D\D	ABC
\W	Un caracter que no es un caracter de palabra \w del lenguaje	\W\W\W\W	*+=)
\S	Un caracter que no es un blanco estandar \s del lenguaje	\S\S\S\S	casa
.	Cualquier caracter, excepto saltos de líneas	a.c	abc
		.*	piso 2, depto "A"
\.	Un punto	\w\.\d	a.3
\	Escape de caracteres especiales	*\?\\$\^\	*?\\$^
		\[\{\(\)\}\]	[{()}]

3.4.10.2 Cuantificadores

Expresión	Significado	Ejemplo	Match
+	Una o más apariciones	\w-\w+	C-125x_1
{3}	Exactamente tres apariciones	\D{3}	ANA
{2,4}	Entre dos y cuatro apariciones	\W{2,4}	{+}
*	Cero o más aparaciones	A*B*C*	AAAACCCC
?	Cero o una aparición	casas?	casa

3.4.10.3 Lógica

Expresión	Significado	Ejemplo	Match
	Or	22 33	22
(...)	Captura un grupo y lo asocia a una variable numerada	UN(0 TREF)	UNTREF (y captura TREF)
\1	Lo capturado en el grupo 1	r(\w)g\1\w	regex
\2	Lo capturado en el grupo 2	(\d+)+(\d+)=\2+\1	25+33=33+25
(?:...)	Grupo que no se captura (se verifica la regex pero no se captura)	A(?:na licia)	Alicia

3.4.10.4 Clases de caracteres

Expresión	Significado	Ejemplo	Match
[...]	Uno de los caracteres entre corchetes	[AEIOU]	A
-	Indicador de rango	[a-z]	Una letra minúscula
		[A-Z]+	Una o más letras mayúsculas
		[AB1-5w-z]	Uno de los caracteres AB12345wxyz
[^x]	Cualquier caracter distinto de x	A[^a]B	AxB
[^x-y]	Cualquier caracter fuera del rango x-y	[^a-z]{3}	A1!
[\xhh]	El caracter con código hh en hexadecimal de la tabla de símbolos ASCII	[\x41-\x45]{3}	ABE

3.4.10.5 Posiciones: fronteras y anclas

Expresión	Significado	Ejemplo	Match
^	Indicador de comienzo de cadena (o comienzo de línea). Tiene que estar fuera de [] (ya que adentro de [] significa negación)	^abc.*	Texto que empieza con abc
\$	Fin de cadena o fin de línea	.*el final\.\$	Texto que termina en el final.
\b	Frontera de la palabra	Bibi.*\bes\b.*	Bibi es mi amiga
\B	No es frontera de palabra	Bibi.*\Bes\B.*	Bibi usa un vestido

3.4.10.6 Miradas alrededor (*look behind* y *look ahead*)

No consumen caracteres, se quedan paradas donde ocurrió el matching

Expresión	Significado	Ejemplo	Match
(?=...)	Mirar hacia adelante con parámetro positivo	(?=\d{10})\d{5}	Si hacia adelante hay 10 dígitos matchear los primeros 5
(?<=...)	Mirar hacia atrás con parámetro positivo	(?<=foo).*	Si lo que está justo detrás de la posición actual es la cadena foo. El matching es todo lo que sigue a foo
(?!...)	Mirar hacia adelante con parámetro negativo	q(?!ue)	matchea una q no este seguida de ue
		(?!teatro)te\w+	cualquier palabra que empiece con te pero no sea teatro
(?<!...)	Mirar hacia atrás con parámetro negativo	(?<!fut)bol	bol siempre y cuando no esté precedida por fut

3.4.11 Recursos adicionales

Para profundizar en expresiones regulares:

- Documentación oficial de re en Python
- Regular Expression HOWTO
- Regex101 - Herramienta online para probar regex
- RegExr - Otra herramienta interactiva
- Python Regular Expressions (Real Python)

Resumen

Las expresiones regulares son una herramienta poderosa para:

- Validar formatos de datos
- Extraer información de textos no estructurados
- Limpiar y normalizar datos
- Procesar logs y archivos de texto
- Implementar búsquedas avanzadas en sistemas de recuperación de información

El dominio de las expresiones regulares es esencial para trabajar con grandes volúmenes de texto y construir sistemas eficientes de procesamiento de información.

3.5 La Web y las APIs

La Web es una de las fuentes de información más grandes y diversas disponibles en la actualidad. Contiene datos estructurados, semiestructurados y no estructurados que pueden ser aprovechados para múltiples propósitos: análisis de datos, investigación, monitoreo de precios, agregación de noticias, entre otros. En este capítulo exploraremos el funcionamiento de la Web y cómo acceder a información a través de APIs.

3.5.1 Introducción al Funcionamiento de la Web

3.5.1.1 Arquitectura Cliente-Servidor

La World Wide Web funciona bajo un modelo cliente-servidor. El siguiente diagrama ilustra cómo interactúan estos componentes:

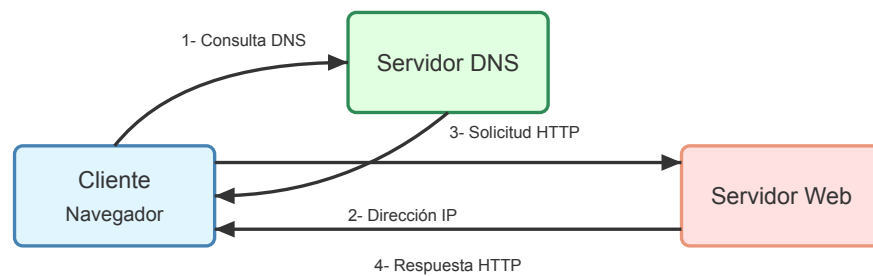


Figure 3.2: Arquitectura Cliente-Servidor

Los componentes principales son:

Cliente (Navegador) Programa que solicita recursos web (páginas HTML, imágenes, videos, etc.). Los navegadores más comunes son Chrome, Firefox, Safari y Edge.

Servidor Web Aplicación que responde a las solicitudes de los clientes, enviando los recursos solicitados. Ejemplos incluyen Apache, Nginx, y servidores de aplicaciones como Node.js o Python con frameworks como Django o FastAPI.

DNS (Domain Name System) Sistema que traduce nombres de dominio legibles (como `www.untref.edu.ar`) a direcciones IP numéricas que los computadores pueden entender.

En primer lugar el cliente o browser realiza una consulta DNS para obtener la dirección IP del servidor web asociado al dominio. Luego, el cliente envía una solicitud HTTP al servidor, que procesa la solicitud y devuelve una respuesta con el recurso solicitado.

Hoy en día a través de la web no solo se puede obtener páginas HTML, sino también se pueden ejecutar aplicaciones web completas, donde el servidor puede enviar datos y código (generalmente JavaScript) que se ejecuta en el navegador del cliente, permitiendo interfaces interactivas y dinámicas.

3.5.1.2 El Protocolo HTTP

HTTP (*HyperText Transfer Protocol*) es el protocolo de comunicación que permite la transferencia de información en la Web. Define cómo los clientes y servidores intercambian mensajes.

HTTP fue diseñado para ser simple y flexible, permitiendo la transferencia de diferentes tipos de datos (HTML, JSON, imágenes, etc.) a través de una estructura de mensajes estándar.

El protocolo HTTP sigue un modelo de solicitud-respuesta (*request-response*), donde el cliente envía una solicitud al servidor y este responde con el recurso solicitado o un mensaje de error.

En el siguiente diagrama de secuencia se muestra una interacción típica entre un cliente y un servidor utilizando HTTP:

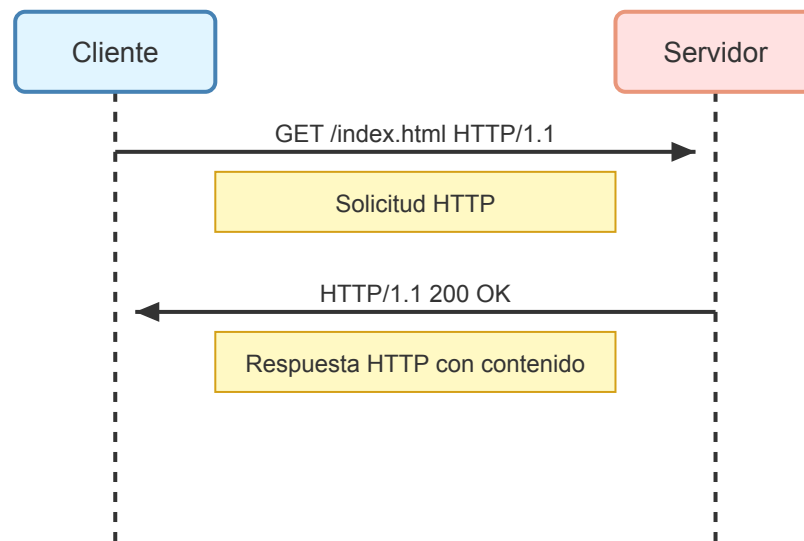


Figure 3.3: Interacción HTTP Cliente-Servidor

Petición enviada:

```
GET /contact HTTP/1.1
Host: example.com
User-Agent: curl/8.6.0
Accept: */*
```

La petición HTTP consta de varias líneas, donde la primera línea indica el método HTTP (GET), el recurso solicitado (/contact) y la versión del protocolo (HTTP/1.1). Las líneas siguientes son las cabeceras (*headers*) que proporcionan información adicional sobre la solicitud. En este caso, se especifica el host, el agente de usuario (navegador) y los tipos de contenido aceptados.

Respuesta recibida:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Date: Fri, 21 Jun 2024 14:18:33 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Content-Length: 1234
```

```
<!doctype html>
<!-- HTML content follows -->
```

La respuesta HTTP también consta de varias líneas, donde la primera línea indica el protocolo que usa el servidor y el código de estado (200 OK), seguido de las cabeceras de respuesta y finalmente el cuerpo del mensaje que contiene el HTML de la página.

3.5.1.2.1 Características de HTTP

Sin estado (stateless) Cada solicitud es independiente, el servidor no mantiene información sobre solicitudes anteriores (algunos servidores implementan algunos mecanismos para manejar sesiones, pero esto no es parte del protocolo HTTP en sí).

Basado en texto Los mensajes son legibles por humanos

Extensible Permite agregar nuevos métodos y cabeceras

Cliente-Servidor Modelo de comunicación request-response

3.5.1.2.2 Métodos HTTP Principales

Los métodos HTTP definen las peticiones que se puede solicitar al servidor sobre un recurso específico:

GET Solicita un recurso específico. Es el método más común para solicitar un recurso al servidor. Si no se especifica un recurso en particular, el servidor generalmente devuelve la página principal, normalmente `index.html`.

POST Envía datos al servidor para crear un nuevo recurso. Comúnmente usado en formularios.

PUT Actualiza un recurso existente con datos nuevos.

DELETE Elimina un recurso específico.

HEAD Similar a GET, pero solo solicita las cabeceras de la respuesta, sin el cuerpo.

PATCH Aplica modificaciones parciales a un recurso.

3.5.1.2.3 Códigos de Estado HTTP

Las respuestas HTTP incluyen un código de estado que indica el resultado de la solicitud:

2xx (Éxito)

- 200 OK: Solicitud exitosa
- 201 Created: Recurso creado exitosamente
- 204 No Content: Éxito pero sin contenido para devolver

3xx (Redirección)

- 301 Moved Permanently: El recurso se ha movido permanentemente
- 302 Found: Redirección temporal
- 304 Not Modified: El recurso no ha sido modificado desde la última solicitud

4xx (Error del Cliente)

- 400 Bad Request: Solicitud mal formada
- 401 Unauthorized: Autenticación requerida
- 403 Forbidden: Acceso denegado
- 404 Not Found: Recurso no encontrado
- 429 Too Many Requests: Límite de velocidad excedido

5xx (Error del Servidor)

- 500 Internal Server Error: Error genérico del servidor
- 502 Bad Gateway: El servidor actuó como gateway y recibió una respuesta inválida
- 503 Service Unavailable: Servidor no disponible temporalmente

3.5.1.2.4 Ejemplo de Solicitud HTTP con Python

En Python, la biblioteca `requests` facilita la realización de solicitudes HTTP. Aquí hay un ejemplo básico de cómo hacer una solicitud GET:

```
import requests

# Realizar una solicitud GET
response = requests.get("https://untref.edu.ar/", timeout=10)

print(f"Código de estado: {response.status_code}")
for k, v in response.headers.items():
    print(f"{k}: {v}")
print(f"\nPrimeros 200 caracteres del contenido (página html):")
print(f"{response.text[:200]}")
```

El intercambio entre cliente y servidor puede verse en “*crudo*” utilizando herramientas como `curl` en la línea de comandos. Aquí hay un ejemplo de cómo se vería una solicitud y respuesta HTTP:

```
curl -v https://untref.edu.ar/
```

Nota

cURL es una herramienta de línea de comandos, gratuita y de código abierto, para transferir datos usando diversas URLs y protocolos, comúnmente utilizada para interactuar con APIs, descargar archivos y probar recursos web. Es compatible con una amplia gama de protocolos como HTTP, HTTPS, FTP y SMB. cURL está disponible de forma nativa en sistemas operativos basados en Unix, incluyendo Linux y macOS, y está preinstalado en las versiones modernas de Windows.

Petición enviada:

```
GET / HTTP/1.1
Host: www.untref.edu.ar
User-Agent: curl/8.12.1
Accept: */*
```

Respuesta recibida:

```
HTTP/1.1 200 OK
Date: Mon, 06 Oct 2025 15:06:59 GMT
Server: Apache
Cache-Control: no-cache
Set-Cookie: XSRF-
TOKEN=eyJpdiI6IkxZbnpaYmpEV0ZaMlZaRVVwYUQzbDZRPT0iLCJ2YWx1ZSI6ImtcL240ZFRjS1BhYWZmZWNCLOt3
expires=Mon, 06-Oct-2025 17:07:00 GMT; Max-Age=7200; path=/
Set-Cookie:
laravel_session=eyJpdiI6IkxIeHIyRElDdm82bFVlWDdrRTRickeE9PSIsInZhbnVlIjoIYTRMZklQZDk2RTgwMj
expires=Mon, 06-Oct-2025 17:07:00 GMT; Max-Age=7200; path=/; HttpOnly
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
<!DOCTYPE html>
<html lang="es">

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

  <title>UNTREF</title>
  ...
```

3.5.1.3 HTTPS: HTTP Seguro

HTTPS (HTTP Secure) es la versión segura de HTTP que utiliza encriptación TLS/SSL para proteger la comunicación entre cliente y servidor. Es fundamental para:

- Proteger datos sensibles (contraseñas, información financiera)
- Verificar la identidad del servidor
- Prevenir ataques de intermediarios (man-in-the-middle)
- Mejorar el posicionamiento en motores de búsqueda

3.5.2 APIs: Interfaces de Programación de Aplicaciones

Las APIs (*Application Programming Interfaces*) son interfaces que permiten que diferentes aplicaciones se comuniquen entre sí de manera programática. En el contexto web, las APIs proporcionan puntos de acceso (*endpoints*) que los desarrolladores pueden usar para acceder a datos y funcionalidades de un servicio.

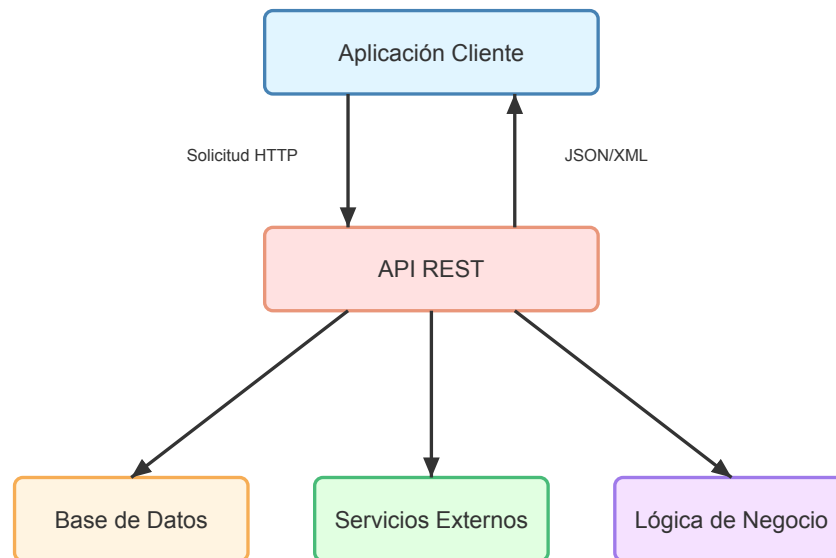


Figure 3.4: Arquitectura de una API REST

Una aplicación cliente (por ejemplo una aplicación web o móvil) realiza solicitudes HTTP a una API REST, que procesa la solicitud, interactúa con bases de datos u otros servicios, y devuelve los datos en formatos como JSON o XML.

El protocolo base es HTTP, y los recursos se acceden a través de URLs específicas. Por ejemplo una URL típica de una API REST podría ser:

`https://api.ejemplo.com/v1/usuarios/123`

Donde:

- `https://api.ejemplo.com` es el dominio de la API. Es decir el servidor donde está alojada la API.
- `/v1` indica la versión de la API. Un servidor puede tener múltiples versiones de una API para mantener compatibilidad con clientes antiguos.
- `/usuarios/123` es el recurso específico (usuario con ID 123).

Ante esta solicitud, la API podría devolver un JSON con los datos del usuario:

```
{
  "id": 123,
  "nombre": "Juan Pérez",
  "email": "juan.perez@ejemplo.com"
}
```

3.5.2.1 API REST (Representational State Transfer)

REST es un estilo arquitectónico para diseñar servicios web que se basa en los principios de HTTP. Una API REST expone recursos a través de URLs y utiliza los métodos HTTP estándar para operaciones CRUD (Create, Read, Update, Delete).

3.5.2.1.1 Principios de REST

Arquitectura Cliente-Servidor Separación de responsabilidades entre interfaz de usuario y almacenamiento de datos.

Sin Estado Cada solicitud contiene toda la información necesaria; el servidor no mantiene sesiones.

Cacheable Las respuestas deben indicar si pueden ser almacenadas en caché.

Interfaz Uniforme Uso consistente de URLs y métodos HTTP.

Sistema en Capas La arquitectura puede tener múltiples capas intermedias.

3.5.2.1.2 Ejemplo: Consumir una API REST Pública

Vamos a consumir una API pública para consultar resultados electorales de Argentina, disponible en <https://resultados.mininterior.gob.ar>.

```
import requests
import json

# Realizar una solicitud GET a la API del Ministerio del Interior
response = requests.get(
    "https://resultados.mininterior.gob.ar/api/resultados/getResultados?"
    "anioEleccion=2019&tipoRecuento=1&tipoEleccion=2&categoriaId=1&"
    "distritoId=2&seccionProvincialId=1&seccionId=118",
    timeout=10
)
if response.status_code == 200:
    datos = response.json()
    print(json.dumps(datos, indent=2, ensure_ascii=False))
else:
    print(f"Error al acceder a la API: {response.status_code}")
    print(json.dumps(response.json(), indent=2, ensure_ascii=False))
```

En la solicitud anterior, se puede ver que se utilizan varios parámetros en la URL para especificar los datos que se quieren consultar

Parámetro	Valor	Significado
anioEleccion=2019	2019	Año de la elección consultada.
tipoRecuento=1	1	1 = Recuento Provisional
tipoEleccion=2	2	2 = Elecciones Generales
categoriaId=1	1	1 = Presidente de la Nación.
distritoId=2	2	2 = Provincia de Buenos Aires.
seccionProvincialId=1	1	1 = Primera Sección Electoral.
seccionId=118	118	118 = Tres de Febrero.

La documentación de la API se puede descargar desde el sitio oficial del Ministerio del Interior.

La respuesta de la API es un JSON con los resultados detallados para Tres de Febrero.

3.5.2.1.3 Ejemplo: API de Información Geográfica

Muchas APIs REST proporcionan datos estructurados útiles. Vamos a consultar OpenStreetMap (OSM) que ofrece datos geográficos.

Nota

OpenStreetMap es un proyecto colaborativo para crear un mapa libre y editable del mundo. Los datos son aportados por voluntarios y están disponibles bajo la licencia Open Database License (ODbL).

```
import requests
from lxml import etree as ET

# Realizar una solicitud GET a la API de Open Maps
# Way Id = 1275831310 (Sede Caseros I de la UNTREF)
response = requests.get(
    "https://api.openstreetmap.org/api/0.6/way/1275831310", timeout=10
)
if response.status_code == 200:
    # Parsear la respuesta XML
    root = ET.fromstring(response.content)
```

```
# Recorrer el XML de OpenStreetMap

# Buscar el elemento <way>
way = root.find("way")
if way is not None:
    print(f"ID del way: {way.get('id')}")
    print("Etiquetas asociadas:")
    for tag in way.findall("tag"):
        clave = tag.get("k")
        valor = tag.get("v")
        print(f"  {clave}: {valor}")
else:
    print("No se encontró el elemento <way> en la respuesta.")
```

La documentación de la API de OpenStreetMap está disponible en https://wiki.openstreetmap.org/wiki/API_v0.6.

Con el way ID 1275831310 también se puede obtener el mapa correspondiente a través del servicio Overpass API, que permite consultas más complejas. Aquí hay un ejemplo de cómo obtener la geometría del way en formato GeoJSON y visualizarlo en un mapa interactivo usando la librería folium:

```
import requests
import folium

def obtener_geojson_way(osm_way_id):
    # Consulta Overpass para el way específico
    url = "https://overpass-api.de/api/interpreter"
    # Query Overpass: way + nodos + metadata
    query = f"""
[out:json];
way({osm_way_id});
out body;
>;
out meta;
"""
    response = requests.get(url, params={"data": query})
    response.raise_for_status()
    return response.json()

def construir_mapa(geojson_data):
    # Extraer nodos del way y sus coordenadas
    # Overpass pone los nodos como elementos tipo "node" en el array
    "elements"
    nodes = {}
    for el in geojson_data.get("elements", []):
        if el["type"] == "node":
            nodes[el["id"]] = (el["lat"], el["lon"])
    # Construir lista ordenada de coordenadas del way
    coords = []
    for el in geojson_data.get("elements", []):
        if el["type"] == "way":
            for nid in el["nodes"]:
                if nid in nodes:
                    coords.append(nodes[nid])
    # Centrar el mapa en la primera coordenada
    if not coords:
        raise RuntimeError("No se hallaron coordenadas del way")
    centro = coords[0]
    mapa = folium.Map(location=centro, zoom_start=18)
    # Añadir polígono (o línea) al mapa
    folium.PolyLine(locations=coords, color="blue", weight=3).add_to(mapa)
    # También podrías usar folium.Polygon si es cerrado
    return mapa
```

```
# Visualizar el mapa en el notebook

def main():
    osm_way_id = 1275831310 # el way de la Sede Caseros I
    geojson = obtener_geojson_way(osm_way_id)
    mapa = construir_mapa(geojson)
    # Mostrar el mapa en el notebook
    display(mapa)

if __name__ == "__main__":
    main()
```

El fragmento de código anterior realiza los siguientes pasos:

1. Extraer coordenadas de los nodos que forman el “way” desde un objeto GeoJSON.
2. Centrar el mapa en la primera coordenada encontrada.
3. Dibujar la línea (o polígono) sobre el mapa usando folium.PolyLine.
4. Mostrar el mapa en un entorno interactivo (como Jupyter Notebook) usando display(mapa).

La función principal (main) obtiene el GeoJSON de un “way” específico, en este caso la Sede Caseros I, construye el mapa y lo muestra.

overpass-api.de es un servicio web que permite consultar y extraer datos de OpenStreetMap mediante un lenguaje de consultas específico (Overpass QL). Se usa para obtener información geográfica detallada, como nodos, caminos y relaciones, de la base de datos de OSM.

Nota

GeoJSON es un formato basado en JSON para representar datos geográficos. Define varias estructuras como Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon y GeometryCollection para describir diferentes tipos de geometrías espaciales.

Los servicios que ofrece OpenStreetMap se pueden consultar en su wiki.

3.5.2.1.4 Autenticación en APIs REST

Las APIs frecuentemente requieren autenticación, sobre todo cuando se trata de servicios **privados**. Los métodos más comunes son:

API Keys Una clave secreta que se envía en la URL o en las cabeceras.

OAuth 2.0 Protocolo de autorización más complejo pero seguro, usado por servicios como Google, Facebook, Twitter.

Bearer Tokens Tokens de acceso que se envían en el header de autorización.

En general, antes de poder consultar una API, es necesario registrarse y obtener las credenciales necesarias.

3.5.3 Mejores Prácticas para Usar APIs

1. **Leer la documentación:** Entender límites de velocidad, autenticación y términos de uso
2. **Manejar errores:** Implementar reintentos con backoff exponencial
3. **Cachear respuestas:** Evitar solicitudes repetidas
4. **Monitorear cuotas:** Estar atento a los límites de uso
5. **Versionar:** Usar versiones específicas de APIs para evitar cambios inesperados

3.5.4 Referencias y Recursos Adicionales

3.5.4.1 Documentación Oficial

- [Requests Documentation](#)
- [HTTP Documentation \(MDN\)](#)
- [REST API Tutorial](#)

3.5.4.2 APIs Públicas para Practicar

- [JSONPlaceholder](#) - API REST falsa para testing
- [REST Countries](#) - Información sobre países
- [OpenWeatherMap](#) - Datos meteorológicos
- [The Star Wars API](#) - Datos de Star Wars
- [PokéAPI](#) - Información sobre Pokémon

3.5.4.3 Libros y Referencias Académicas

- En el capítulo 19: Web Search Basics del libro (Manning et al., 2008) se presenta la estructura de la Web y los protocolos HTTP. Este libro se encuentra gratis en formato PDF y html en el sitio web de la Universidad de Stanford.

3.6 Web Scraping

Web scraping es el proceso de extraer información de sitios web de forma automatizada.

Mientras que las APIs proporcionan interfaces estructuradas para acceder a datos, el web scraping permite obtener información de sitios que no ofrecen APIs o cuando se necesita acceder a datos que no están disponibles a través de ellas.

Los artefactos que realizan web scraping se conocen comúnmente como “*scrapers*”, “*spiders*” o “*crawlers*”. Estos programas navegan por las páginas web, descargan su contenido HTML y extraen la información relevante.

Los buscadores web utilizan *crawlers* para indexar el contenido de la web y hacer que sea accesible a través de búsquedas. De alguna manera, los crawlers son la columna vertebral de los motores de búsqueda que permite a los buscadores descubrir y organizar la vasta cantidad de información disponible en Internet, almacenando en sus bases de datos no solo las URLs, sino también fragmentos de texto y metadatos asociados a cada página.

El siguiente diagrama ilustra la arquitectura básica de un *crawler*:

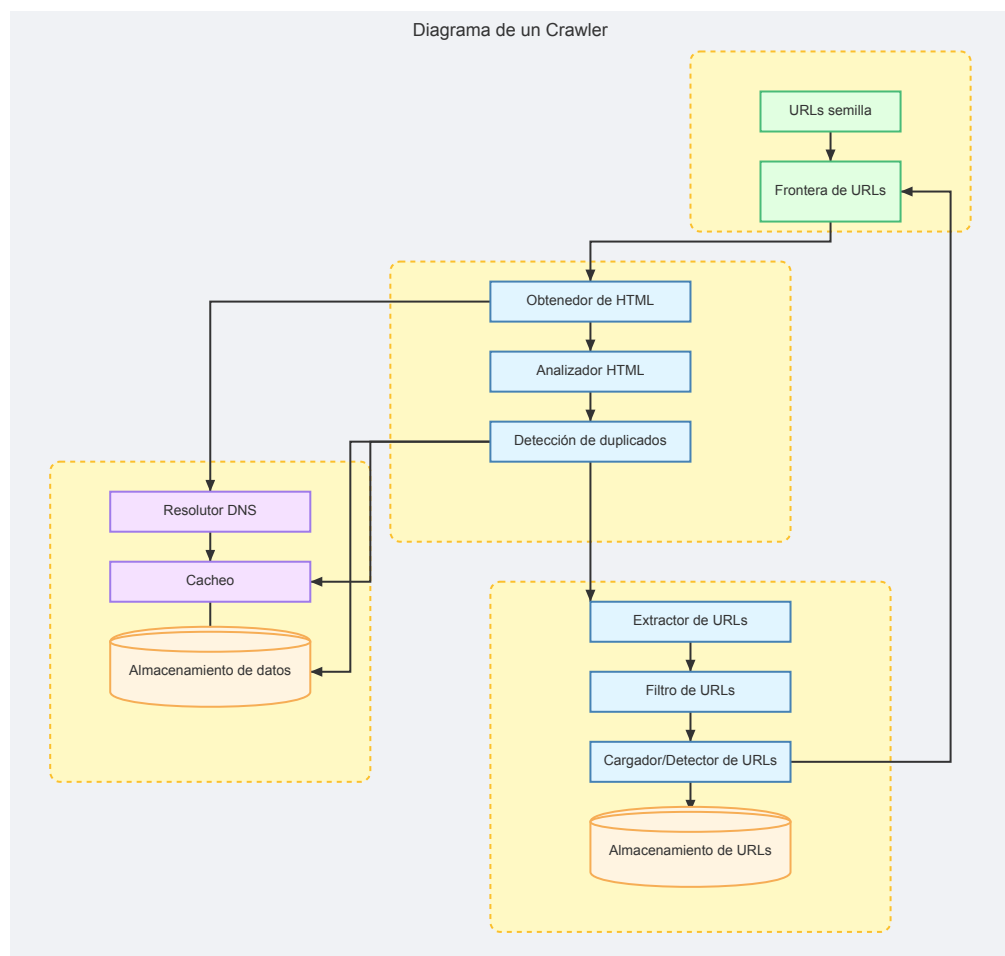


Figure 3.5: Arquitectura básica de un crawler

URLs semilla Puntos de partida para el crawler. Una serie de URLs iniciales desde donde comenzar la exploración.

Frontera de URLs Estructura de datos que almacena las URLs pendientes de visitar. Cada vez que el *crawler* visita una página, extrae nuevas URLs y las añade a esta frontera.

Obtenedor de HTML Componente que realiza solicitudes HTTP para descargar el contenido HTML de las páginas web.

Analizador HTML Procesa el HTML descargado para extraer información relevante, como texto, enlaces, imágenes, etc.

Detección de duplicados Módulo que verifica si una URL ya ha sido visitada para evitar procesarla nuevamente.

Extractor de URLs Extrae todas las URLs presentes en la página web analizada.

Filtro de URLs Aplica reglas para decidir qué URLs deben ser añadidas a la frontera (por ejemplo, solo URLs del mismo dominio).

Cargador/Detector de URLs Añade nuevas URLs a la frontera y marca las URLs visitadas.

Almacenamiento de URLs Base de datos o archivo donde se guardan las URLs visitadas y pendientes.

Resolutor DNS Convierte nombres de dominio en direcciones IP para realizar las solicitudes HTTP.

Cacheo Almacena temporalmente respuestas HTTP para mejorar la eficiencia y reducir la carga en los servidores web.

Almacenamiento de datos Base de datos o archivo donde se guardan los datos extraídos del contenido web.

El proceso de web scraping puede variar en complejidad dependiendo del sitio web objetivo y de los datos que se desean extraer. Algunos sitios pueden tener estructuras HTML simples, mientras que otros pueden utilizar JavaScript para cargar contenido dinámicamente, lo que requiere técnicas más avanzadas.

En general el proceso de búsqueda inicia con una lista de URLs semilla, que son las páginas iniciales que el crawler visitará. A partir de estas páginas, el crawler descarga el contenido HTML y lo analiza para extraer información relevante y nuevas URLs. Estas nuevas URLs se añaden a la frontera de URLs pendientes de visitar, y el proceso se repite hasta que se alcanzan ciertos límites, como un número máximo de páginas visitadas o una profundidad máxima de exploración.

Para gestionar la frontera de URLs, se pueden utilizar diferentes estructuras de datos como colas (FIFO) para una exploración en anchura o pilas (LIFO) para una exploración en profundidad. Además, es importante implementar mecanismos para evitar visitar la misma URL múltiples veces, lo que se puede lograr mediante el uso de conjuntos o bases de datos para rastrear las URLs ya visitadas.

También se pueden establecer reglas para filtrar las URLs que se añaden a la frontera, como limitar la exploración a un dominio específico o evitar ciertos tipos de contenido.

3.6.1 Consideraciones Legales y Éticas

Antes de realizar una exploración de la web, es fundamental considerar aspectos legales y éticos. Algunos sitios web prohíben el scraping en sus términos de servicio, y es importante respetar estas políticas para evitar problemas legales.

También es crucial ser respetuoso con los servidores web, evitando sobrecargar el sitio con demasiadas solicitudes en poco tiempo.

Los servidores pueden tener mecanismos para detectar y bloquear actividades sospechosas, como un número excesivo de solicitudes en un corto período.

En general las políticas de acceso a un sitio web por parte de los scrapers se regulan mediante:

robots.txt Archivo en la raíz del sitio web que especifica qué partes pueden ser accedidas por robots automatizados.

```
import requests
```

```
# Verificar el archivo robots.txt
url_robots = 'https://python.org/robots.txt'
response = requests.get(url_robots)
```

```
print("Contenido de robots.txt de python.org:")
print('\n'.join(response.text.split('\n')))
```

El formato típico de un archivo robots.txt incluye directivas como User-agent, Disallow, y Allow para controlar el acceso de diferentes tipos de bots a distintas partes del sitio web.

El protocolo Robots Exclusion Standard define cómo los bots deben interpretar estas directivas para respetar las políticas del sitio. Este protocolo se encuentra estandarizado a través de la RFC 9309.

Términos de Servicio Muchos sitios web prohíben explícitamente el scraping en sus términos de uso.

Leyes de Protección de Datos Regulaciones como GDPR en Europa o leyes locales de protección de datos personales.

Propiedad Intelectual El contenido scrapeado puede estar protegido por derechos de autor.

Identificarse correctamente Usar un User-Agent descriptivo que permita al administrador del sitio contactarte.

Uso responsable de los datos No usar los datos scrapeados para propósitos no éticos o ilegales.

3.6.2 Web Scraping Manual con Python

Python ofrece excelentes bibliotecas para web scraping. Las más populares son requests para realizar solicitudes HTTP y BeautifulSoup para parsear HTML.

3.6.2.1 Instalación de Bibliotecas

```
pip install requests beautifulsoup4 lxml
```

BeautifulSoup es una biblioteca para parsear documentos HTML y XML, facilitando la navegación y búsqueda de elementos dentro del árbol del documento.

3.6.2.2 Ejemplo Básico de un crawler

```
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse
import csv
import time

def es_mismo_dominio(url, dominio_base):
    """Verifica si la URL pertenece al mismo dominio base."""
    return urlparse(url).netloc == dominio_base

def crawler_frontera(url_semilla,
                    max_paginas=50,
                    retraso=1,
                    archivo_csv='enlaces.csv'):
    """
    Función para realizar crawling web utilizando una frontera de enlaces tipo
    FIFO (cola).
    Recorre páginas web comenzando desde una URL semilla, siguiendo enlaces
    encontrados hasta un máximo de páginas.

    Parámetros:
    - url_semilla (str): URL inicial desde donde comienza el crawling.
    - max_paginas (int, opcional): Número máximo de páginas a visitar
      (por defecto 50).
    - retraso (int o float, opcional): Tiempo de espera (en segundos) entre
      solicitudes para evitar sobrecargar el servidor (por defecto 1).
    - archivo_csv (str, opcional): Nombre del archivo CSV donde se
      guardarán los enlaces encontrados (por defecto 'enlaces.csv').
    """
    frontera = [url_semilla]
    visitadas = set()
    enlaces_extraidos = []

    dominio_base = urlparse(url_semilla).netloc

    while frontera and len(visitadas) < max_paginas:
        url_actual = frontera.pop(0)
```

```

if url_actual in visitadas:
    continue

print(f"Visitando: {url_actual}")
try:
    response = requests.get(url_actual, timeout=10, headers={
        'User-Agent': 'MiCrawler/1.0 (contacto@ejemplo.com)'
    })
    response.raise_for_status()
except Exception as e:
    print(f" Error al acceder: {e}")
    continue

soup = BeautifulSoup(response.text, 'lxml')
visitadas.add(url_actual)

# Extraer y guardar enlaces
for enlace in soup.find_all('a', href=True):
    # En una página html los enlaces están en etiquetas <a href="...">
    url_encontrada = urljoin(url_actual, enlace['href'])
    url_encontrada = url_encontrada.split('#')[0] # Quitar fragmentos
    if es_mismo_dominio(url_encontrada, dominio_base):
        if url_encontrada not in visitadas \
            and url_encontrada not in frontera:
            frontera.append(url_encontrada)
            enlaces_extraidos.append({'pagina': url_actual,
                                      'enlace': url_encontrada})

time.sleep(retraso) # Ser respetuoso con el servidor

# Guardar enlaces en un archivo CSV
with open(archivo_csv, 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=['pagina', 'enlace'])
    writer.writeheader()
    writer.writerows(enlaces_extraidos)

print(f"\nTotal de páginas visitadas: {len(visitadas)}")
print(f"Enlaces guardados en: {archivo_csv}")

# Ejemplo de uso:
crawler_frontera('https://quotes.toscrape.com/', max_paginas=10,
                 archivo_csv='enlaces_quotes.csv')

```

Contenido de enlaces_quotes.csv

A continuación se muestra el contenido del archivo generado:

pagina	enlace
https://quotes.toscrape.com/	https://quotes.toscrape.com/
https://quotes.toscrape.com/	https://quotes.toscrape.com/login
https://quotes.toscrape.com/	https://quotes.toscrape.com/author/Albert-Einstein
https://quotes.toscrape.com/	https://quotes.toscrape.com/tag/change/page/1/
https://quotes.toscrape.com/	https://quotes.toscrape.com/tag/deep-thoughts/page/1/
https://quotes.toscrape.com/	https://quotes.toscrape.com/tag/thinking/page/1/
https://quotes.toscrape.com/	https://quotes.toscrape.com/tag/world/page/1/
https://quotes.toscrape.com/	https://quotes.toscrape.com/author/J-K-Rowling

```
| https://quotes.toscrape.com/ | https://quotes.toscrape.com/tag/abilities/
page/1/ |
| https://quotes.toscrape.com/ | https://quotes.toscrape.com/tag/choices/page/
1/ |
```

3.6.3 Scrapy

Scrapy es un *framework* de Python para web scraping a gran escala. Proporciona funcionalidades avanzadas como:

- Gestión automática de solicitudes concurrentes
- Manejo de robots.txt
- Extracción de datos con selectores CSS y XPath
- Exportación a múltiples formatos (JSON, CSV, XML)
- Middleware para personalizar el comportamiento

3.6.3.1 Instalación de Scrapy

```
pip install scrapy
```

Anatomía de un Spider en Scrapy

Una *spider* en Scrapy es una clase que hereda de `scrapy.Spider` y define cómo navegar y extraer información de un sitio web. Los elementos clave que se deben configurar son:

- **name**: Identificador único de la spider dentro del proyecto.
- **allowed_domains**: Lista de dominios permitidos para evitar que la spider navegue fuera del sitio objetivo.
- **start_urls**: Lista de URLs iniciales desde donde comenzará el scraping.
- **custom_settings** (*opcional*): Permite definir configuraciones específicas para esta spider, como el retraso entre descargas, el User-Agent, el respeto a robots.txt, número de solicitudes concurrentes, etc.
- **parse**: Método principal que procesa la respuesta de cada URL y define cómo extraer los datos o seguir enlaces adicionales.

Ejemplo básico:

```
import scrapy

class MiSpider(scrapy.Spider):
    name = 'mi_spider'
    allowed_domains = ['ejemplo.com']
    start_urls = ['https://ejemplo.com/']

    custom_settings = {
        'DOWNLOAD_DELAY': 1, # Espera 1 segundo entre solicitudes
        'ROBOTSTXT_OBEY': True, # Respeto robots.txt
        'USER_AGENT': 'MiSpider/1.0 (contacto@ejemplo.com)'
    }

    def parse(self, response):
        # Lógica de extracción de datos
        for elemento in response.css('div.item'):
            yield {
                'titulo': elemento.css('h2::text').get(),
                'enlace': elemento.css('a::attr(href)').get(),
            }
        # Seguir enlaces a otras páginas si es necesario
        siguiente = response.css('a.siguiente::attr(href)').get()
        if siguiente:
            yield response.follow(siguiente, self.parse)
```

Resumen de configuración esencial:

- Define el nombre y dominios permitidos.

- Especifica las URLs de inicio.
- Ajusta `custom_settings` para controlar el comportamiento de la spider.
- Implementa el método `parse` para extraer y procesar la información. En el ejemplo, se extraen títulos y enlaces de elementos con la clase `item`, y se sigue un enlace de paginación si está presente.

Para spiders más avanzados, se pueden sobrescribir otros métodos o definir múltiples funciones de parseo según la estructura del sitio.

3.6.4 Proyecto Práctico: Spider de Libros con Scrapy

A continuación se presenta un tutorial paso a paso para crear un spider con **Scrapy** que visite el sitio Books to Scrape y genere un archivo CSV con títulos y precios de los libros de la categoría “Horror”.

3.6.4.1 Paso 1: Crear un Proyecto Scrapy

Crear un nuevo proyecto de Scrapy en el directorio actual:

Iniciar un nuevo proyecto y una spider:

```
scrapy startproject books_scraper
cd books_scraper
scrapy genspider books books.toscrape.com
```

3.6.4.2 Paso 2: Estructura del Proyecto

El comando anterior crea la siguiente estructura de directorios:

```
books_scraper/
  scrapy.cfg
  books_scraper/
    __init__.py
    items.py
    middlewares.py
    pipelines.py
    settings.py
    spiders/
      __init__.py
      books.py
```

3.6.4.3 Paso 3: Definir los Items

Editar el archivo `items.py` para definir la estructura de datos que queremos extraer:

```
# books_scraper/items.py
import scrapy

class BookItem(scrapy.Item):
    title = scrapy.Field()
    price = scrapy.Field()
    category = scrapy.Field()
    availability = scrapy.Field()
    rating = scrapy.Field()
```

3.6.4.4 Paso 4: Implementar el Spider

Editar el archivo `spiders/books.py` con la lógica de extracción:

```
# books_scraper/spiders/books.py
import scrapy
from books_scraper.items import BookItem

class BooksSpider(scrapy.Spider):
    name = "books"
```

```

allowed_domains = ["books.toscrape.com"]
start_urls = [
    "https://books.toscrape.com/catalogue/category/books/horror_31/index.
html"
]

def parse(self, response):
    """Extrae información de libros de la página actual"""

    # Extraer todos los libros de la página
    books = response.xpath("//article[contains(@class, 'product_pod')]")

    for book in books:
        item = BookItem()

        # Extraer título
        item["title"] = book.xpath("./h3/a/@title").get().strip()

        # Extraer precio
        price_text = (
            book.xpath("./p[contains(@class, 'price_color')]/
text()).get().strip()
        )
        item["price"] = price_text.replace("£", "") if price_text else
None

        # Extraer disponibilidad
        availability_xpath = (
            "./p[contains(@class, 'instock') and
contains(@class, 'availability')]"
            "/text()"
        )
        availability = book.xpath(availability_xpath).getall()
        item["availability"] = (
            "".join(availability).strip() if availability else None
        )

        # Extraer calificación
        rating_class = book.xpath(
            "./p[contains(@class, 'star-rating')]/@class"
        ).get()
        if rating_class:
            rating = rating_class.split()[-1]
            item["rating"] = rating
        else:
            item["rating"] = None

        item["category"] = "Horror"

        yield item

    # Seguir a la siguiente página si existe
    next_page = response.xpath("//li[contains(@class, 'next')]/a/
@href").get()
    if next_page:
        next_page_url = response.urljoin(next_page)
        yield scrapy.Request(next_page_url, callback=self.parse)

```

3.6.4.5 Paso 5: Configurar Pipeline para CSV

Crear un pipeline personalizado para exportar a CSV. Editar pipelines.py:

```

# books_scraper/pipelines.py
import csv
import os

```

```

class CsvExportPipeline:
    def __init__(self):
        self.file = None
        self.writer = None

    def open_spider(self, spider):
        """Se ejecuta cuando se abre el spider"""
        self.file = open("horror_books.csv", "w", newline="",
encoding="utf-8")
        self.writer = csv.DictWriter(
            self.file,
            fieldnames=["title", "price", "category", "availability",
"rating"],
        )
        self.writer.writeheader()

    def close_spider(self, spider):
        """Se ejecuta cuando se cierra el spider"""
        if self.file:
            self.file.close()

    def process_item(self, item, spider):
        """Procesa cada item extraído"""
        self.writer.writerow(dict(item))
        return item

```

3.6.4.6 Paso 6: Configurar Settings

Editar settings.py para activar el pipeline y configurar el comportamiento del spider:

```

# books_scraper/settings.py
BOT_NAME = "books_scraper"

SPIDER_MODULES = ["books_scraper.spiders"]
NEWSPIDER_MODULE = "books_scraper.spiders"

# Respetar robots.txt
ROBOTSTXT_OBEY = True

# Configurar pipelines
ITEM_PIPELINES = {
    "books_scraper.pipelines.CsvExportPipeline": 300,
}

# Configurar delays para ser respetuosos con el servidor
DOWNLOAD_DELAY = 1 # Esperar 1 segundo entre requests
RANDOMIZE_DOWNLOAD_DELAY = 0.5 # Variar el delay ±50%

# User agent personalizado
USER_AGENT = "books_scraper (untref.edu.ar)"

# Configuración de logging
LOG_LEVEL = "INFO"

```

3.6.4.7 Paso 7: Ejecutar el Spider

Para ejecutar el spider y generar el archivo CSV:

```

cd books_scraper
scrapy crawl books
# Esto generará un archivo 'horror_books.csv' con los resultados

```

3.6.4.8 Paso 8: Análisis de Resultados (Opcional)

Podemos analizar los resultados usando pandas:

```
import pandas as pd

# Cargar los datos
df = pd.read_csv(csv_path)

# Estadísticas básicas
print("Estadísticas de los libros de Horror:")
print(f"Total de libros: {len(df)}")
print(f"Precio promedio: {df['price'].astype(float).mean():.2f}")
print(f"Precio mínimo: {df['price'].astype(float).min():.2f}")
print(f"Precio máximo: {df['price'].astype(float).max():.2f}")

# Distribución de calificaciones
print("\nDistribución de calificaciones:")
print(df['rating'].value_counts())
```

Descargar código completo del Spider

3.6.4.9 Extensiones Posibles

- **Múltiples categorías:** Modificar `start_urls` para incluir más categorías
- **Imágenes:** Agregar extracción de URLs de imágenes de libros
- **Detalles adicionales:** Visitar páginas individuales de libros para más información
- **Base de datos:** Cambiar el pipeline para guardar en SQLite o PostgreSQL
- **Monitoreo:** Agregar logging y métricas de rendimiento

3.6.5 Comparación: APIs vs Web Scraping

Aspecto	APIs	Web Scraping
Acceso a datos	Estructurado y oficial	No estructurado, extraído del HTML
Estabilidad	Alta (con versionado)	Baja (cambios en el HTML rompen el código)
Legalidad	Generalmente legal con términos claros	Zona gris, depende del sitio
Límites de velocidad	Explícitos y documentados	Implícitos, basados en el comportamiento del servidor
Facilidad de uso	Diseñado para ser consumido	Requiere ingeniería inversa del HTML
Cobertura de datos	Solo lo que la API expone	Potencialmente todo lo visible en el sitio
Mantenimiento	Bajo (cambios notificados)	Alto (cambios no notificados)

3.6.6 Mejores Prácticas para Web Scraping

1. **Verificar legalidad:** Revisar términos de servicio y robots.txt
2. **Identificarse:** Usar un User-Agent descriptivo
3. **Ser respetuoso:** Limitar la frecuencia de solicitudes
4. **Manejar errores:** Anticipar cambios en la estructura del sitio
5. **Considerar alternativas:** Preferir APIs cuando estén disponibles
6. **Mantener el código:** Si los sitios cambian, el scraper debe actualizarse

3.6.7 Herramientas y Bibliotecas Adicionales

3.6.7.1 Parseo y Análisis

- **lxml:** Parser XML/HTML muy rápido
- **html5lib:** Parser que simula el comportamiento de navegadores
- **parsel:** Librería de extracción usada por Scrapy

3.6.7.2 Automatización de Navegadores

- **Selenium:** Control de navegadores web para automatización
- **Playwright:** Alternativa moderna a Selenium
- **Puppeteer:** Control de Chrome/Chromium (Node.js)

3.6.7.3 Gestión de Solicitudes

- **httpx:** Cliente HTTP asíncrono moderno
- **aiohttp:** Cliente HTTP asíncrono
- **requests-html:** Requests con soporte para JavaScript

3.6.7.4 Almacenamiento y Procesamiento

- **pandas:** Análisis y manipulación de datos
- **SQLAlchemy:** ORM para bases de datos
- **MongoDB:** Base de datos NoSQL para datos no estructurados

3.6.8 Referencias y Recursos Adicionales

3.6.8.1 Documentación Oficial

- Beautiful Soup Documentation
- Scrapy Documentation

3.6.8.2 Sitios para Practicar Web Scraping

- Quotes to Scrape - Sitio diseñado para practicar scraping
- Books to Scrape - Tienda de libros ficticia para scraping
- Scrape This Site - Ejercicios de scraping

3.6.8.3 Aspectos Legales

- Can I scrape your website?
- Understanding robots.txt

3.6.8.4 Libros y Referencias Académicas

- En el capítulo 20: Web crawling and indexes del libro (Manning et al., 2008) se explican los conceptos básicos de web scraping.
- En el libro (Mitchell, 2024) se profundiza en el tema de web scraping con Python. Este libro se puede leer online en formato html, por un tiempo limitado.

4. Recuperación de la Información

4.1 Introducción a la recuperación de la información

La **recuperación de la información** (*Information Retrieval*, IR) es una disciplina fundamental de las ciencias de la computación que estudia la representación, almacenamiento, organización y acceso a elementos de información. Su objetivo principal es desarrollar sistemas que permitan a los usuarios encontrar documentos o fragmentos de información que satisfagan una necesidad informativa específica dentro de grandes colecciones de datos.

4.1.1 Historia y Evolución

La recuperación de la información como disciplina formal emergió en la década de 1950, impulsada por el crecimiento exponencial de la información científica después de la Segunda Guerra Mundial. Pioneros como Calvin Mooers (quien acuñó el término “Information Retrieval” en 1951) y Gerard Salton (creador del sistema SMART) establecieron las bases teóricas y prácticas de esta área.

Los hitos principales incluyen:

1945 Vannevar Bush propone el sistema “Memex” como precursor conceptual

1950s-1960s Desarrollo de los primeros sistemas automáticos de indexación

1970s Introducción del modelo vectorial y medidas de evaluación estándar

1990s Explosión de Internet y la World Wide Web

2000s Algoritmos de ranking como PageRank revolucionan la búsqueda web

2010s-presente Integración de inteligencia artificial y aprendizaje automático

4.1.2 Tipos de Información

La información puede presentarse en diferentes formas, cada una con características y desafíos específicos para su recuperación:

Información estructurada Datos organizados en un formato predefinido, como bases de datos relacionales, donde los datos se almacenan en tablas con filas y columnas claramente definidas. La consulta y recuperación en este contexto suele realizarse mediante lenguajes como SQL. Los registros y campos que vimos anteriormente son ejemplos de información estructurada.

Características:

- Esquema fijo y bien definido
- Relaciones explícitas entre datos
- Consultas precisas y eficientes
- Ejemplo: bases de datos relacionales, hojas de cálculo

Información semi-estructurada Datos que no siguen un esquema rígido, pero contienen etiquetas o marcadores que facilitan su organización y búsqueda. Ejemplos comunes incluyen documentos HTML o XML, donde la estructura es flexible pero existen elementos identificables.

Características:

- Estructura flexible con etiquetas o metadatos
- Jerarquía y anidamiento de elementos
- Esquema implícito o auto-descriptivo
- Ejemplo: documentos XML, JSON, HTML, correos electrónicos

Información no estructurada Consiste principalmente en texto libre, como artículos, correos electrónicos o páginas web, donde la organización interna es mínima o inexistente. La recuperación en este caso requiere técnicas especializadas de procesamiento de lenguaje natural y modelado de relevancia.

Características:

- Ausencia de esquema predefinido
- Contenido principalmente textual
- Requiere análisis semántico y sintáctico
- Ejemplo: documentos de texto, artículos de noticias, libros digitales

4.1.3 Componentes de un Sistema de Recuperación de Información

Un sistema de IR típico consta de varios componentes interconectados. Primero se procesan los documentos para crear un índice que facilite la búsqueda. Luego, cuando un usuario realiza una consulta, el sistema la procesa y utiliza el índice para encontrar y clasificar los documentos relevantes. El feedback del usuario ayuda a mejorar futuros resultados.

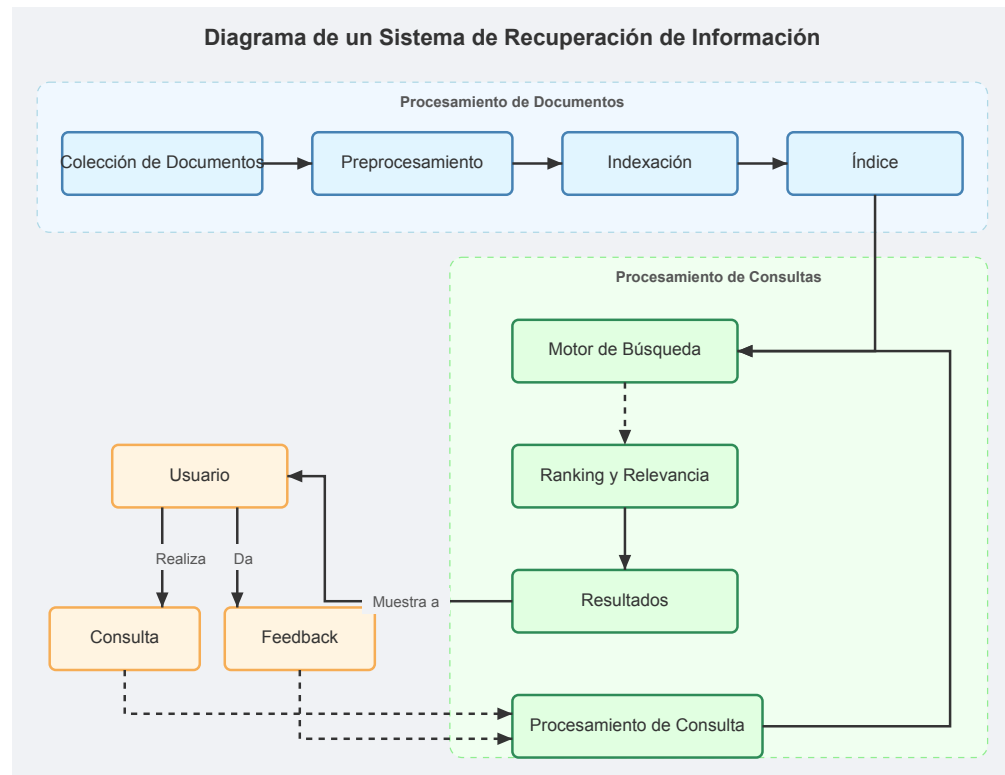


Figure 4.1: Componentes de un sistema de recuperación de información

Colección de Documentos Es el conjunto de información sobre el que opera un sistema de recuperación de información. Esta colección, también llamada *corpus*, puede estar formada por documentos de texto, páginas web, bases de datos, archivos multimedia o cualquier otro tipo de contenido digital. Su función principal es servir como fuente de datos a partir de la cual los usuarios podrán buscar y recuperar información relevante según sus necesidades, siendo fundamental que esté bien organizada y representada para facilitar el acceso eficiente a los datos.

Preprocesamiento El preprocesamiento es una etapa fundamental en los sistemas de recuperación de información, ya que transforma los documentos originales en una forma que facilita su análisis y búsqueda. Este proceso suele incluir la segmentación del texto en palabras o frases, la conversión de todos los caracteres a minúsculas y la eliminación de signos de puntuación y palabras muy comunes que no aportan significado relevante. Además, se pueden aplicar técnicas para reducir las palabras a su raíz o forma base, lo que ayuda a unificar variantes y mejorar la coincidencia entre consultas y documentos. El objetivo principal es obtener una representación más uniforme y manejable del contenido, optimizando así la eficiencia y precisión del sistema.

Indexación La indexación es el proceso mediante el cual se crean estructuras de datos que permiten localizar rápidamente la información relevante dentro de una colección de documentos. Consiste en analizar los documentos para extraer los términos más significativos y construir un índice que asocie cada término con los documentos en los que aparece. Este índice facilita la búsqueda eficiente, ya que evita la necesidad de

examinar todos los documentos cada vez que se realiza una consulta. Además, la indexación puede incorporar información adicional, como la frecuencia de los términos o relaciones semánticas, lo que mejora la precisión y relevancia de los resultados recuperados.

Procesamiento de Consultas El procesamiento de consultas es la etapa en la que el sistema interpreta la necesidad informativa expresada por el usuario y la transforma en una forma adecuada para la búsqueda. Este proceso implica analizar la consulta para comprender su significado, identificar posibles errores o ambigüedades y, en ocasiones, enriquecerla mediante la expansión de términos o la corrección ortográfica. El objetivo es maximizar la probabilidad de recuperar información relevante, ajustando la consulta para que refleje de manera precisa la intención del usuario y se adapte a las características del sistema de recuperación.

Motor de Búsqueda El motor de búsqueda es el componente central encargado de analizar la consulta del usuario y comparar su contenido con los documentos indexados para determinar cuáles son los más relevantes. Utiliza diferentes modelos matemáticos y algoritmos para calcular la similitud o la probabilidad de relevancia entre la consulta y los documentos, considerando factores como la presencia de términos clave, la frecuencia de aparición y la importancia relativa de cada palabra. El objetivo principal del motor de búsqueda es ofrecer resultados precisos y útiles, priorizando aquellos documentos que mejor satisfacen la necesidad informativa del usuario.

Ranking y Relevancia El ranking y la relevancia constituyen el proceso mediante el cual un sistema de recuperación de información determina el orden en que se presentan los resultados al usuario, priorizando aquellos que mejor satisfacen su necesidad informativa. Este proceso se basa en la estimación de la pertinencia de cada documento respecto a la consulta, utilizando modelos matemáticos y algoritmos que consideran factores como la similitud entre los términos de la consulta y los documentos, la autoridad de las fuentes, la calidad del contenido y su actualidad. El objetivo es que los resultados más útiles y significativos aparezcan en las primeras posiciones, facilitando así una experiencia de búsqueda eficiente y satisfactoria.

4.1.4 Aplicaciones Modernas

Los sistemas de recuperación de información están presentes en numerosas aplicaciones cotidianas

Motores de Búsqueda Web Los más conocidos como Google, Bing o DuckDuckGo que indexan billones de páginas web y proporcionan resultados relevantes en fracciones de segundo.

Sistemas de Recomendación Plataformas como Netflix, Amazon o Spotify que sugieren contenido basándose en preferencias y comportamientos de usuarios.

Bibliotecas Digitales Repositorios como PubMed, IEEE Xplore o arXiv que organizan literatura científica y académica.

Motores de Búsqueda Empresariales Herramientas internas que permiten buscar en documentos corporativos, bases de conocimiento y sistemas de gestión.

4.1.5 Tendencias Futuras

En la actualidad, la inteligencia artificial y el aprendizaje automático están transformando la recuperación de la información. Por ejemplo, *deep learning* utiliza redes neuronales para comprender mejor el significado de los textos, permitiendo que los sistemas sean más precisos al buscar información relevante. Modelos avanzados como BERT y los llamados transformers ayudan a los sistemas a entender el contexto de las palabras en una oración, mejorando la calidad de las respuestas. Además, existen sistemas de recuperación de información basados completamente en redes neuronales, que pueden aprender de grandes cantidades de datos y adaptarse a diferentes tipos de consultas de manera automática.

La búsqueda multimodal es otra tendencia importante. Esto significa que los sistemas ya no solo buscan información en texto, sino que también pueden analizar imágenes, audio y videos. Por ejemplo, ahora es posible buscar una imagen similar a otra, o encontrar

información relevante en un video. Incluso, con la realidad aumentada, se pueden obtener datos útiles sobre objetos o lugares en tiempo real usando la cámara de un dispositivo.

La personalización avanzada permite que los sistemas adapten los resultados de búsqueda a cada usuario. Esto se logra creando perfiles dinámicos que consideran el historial y las preferencias de cada persona. Así, los resultados pueden ser más útiles y relevantes. Sin embargo, también es importante encontrar un equilibrio entre la personalización y la privacidad de los usuarios.

Finalmente, la búsqueda conversacional está ganando popularidad. Los asistentes virtuales y chatbots permiten interactuar con los sistemas mediante lenguaje natural, ya sea escribiendo o hablando. Esto facilita que los usuarios puedan refinar sus consultas a través de un diálogo, haciendo la búsqueda más intuitiva y accesible para todos.

La recuperación de información seguirá siendo un campo fundamental conforme la cantidad de datos digitales continúe creciendo exponencialmente, requiriendo sistemas cada vez más sofisticados y eficientes para ayudar a los usuarios a encontrar la información que necesitan.

4.2 Índices Invertidos

Los **índices invertidos** son la estructura de datos fundamental en los sistemas de recuperación de información modernos. Son utilizados por motores de búsqueda, sistemas de búsqueda en documentos, y cualquier aplicación que necesite encontrar documentos que contengan ciertos términos de manera eficiente.

La idea central es simple pero poderosa: en lugar de ir de documento a documento buscando términos (búsqueda secuencial), creamos una estructura que va de término a documentos. Es decir, para cada término del vocabulario, mantenemos una lista de los documentos donde aparece.

Nos podemos imaginar que un índice invertido es una especie de diccionario, donde las claves son las palabras (términos) y los valores son listas de id de documentos (postings) que contienen esas palabras. Donde previamente, a cada documento que forma parte de la colección se le asignó un identificador único (doc_id).

4.2.1 Motivación

Imaginemos que tenemos una colección de documentos y queremos buscar aquellos que contienen la palabra "Python". Sin un índice, tendríamos que:

1. Leer cada documento completo
2. Buscar la palabra "Python" en cada uno
3. Guardar los documentos que la contengan

Este proceso es extremadamente ineficiente para colecciones grandes. Con un índice invertido, simplemente buscamos "python" en el diccionario y obtenemos directamente la lista de documentos que lo contienen.

```
# Ejemplo: búsqueda sin índice (ineficiente)
documentos = {
    1: "Python es un lenguaje de programación",
    2: "Java es un lenguaje orientado a objetos",
    3: "Python y Java son lenguajes populares",
    4: "Machine learning con Python",
}

def buscar_sin_indice(termino, documentos):
    """Busca documentos que contienen un término (búsqueda secuencial)"""
    resultado = []
    for doc_id, contenido in documentos.items():
        if termino.lower() in contenido.lower():
            resultado.append(doc_id)
    return resultado

# Buscar documentos con "Python"
docs_con_python = buscar_sin_indice("Python", documentos)
print(f"Documentos con 'Python': {docs_con_python}")
```

Como se puede ver, este método requiere examinar cada documento completo. Para colecciones con millones de documentos, si cada vez que realizamos una búsqueda hay que leer todos los documentos el método resulta impracticable.

La idea detrás de los índices invertidos es leer una sola vez todos los documentos para construir el índice, y luego usar ese índice para responder consultas de manera eficiente.

4.2.2 Recuperación Booleana

La **recuperación booleana** es el modelo más simple de recuperación de información. En este modelo, las consultas se formulan como expresiones booleanas con operadores AND, OR y NOT.

Por ejemplo la siguiente matriz representa la incidencia de términos en documentos, donde las filas son términos (palabras) y las columnas son las páginas de este apuntes (documentos), en cada celda indica si el término aparece (1) o no (0) en el documento correspondiente:

	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
XML	1	0	0	1	0	1	1	0	1	0	0	0	0	0	0
regex	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
haskell	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
invertido	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0
java	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0

Ejemplos de consultas booleanas:

- XML AND java: documentos que contienen ambos términos

```
XML:      100101101000000
              AND
java:     000010001000000
-----
000000001000000 → documento 23
```

- regex OR invertido: documentos que contienen al menos uno de los términos

```
regex:     000010001000000
              OR
invertido: 000010011000000
-----
000010011000000 → documentos 19, 23, 24
```

- XML AND NOT Java: documentos que contienen “XML” pero no “Java”

```
XML:      100101101000000
              AND
NOT java:  111101110111111
-----
1001011000000000 → documentos 15, 18, 20, 21
```

El modelo booleano es determinístico: un documento o bien coincide con la consulta o no. No hay noción de “cuán bien” coincide un documento.

4.2.3 Estructura del Índice Invertido

Un índice invertido consta de dos componentes principales:

1. **Diccionario (o vocabulario):** Contiene todos los términos únicos que aparecen en la colección
2. **Listas de postings:** Para cada término del diccionario, una lista de documentos donde aparece

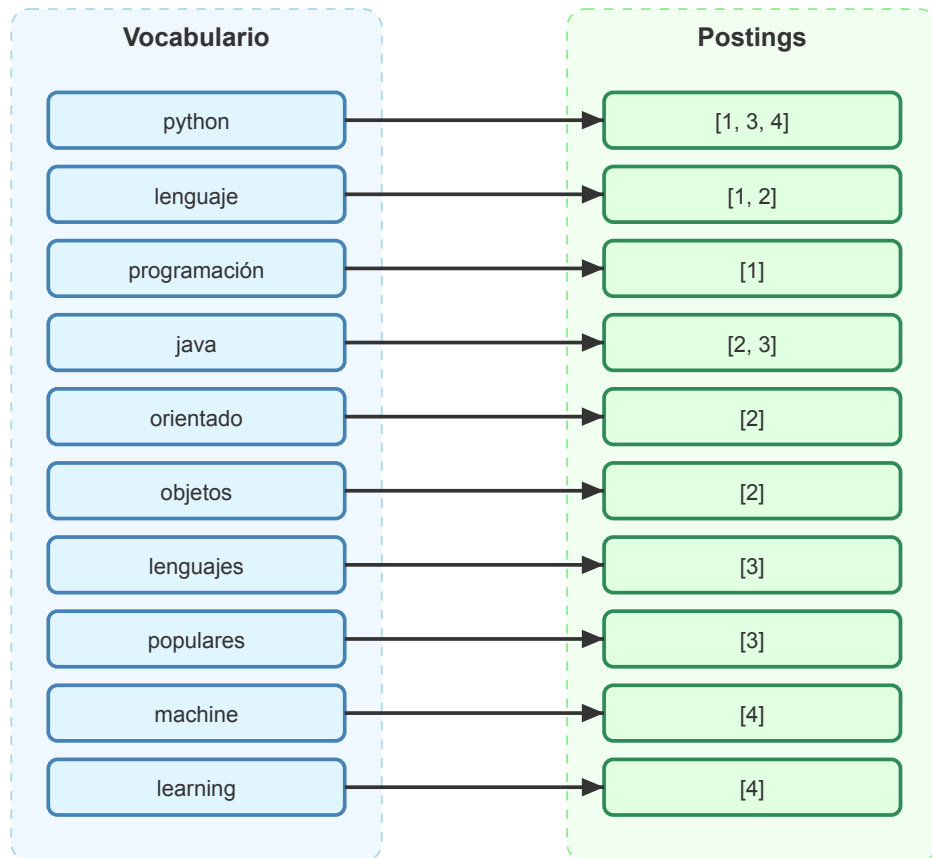


Figure 4.3: Estructura de un índice invertido

A continuación se muestra una implementación simple de un índice invertido en Python que permite agregar documentos y realizar búsquedas booleanas.

En esta primera implementación, suponemos que tanto los documentos, como el índice caben en memoria. No se indexan stopwords.

Definición de Stopwords

Las stopwords son palabras comunes que no se indexan porque aportan poco valor semántico en las búsquedas. En general no se indexan artículos, preposiciones, pronombres, adverbios comunes, algunos verbos conjugados, etc.

```
from collections import defaultdict

# Palabras que NO se deben indexar (stopwords indicadas)
STOPWORDS = {
    "es",
    "un",
    "de",
    "a",
    "son",
    "con",
    "y",
    "la",
    "el",
    "en",
    "los",
    "las",
    "por",
    "para",
}
```

```

class IndiceInvertido:
    """Implementación simple de un índice invertido"""

    def __init__(self):
        # Diccionario: término -> conjunto de IDs de documentos
        self.indice = defaultdict(set)
        # Almacena los documentos originales
        self.documentos = {}

    def agregar_documento(self, doc_id, texto):
        """Agrega un documento al índice (no indexa palabras en STOPWORDS)"""
        self.documentos[doc_id] = texto

        # Tokenizar o separar el documento como una lista de palabras
        # Las palabras se normalizan a minúsculas y se eliminan signos
        # de puntuación básicos
        palabras = texto.lower().split()

        # Agregar cada palabra al índice salvo las stopwords indicadas
        for palabra in palabras:
            # Eliminar puntuación básica alrededor de la palabra
            palabra = palabra.strip(".,;:!?()[\]{}\\'")
            if not palabra:
                continue
            if palabra in STOPWORDS:
                continue
            self.indice[palabra].add(doc_id)

    def buscar(self, termino):
        """Busca documentos que contienen el término
        (normaliza a minúsculas)"""
        termino = termino.lower().strip(".,;:!?()[\]{}\\'")
        return self.indice.get(termino, set())

    def buscar_and(self, termino1, termino2):
        """Busca documentos que contienen ambos términos"""
        docs1 = self.buscar(termino1)
        docs2 = self.buscar(termino2)
        return docs1 & docs2

    def buscar_or(self, termino1, termino2):
        """Busca documentos que contienen al menos uno de los términos"""
        docs1 = self.buscar(termino1)
        docs2 = self.buscar(termino2)
        return docs1 | docs2

    def buscar_not(self, termino1, termino2):
        """Busca documentos que contienen termino1 pero no termino2"""
        docs1 = self.buscar(termino1)
        docs2 = self.buscar(termino2)
        return docs1 - docs2

    def __repr__(self):
        """Representación del índice para inspección"""
        resultado = []
        for termino in self.indice.keys():
            docs = self.indice[termino]
            resultado.append(f"{termino}: {docs}")
        return "\n".join(resultado)

# Crear el índice
indice = IndiceInvertido()
indice.agregar_documento(1, "Python es un lenguaje de programación")
indice.agregar_documento(2, "Java es un lenguaje orientado a objetos")

```

```

indice.agregar_documento(3, "Python y Java son lenguajes populares")
indice.agregar_documento(4, "Machine learning con Python")

# Mostrar el índice
print("Índice invertido:")
print(indice)

```

Nota

defaultdict de la librería estándar de Python se utiliza para simplificar la creación del diccionario de listas de postings. Cada vez que se accede a una clave que no existe, se crea automáticamente un conjunto vacío sin necesidad de inicializarlo con setdefault como un diccionario estándar.

Para realizar búsquedas booleanas, conviene representar las listas de postings como conjuntos (set) para aprovechar las operaciones de intersección, unión y diferencia que son eficientes en conjuntos.

4.2.3.1 Búsquedas con el Índice

Ahora podemos realizar búsquedas muy eficientemente:

```

# Búsquedas simples
print(f"\nDocumentos con 'python': {indice.buscar('python')}")
print(f"Documentos con 'lenguaje': {indice.buscar('lenguaje')}")

# Búsquedas booleanas
print(f"\nPython AND Java: {indice.buscar_and('python', 'java')}")
print(f"Python OR Machine: {indice.buscar_or('python', 'machine')}")
print(f"Python AND NOT Java: {indice.buscar_not('python', 'java')}")

```

4.2.4 El Vocabulario y Procesamiento de Términos

En la práctica, el procesamiento de términos es más sofisticado que simplemente convertir a minúsculas y separar por espacios. Los sistemas reales aplican varias técnicas para mejorar la calidad del índice y la recuperación, entre ellas “normalización”, “tokenización”, “eliminación de stopwords” y “stemming/lematización”. A continuación se muestran ejemplos de cada técnica usando la librería NLTK en Python.

4.2.4.1 NLTK

NLTK (Natural Language Toolkit) es una librería popular en Python para procesamiento de lenguaje natural. Proporciona herramientas para tokenización, stemming, lematización, y manejo de stopwords, entre otras funcionalidades.

Para instalar NLTK, ejecutar en la terminal:

```
pip install nltk
```

Luego es necesario descargar algunos recursos adicionales (stopwords, modelos de tokenización) usando:

```
import nltk

nltk.download()
```

Se abrirá una ventana gráfica para seleccionar los recursos a descargar. Alternativamente, se pueden descargar recursos específicos directamente en el código como se muestra en los ejemplos a continuación.

```
import nltk

nltk.download("stopwords", quiet=True)
nltk.download("punkt_tab", quiet=True)
```

4.2.5 Procesamiento de Términos con NLTK

4.2.5.1 Normalización

El proceso de normalización consiste en convertir el texto a una forma estándar. Aquí se muestra un ejemplo básico de normalización usando NLTK para tokenizar y limpiar el texto.

```
import re
import nltk
from nltk.tokenize import word_tokenize

nltk.download("punkt_tab", quiet=True)

def normalizar_texto(texto):
    """Normaliza un texto para indexación usando NLTK para tokenizar."""
    # Convertir a minúsculas
    texto = texto.lower()
    # Reemplazar signos de puntuación por espacios
    # (dejando letras, números y espacios)
    texto = re.sub(r"[^\w\s]", " ", texto)
    # Tokenizar con NLTK (maneja mejor clíticos, contracciones, etc.)
    tokens = word_tokenize(texto, language="spanish")
    # Reconstruir string normalizado (tokens separados por espacio)
    return " ".join(tokens)

# Ejemplo
texto = "¡Python es GENIAL! ¿No lo crees? Dímelo."

print(f"Original: {texto}")
print(f"Normalizado: {normalizar_texto(texto)}")
```

4.2.5.2 Tokenización

Es el proceso de dividir el texto en unidades (tokens).

```
import nltk
from nltk.tokenize import word_tokenize

nltk.download("punkt_tab", quiet=True)

def tokenizar(texto):
    """Tokeniza un texto en palabras usando NLTK (Spanish punkt)."""
    texto_normalizado = texto.lower()
    tokens = word_tokenize(texto_normalizado, language="spanish")
    return tokens

texto = "Python 3.9 es la versión más reciente"
tokens = tokenizar(texto)
print(f"Tokens: {tokens}")
```

4.2.5.3 Eliminación de Stopwords

NLTK proporciona listas de stopwords para varios idiomas, incluyendo español.

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

nltk.download("stopwords", quiet=True)
nltk.download("punkt_tab", quiet=True)

STOPWORDS_NLTK = set(stopwords.words("spanish"))
```

```

def eliminar_stopwords(tokens):
    """Elimina stopwords usando la lista de NLTK para español."""
    return [t for t in tokens if t not in STOPWORDS_NLTK]

# Ejemplo
texto = "Python es un lenguaje de programación muy popular"
tokens = word_tokenize(texto.lower(), language="spanish")
tokens_sin_stop = eliminar_stopwords(tokens)

print(f"Tokens originales: {tokens}")
print(f"Sin stopwords: {tokens_sin_stop}")

import textwrap
import nltk
from nltk.corpus import stopwords

nltk.download("stopwords", quiet=True)

STOPWORDS_NLTK = set(stopwords.words("spanish"))

print("Stopwords en español (NLTK):")
palabras = sorted(STOPWORDS_NLTK)
texto = ", ".join(palabras)
for linea in textwrap.wrap(texto, width=80):
    print(linea)

```

4.2.5.4 Stemming y Lematización

El proceso de stemming consiste en reducir las palabras a su raíz o forma base. La lematización es un proceso más sofisticado, que para recortar las palabras utiliza el contexto.

Se muestra stemming en español con SnowballStemmer (disponible en NLTK). NLTK no ofrece un lematizador robusto en español, por lo que incluye un ejemplo de lematización en inglés con WordNet (por si hay textos en inglés).

```

import nltk
from nltk.stem.snowball import SnowballStemmer
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize

nltk.download("wordnet", quiet=True)

# Stemmer para español
stemmer_es = SnowballStemmer("spanish")

# Lemmatizer para inglés (WordNet)
lemmatizer_en = WordNetLemmatizer()

def stem_tokens_es(tokens):
    """Aplica SnowballStemmer (español) a una lista de tokens."""
    return [stemmer_es.stem(t) for t in tokens]

def lematize_tokens_en(tokens):
    """Ejemplo de lematización en inglés con WordNetLemmatizer."""
    return [lemmatizer_en.lemmatize(t) for t in tokens]

# Ejemplos
pal_es = ["programación", "programar", "programador", "estudiante", "estudiar"]
stems = stem_tokens_es(pal_es)
print("Stemming (es):")
for p, s in zip(pal_es, stems):
    print(f"{p} → {s}")

```

```

pal_en = ["runners", "run", "philosophy", "philosophical", "philosopher"]
lemmas_en = lemmatize_tokens_en(pal_en)
print("\nLematización (en, WordNet):")
for p, l in zip(pal_en, lemmas_en):
    print(f"{p} → {l}")

```

Tanto el stemming como la lematización ayudan a agrupar diferentes formas de una misma palabra, reduciendo el tamaño del vocabulario y mejorando la recuperación al costo de pérdida de la información, ya que diferentes palabras pueden mapear al mismo stem o lema.

En el ejemplo anterior, “programación”, “programar” y “programador” se reducen al mismo stem “program”. Esto puede ser beneficioso para la recuperación, pero también puede causar ambigüedad lo que conduce a resultados menos precisos. Una decisión de diseño importante en la construcción del índice es elegir entre usar stemming, lematización o ninguna de las dos técnicas, dependiendo de los requisitos específicos de la aplicación.

Una técnica común es experimentar con diferentes configuraciones y evaluar su impacto en la precisión y recall de las búsquedas.

Por ejemplo se puede utilizar un stemmer o lematizador durante la fase de construcción del índice para normalizar los términos, y luego aplicar el mismo proceso a las consultas de los usuarios para asegurar que coincidan con los términos indexados.

4.2.5.4.1 spaCy

Otra librería popular para procesamiento de lenguaje natural es **spaCy**, que ofrece modelos robustos para varios idiomas, incluyendo español. SpaCy proporciona tokenización, lematización, y reconocimiento de entidades nombradas, entre otras funcionalidades avanzadas. Para instalar spaCy y el modelo en español, ejecutar:

```

pip install spacy
python -m spacy download es_core_news_sm

```

o de forma alternativa se puede descargar el modelo desde Python con:

```

import spacy

spacy.cli.download("es_core_news_sm")

```

A continuación se muestra un ejemplo de uso de spaCy para tokenización y lematización en español.

```

import spacy
import textwrap # Para imprimir en 77 columnas

# Cargar modelo en español
nlp = spacy.load("es_core_news_sm")

# Texto de ejemplo (puede reutilizarse el texto definido anteriormente)
texto = """La recuperación de la información en Python combina técnicas de
procesamiento de texto con estructuras de datos eficientes para permitir
búsquedas rápidas y relevantes sobre colecciones de documentos. En la práctica
se sigue un pipeline que incluye: 1) extracción y normalización del texto
(minúsculas, eliminación de acentos y puntuación), 2) tokenización (dividir en
tokens o palabras), 3) eliminación de stopwords, 4) stemming o lematización
para agrupar formas de una misma palabra, y 5) construcción de una estructura
de índice invertido que asocia cada término a una lista de documentos
(postings).

```

Para implementar esto en Python existen herramientas y bibliotecas útiles: NLTK y spaCy para tokenización, stopwords y lematización; scikit-learn para transformar colecciones en matrices TF-IDF y calcular similitud coseno; y bibliotecas especializadas como Whoosh o clientes para motores externos (Elasticsearch) cuando la escala crece. Un índice invertido básico se puede representar con dicts y sets (término -> set(doc_id)) o con listas ordenadas

de postings para operaciones booleanas y de fusión eficientes.

En el modelo de recuperación ponderada, se suele representar cada documento como un vector en un espacio de términos usando TF-IDF. Con scikit-learn, TfidfVectorizer facilita la tokenización, normalización y cálculo de pesos; luego, para una consulta, se transforma la consulta al mismo espacio y se calculan similitudes (por ejemplo, coseno) para ordenar resultados por relevancia. Para colecciones grandes se deben considerar técnicas de dimensionalidad y búsqueda aproximada (ANN) para acelerar consultas.

Cuando la colección no cabe en memoria, se aplican algoritmos como BSBI y SPIMI para construir índices por bloques y luego fusionarlos; en entornos distribuidos se emplea MapReduce o motores distribuidos (Elasticsearch, Solr) que gestionan particionado, replicación y tolerancia a fallos. Además, la compresión de postings (delta encoding, gamma codes, varint) reduce drásticamente el espacio en disco y mejora la transferencia I/O.

En proyectos reales conviene mantener separación clara entre fases: extracción (parsing de documentos), normalización y tokenización (pipelines reutilizables), construcción del índice (API clara para agregar/eliminar documentos) y la capa de búsqueda (consultas booleanas y ponderadas, paginación y highlights). También es crucial añadir pruebas automáticas, métricas de evaluación (precisión, recall, MAP, NDCG) y pipelines de validación para comparar variantes de preprocesamiento y ponderación.

Finalmente, en Python es habitual prototipar con estructuras sencillas (dicts, sets) para validar ideas y luego migrar a soluciones más robustas: persistencia del índice (SQLite, LevelDB, archivos binarios), servicios de búsqueda (Elasticsearch) o bindings a Lucene para producción. Estas decisiones dependen de requisitos de latencia, volumen de datos y la necesidad de actualizaciones en tiempo real."""

```
# Procesar texto
doc = nlp(texto)

# Filtrar tokens y lemas: excluir stopwords, puntuación y tokens no
alfabéticos
tokens_sin_stop = [
    token.text for token in doc if not token.is_stop and token.is_alpha]
lemmas_sin_stop = [
    token.lemma_ for token in doc if not token.is_stop and token.is_alpha
]

label = "Tokens (sin stopwords): "
s = ", ".join(tokens_sin_stop)
print(
    textwrap.fill(s, width=77,
        initial_indent=label,
        subsequent_indent=" " * len(label))
)

label = "Lemas (sin stopwords): "
s = ", ".join(lemmas_sin_stop)
print(
    textwrap.fill(s, width=77,
        initial_indent=label,
        subsequent_indent=" " * len(label))
)
```

4.2.6 Algoritmos de Construcción de Índices

Cuando trabajamos con colecciones grandes de documentos que no caben en memoria RAM, necesitamos algoritmos especializados para construir el índice invertido. Existen tres enfoques principales: BSBI, SPIMI, y construcción distribuida con MapReduce.

4.2.6.1 BSBI (Blocked Sort-Based Indexing)

El algoritmo **BSBI** (Blocked Sort-Based Indexing) es una técnica que construye índices cuando la colección de documentos no cabe en memoria. Divide el procesamiento en dos fases:

Fase 1: Generación de bloques ordenados

1. Lee documentos en bloques que sí caben en memoria
2. Para cada bloque, extrae pares (término, doc_id)
3. Ordena los pares en memoria, agrupando por términos
4. Escribe el bloque ordenado a disco

Fase 2: Fusión de bloques (merge de k-vías)

1. Abre todos los archivos de bloques simultáneamente
2. Usa un heap para fusionar eficientemente
3. Produce el índice final ordenado

4.2.6.1.1 Pseudocódigo BSBI

```
ALGORITMO BSBI(colección_documentos)
    bloques = []
    buffer = []

    // Fase 1: Crear bloques ordenados
    PARA CADA documento EN colección_documentos:
        pares = ParsearDocumento(documento)
        buffer.agregar(pares)

        SI buffer.tamaño >= TAMAÑO_BLOQUE:
            índice_bloque = InvertirYOrdenar(buffer)
            archivo_bloque = EscribirADisco(índice_bloque)
            bloques.agregar(archivo_bloque)
            buffer.limpiar()
    FIN PARA

    // Procesar último bloque si existe
    SI buffer NO está vacío:
        índice_bloque = InvertirYOrdenar(buffer)
        archivo_bloque = EscribirADisco(índice_bloque)
        bloques.agregar(archivo_bloque)
    FIN SI

    // Fase 2: Fusionar todos los bloques
    índice_final = FusionarBloques(bloques)
    RETORNAR índice_final
FIN ALGORITMO

FUNCIÓN InvertirYOrdenar(pares_término_docid):
    // Ordena los pares por (término, doc_id)
    pares_ordenados = Ordenar(pares_término_docid)

    // Construye diccionario término -> [doc_ids]
    índice = diccionario_vacío()
    PARA CADA (término, doc_id) EN pares_ordenados:
        índice[término].agregar(doc_id)
    FIN PARA

    RETORNAR índice
```

FIN FUNCIÓN

```
FUNCIÓN FusionarBloques(lista_bloques):
    // Merge de k-vías usando un heap
    heap = heap_vacío()
    archivos = []

    // Inicializar heap con primera línea de cada bloque
    PARA CADA bloque EN lista_bloques:
        archivo = Abrir(bloque)
        archivos.agregar(archivo)
        (término, postings) = LeerLínea(archivo)
        heap.insertar((término, postings, archivo))
    FIN PARA

    índice_final = diccionario_vacío()
    término_actual = NULL
    postings_acumulados = []

    MIENTRAS heap NO vacío:
        (término, postings, archivo) = heap.extraer_mínimo()

        SI término ≠ término_actual Y término_actual ≠ NULL:
            índice_final[término_actual] = postings_acumulados
            postings_acumulados = []
        FIN SI

        término_actual = término
        postings_acumulados.agregar(postings)

        // Leer siguiente línea del mismo archivo
        SI NO archivo.fin():
            (término, postings) = LeerLínea(archivo)
            heap.insertar((término, postings, archivo))
        FIN SI
    FIN MIENTRAS

    // Guardar último término
    SI término_actual ≠ NULL:
        índice_final[término_actual] = postings_acumulados
    FIN SI

    RETORNAR índice_final
FIN FUNCIÓN
```

4.2.6.1.2 Complejidad de BSBI

- **Tiempo:** $O(T \log T)$ donde T es el número total de pares (término, doc_id)
 - Ordenamiento de bloques: $O(T \log T)$
 - Merge de k bloques: $O(T \log k)$
- **Espacio:** $O(B)$ donde B es el tamaño del bloque en memoria
- **I/O:** Cada par (término, doc_id) se lee y escribe una vez

4.2.6.1.3 Ventajas y Desventajas de BSBI

Ventajas:

- Simple de implementar
- Funciona bien con colecciones que no caben en memoria
- El ordenamiento garantiza postings ordenados
- Eficiente uso de I/O secuencial

Desventajas:

- Requiere espacio en disco para bloques intermedios
- El merge de k-vías puede ser complejo con muchos bloques
- Manejo de términos muy frecuentes puede ser ineficiente

4.2.6.2 SPIMI (Single-Pass In-Memory Indexing)

El algoritmo **SPIMI** mejora BSBI al generar directamente un diccionario de términos arrow.r postings en cada bloque, en lugar de generar y ordenar pares. Esto es más eficiente en memoria.

4.2.6.2.1 Pseudocódigo SPIMI

```
ALGORITMO SPIMI(flujo_tokens)
    bloques = []
    diccionario = diccionario_vacío()

    MIENTRAS hay_más_tokens():
        MIENTRAS hay_memoria_disponible():
            (término, doc_id) = siguiente_token()

            SI término NO EN diccionario:
                diccionario[término] = nueva_lista_postings()
                AgregarADiccionario(término)
            FIN SI

            SI doc_id NO EN diccionario[término]:
                diccionario[término].agregar(doc_id)
            FIN SI
        FIN MIENTRAS

        // Memoria llena, escribir bloque a disco
        bloque = OrdenarTérminos(diccionario)
        archivo = EscribirBloque(bloque)
        bloques.agregar(archivo)
        diccionario.limpiar()
    FIN MIENTRAS

    // Fusionar bloques
    índice_final = FusionarBloques(bloques)
    RETORNAR índice_final
FIN ALGORITMO
```

4.2.6.2.2 Ventajas y Desventajas de SPIMI

Ventajas:

- Más eficiente en memoria que BSBI
- Una sola pasada por los datos
- No requiere ordenamiento explícito de pares
- Genera postings ordenados por doc_id naturalmente

Desventajas:

- Requiere estructura de datos dinámica (diccionario)
- Puede fragmentar memoria si hay muchos términos
- Más complejo que BSBI

4.2.6.3 Construcción Distribuida con MapReduce

Para colecciones masivas (terabytes o petabytes), se usa procesamiento distribuido con el paradigma **MapReduce**. Este enfoque distribuye el trabajo entre múltiples máquinas.

4.2.6.3.1 Diagrama MapReduce para Construcción de Índices

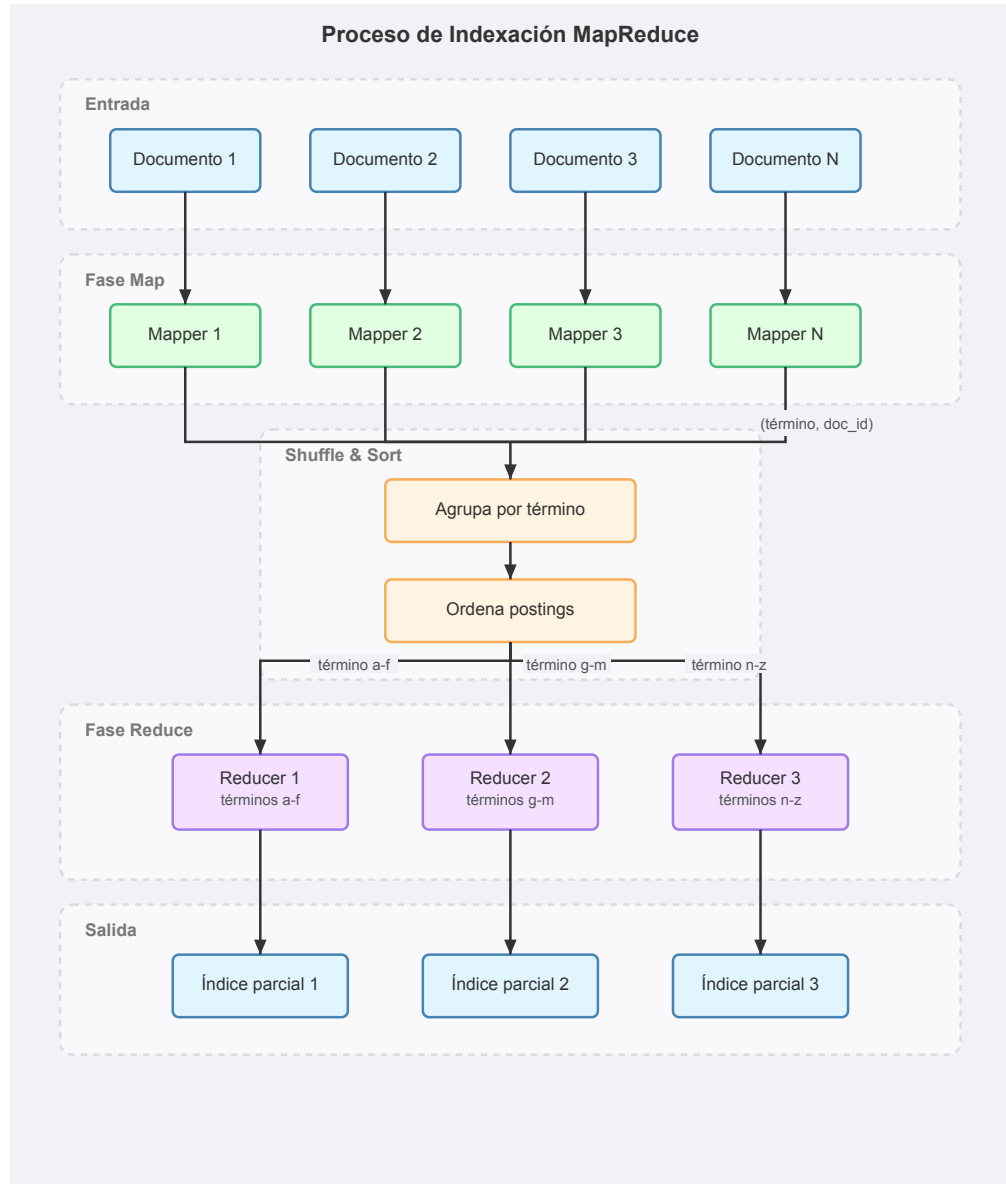


Figure 4.4: Indexado con Map-Reduce

4.2.6.3.2 Componentes de MapReduce

Fase Map:

- **Entrada:** Documento completo
- **Proceso:** Tokeniza y normaliza el texto
- **Salida:** Pares (término, doc_id) para cada término en el documento

Fase Shuffle & Sort:

- **Entrada:** Todos los pares (término, doc_id) de todos los mappers
- **Proceso:** Agrupa todos los doc_ids por término y los ordena
- **Salida:** Pares (término, lista_doc_ids) agrupados por término

Fase Reduce:

- **Entrada:** (término, lista_doc_ids) para un subconjunto de términos
- **Proceso:** Consolida y ordena la lista de doc_ids
- **Salida:** Entradas del índice invertido final

4.2.6.3.3 Pseudocódigo MapReduce

```
FUNCIÓN Map(doc_id, contenido_documento):
    términos = Tokenizar(contenido_documento)
    PARA CADA término EN términos:
        término_normalizado = Normalizar(término)
        EMITIR (término_normalizado, doc_id)
    FIN PARA
FIN FUNCIÓN

FUNCIÓN Reduce(término, lista_doc_ids):
    // Recibe: término y todos los doc_ids donde aparece
    postings = []

    PARA CADA doc_id EN lista_doc_ids:
        SI doc_id NO EN postings:
            postings.agregar(doc_id)
        FIN SI
    FIN PARA

    postings_ordenados = Ordenar(postings)
    EMITIR (término, postings_ordenados)
FIN FUNCIÓN
```

4.2.6.3.4 Proceso de Indexación Paso a Paso con MapReduce

1. **Particionamiento:** La colección se divide en splits (bloques) de documentos
2. **Map en paralelo:** Cada mapper procesa un split:
 - Lee documentos asignados
 - Tokeniza y normaliza términos
 - Emite pares (término, doc_id)
3. **Shuffle:** El framework agrupa automáticamente:
 - Todos los pares con el mismo término van al mismo reducer
 - Los doc_ids se agrupan en listas
4. **Reduce en paralelo:** Cada reducer procesa un rango de términos:
 - Recibe (término, [doc_id₁, doc_id₂, ..., doc_id_n])
 - Elimina duplicados y ordena la lista
 - Escribe el índice parcial a disco
5. **Consolidación:** Los índices parciales se combinan en el índice final

4.2.6.3.5 Ventajas y Desventajas de MapReduce

Ventajas:

- Escalabilidad masiva (miles de máquinas)
- Tolerancia a fallos automática
- Procesamiento paralelo eficiente
- Ideal para colecciones enormes (TB/PB)

Desventajas:

- Overhead de comunicación entre nodos
- Requiere infraestructura distribuida
- Más complejo de implementar y depurar
- Overkill para colecciones pequeñas

4.2.6.4 Tabla Comparativa de Algoritmos

Característica	BSBI	SPIMI	MapReduce
Tamaño de colección	Mediano (GB)	Mediano (GB)	Masivo (TB-PB)

Característica	BSBI	SPIMI	MapReduce
Requisito de memoria	Bajo (tamaño de bloque)	Medio (diccionario dinámico)	Distribuido
Complejidad implementación	Baja	Media	Alta
Velocidad (single machine)	Media	Alta	N/A
Escalabilidad	Limitada	Limitada	Excelente
Tolerancia a fallos	Manual	Manual	Automática
I/O en disco	2 pasadas (leer + escribir)	2 pasadas	Red + disco
Uso de CPU	Alto (ordenamiento)	Medio	Distribuido
Mejor caso de uso	Colecciones medianas, recursos limitados	Colecciones medianas, más memoria	Colecciones masivas, cluster disponible

4.2.6.5 Consideraciones Prácticas

Al elegir un algoritmo de construcción de índices se debe considerar:

- Tamaño de la colección:**
 - < 1 GB: Construcción en memoria simple.
 - 1-100 GB: BSBI o SPIMI.
 - > 100 GB: MapReduce o sistemas especializados.
- Recursos disponibles:**
 - RAM limitada: BSBI (menor uso de memoria).
 - RAM abundante: SPIMI (más rápido).
 - Cluster disponible: MapReduce.
- Frecuencia de actualización:**
 - Actualizaciones frecuentes: Índices incrementales.
 - Reconstrucción completa: Batch processing.
- Requisitos de tiempo:**
 - Tiempo real: Índices incrementales.
 - Batch: Cualquier algoritmo según tamaño.

4.2.6.5.1 Diseño del índice

En general se habla de términos y documentos, en la práctica hay que definir que es un documento para la aplicación específica. Por ejemplo si se quiere indexar un libro completo, ¿se indexa como un solo documento o se divide en capítulos o páginas? La granularidad afecta el tamaño del índice y la precisión de las búsquedas.

Si por el contrario se quiere indexar una colección de tweets, cada tweet puede ser un documento individual. La elección depende del caso de uso y los requisitos de recuperación.

4.2.7 Implementación con BSBI

En el siguiente enlace se encuentra una implementación en Python del algoritmo BSBI para construir un índice invertido a partir de una colección de documentos. Esta implementación incluye procesamiento básico de texto (normalización, tokenización, eliminación de stopwords y stemming) y compresión del índice.

<https://github.com/untref-edd/IndiceInvertido>

4.2.7.1 Complejidad

La construcción de un índice tiene complejidad:

- **Tiempo:** $O(n \times m)$ donde n es el número de documentos y m es el promedio de términos por documento.
- **Espacio:** $O(T)$ donde T es el número total de términos únicos en la colección.

Para colecciones muy grandes que no caben en memoria, se utilizan técnicas como:

- **Construcción por bloques:** Dividir la colección en bloques, crear índices parciales y luego fusionarlos.
- **Ordenamiento externo:** Usar algoritmos de ordenamiento que funcionen con datos en disco.
- **Procesamiento distribuido:** Utilizar frameworks como MapReduce para procesar en paralelo.

4.2.8 Resumen

Los índices invertidos son esenciales para la recuperación eficiente de información:

- Permiten búsquedas rápidas al ir de término a documentos
- Consisten en un diccionario de términos y listas de postings
- El procesamiento de texto (normalización, tokenización, stopwords, stemming) mejora la efectividad
- Algoritmos como BSBI, SPIMI y MapReduce permiten construir índices para colecciones grandes
- Se utilizan en todos los motores de búsqueda modernos
- La construcción para colecciones grandes requiere técnicas especiales

En el siguiente capítulo veremos cómo comprimir estos índices para reducir el espacio que ocupan, lo cual es crucial cuando trabajamos con colecciones de millones de documentos.

4.2.9 Aplicaciones Prácticas

Los índices invertidos se utilizan en:

- **Motores de búsqueda web:** Google, Bing, etc.
- **Búsqueda en documentos:** Elasticsearch, Solr, Lucene
- **Búsqueda en código:** GitHub Code Search
- **Bases de datos full-text:** PostgreSQL, MongoDB con índices de texto
- **Sistemas de recomendación:** Para encontrar ítems similares
- **Detección de plagio:** Comparar documentos eficientemente

4.2.10 Referencias y Recursos Adicionales

4.2.10.1 Bibliografía Principal

- Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. Capítulos 1, 2, 3 y 4. (Manning et al., 2008)
- Modern Information Retrieval: The Concepts and Technology behind Search. (Baeza-Yates & Ribeiro-Neto, 2011)
- Information Retrieval: Implementing and Evaluating Search Engines. (Büttcher et al., 2010)

4.3 Compresión de Índices

La **compresión de índices** es una técnica fundamental para reducir el espacio ocupado por índices invertidos. Cuando trabajamos con colecciones grandes de documentos (como un motor de búsqueda web que indexa miles de millones de páginas), el tamaño del índice puede ser enorme. Comprimir el índice no solo ahorra espacio en disco, sino que también puede mejorar el rendimiento al reducir las transferencias de datos entre disco y memoria.

4.3.1 Motivación

Consideremos un ejemplo real: un motor de búsqueda que indexa mil millones de páginas web. Si cada página contiene en promedio 1000 términos únicos, y cada término aparece en promedio en 10,000 páginas, el índice invertido necesitaría almacenar aproximadamente:

- **Diccionario:** 10 millones de términos \times 10 bytes = 100 MB
- **Listas de postings:** 10 millones de términos \times 10,000 documentos \times 4 bytes (un entero) = 400 GB

Esto es solo para almacenar los IDs de documentos. Agregar información adicional como frecuencias de términos o posiciones incrementa aún más el tamaño. La compresión puede reducir este espacio a una fracción del original.

```
# Ejemplo: cálculo del tamaño sin compresión
num_terminos = 10_000_000
docs_por_termino = 10_000
bytes_por_id = 4 # Entero de 32 bits

tamaño_postings_gb = \
    (num_terminos * docs_por_termino * bytes_por_id) / (1024**3)
print(f"Tamaño estimado de postings sin compresión: {tamaño_postings_gb:.2f}\
    GB")

# Con compresión típica (factor 4x)
tamaño_comprimido_gb = tamaño_postings_gb / 4
print(f"Tamaño con compresión: {tamaño_comprimido_gb:.2f} GB")
print(f"Ahorro: {tamaño_postings_gb - tamaño_comprimido_gb:.2f} GB")
```

Se puede comprimir tanto el diccionario de términos o Vocabulario como las listas de postings asociadas a cada término.

4.3.2 Compresión del Diccionario de Términos

El diccionario contiene todos los términos únicos. Aunque es relativamente pequeño comparado con las listas de postings, su compresión sigue siendo importante.

4.3.2.1 Técnicas para Comprimir el Diccionario

4.3.2.1.1 Cadena Única de Términos

La primera técnica consiste en almacenar todos los términos como si fueran una única cadena de caracteres:

- Guardar todas las palabras del diccionario como una larga cadena de caracteres.
- Se le asocia una estructura de datos de longitud fija para la frecuencia, la referencia a la lista de apariciones (postings) y la referencia al término en la cadena.
- La referencia al próximo término marca el final del término corriente.

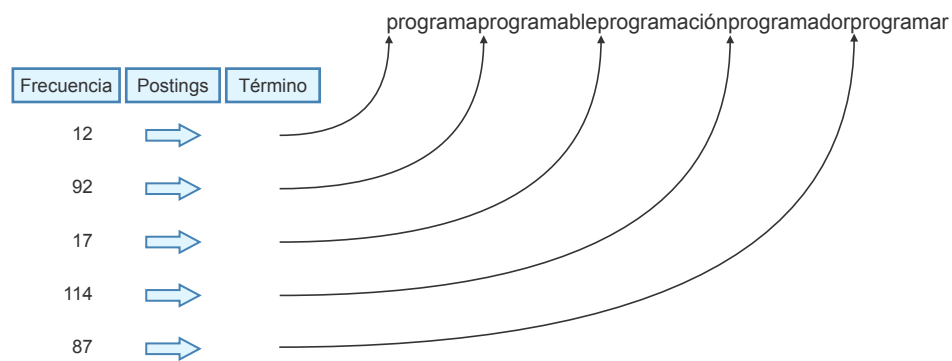


Figure 4.5: Compresión del diccionario de términos usando una cadena única.

En la figura anterior se observa como se almacenan los términos “programa”, “**programable**”, “**programación**”, “**programador**” y “**programar**” en una sola cadena. Cada término se referencia mediante un puntero que indica su posición inicial, la palabra termina justo antes del siguiente puntero.

Con este esquema si se utilizan 4 bytes para la frecuencia, 4 bytes para la referencia a la lista de postings y 4 bytes para la referencia al término, se utilizan 12 bytes por término en el diccionario. Si el diccionario tiene 1 millón de términos, se utilizan 12 MB para almacenar el diccionario más el espacio necesario para la cadena de caracteres.

En memoria se carga la cadena completa y los punteros permiten acceder a cada término. Sin embargo, aún se puede mejorar la compresión del diccionario.

4.3.2.1.2 Front Coding

Front Coding es una técnica que aprovecha los prefijos comunes entre términos consecutivos en orden lexicográfico aprovechando el hecho que, generalmente, las palabras ordenadas alfabéticamente comparten un prefijo común.

Se agrupan las entradas del diccionario en bloques de k términos contiguos. En cada bloque se almacena primero el término base completo; a continuación se coloca un separador “*” que delimita el prefijo base y marca el final de ese primer término. Para los términos restantes del bloque no se repite el prefijo: se escribe un símbolo ‘♦’ que indica el punto donde termina el prefijo común y, a continuación, el sufijo que completa cada término. Antes de cada término completo o de cada sufijo se guarda su longitud en un byte (1 B). En la estructura auxiliar solo se mantiene la referencia (puntero) al primer término de cada bloque; el resto de términos se recupera a partir de la cadena combinada.

Por ejemplo si las palabras son: algoritmo, alguacil, alguien, algas y alguno y $k=5$, se almacenan como:

5alg*as6•oritmo5•uacil4•uien3•uno

- La primera palabra “algas” se almacena completa precedida por su longitud (5). Se añade un “*” en el medio de la palabra para indicar el final del prefijo común “alg” para todo el bloque de 5 términos.
- La segunda palabra “algoritmo” se almacena como “6•oritmo”, donde “6” es la longitud del sufijo “oritmo” y “•” indica el final del prefijo común.
- La tercera palabra “alguacil” se almacena como “5•uacil”.
- La cuarta palabra “alguien” se almacena como “4•uien”.
- La quinta palabra “alguno” se almacena como “3•uno”.

Implementación de Front Coding para el bloque de ejemplo

```
def common_prefix(strings):
    """Devuelve el prefijo común más largo de una lista de strings."""
    if not strings:
```

```

        return ""
s1, s2 = min(strings), max(strings)
for i, ch in enumerate(s1):
    if i >= len(s2) or ch != s2[i]:
        return s1[:i]
return s1

def front_encode_block(terms):
    """
    Codifica un bloque de términos usando front coding.
    Formato: <len_base><prefijo>*<sufijo_base>{<len_suf>•<sufijo>}...
    """
    if not terms:
        return ""
    prefix = common_prefix(terms)
    base = terms[0]
    base_suffix = base[len(prefix) :]
    encoded = f"{len(base)}{prefix}*{base_suffix}"
    for term in terms[1:]:
        suf = term[len(prefix) :]
        encoded += f"{len(suf)}•{suf}"
    return encoded

def front_decode_block(encoded):
    """Decodifica la cadena generada por front_encode_block y
    devuelve la lista de términos."""
    import re

    if not encoded:
        return []
    m = re.match(r"(\d+)", encoded)
    if not m:
        raise ValueError(
            "Formato inválido: no se encontró la longitud del término base"
        )
    base_len = int(m.group(1))
    rest = encoded[m.end() :]
    # buscar '*' que separa prefijo y sufijo del término base
    star_idx = rest.find("*")
    if star_idx == -1:
        raise ValueError("Formato inválido: falta '*'")
    prefix = rest[:star_idx]
    # calcular sufijo del base usando la longitud indicada
    base_suffix_len = base_len - len(prefix)
    base_suffix = rest[star_idx + 1 : star_idx + 1 + base_suffix_len]
    base = prefix + base_suffix
    terms = [base]
    rem = rest[star_idx + 1 + base_suffix_len :]
    i = 0
    while i < len(rem):
        # leer número de longitud del sufijo
        j = i
        while j < len(rem) and rem[j].isdigit():
            j += 1
        if j == i:
            raise ValueError("Formato inválido al leer longitud de sufijo")
        num = int(rem[i:j])
        if j >= len(rem) or rem[j] != "•":
            raise ValueError("Formato inválido: falta '•' separador")
        suf = rem[j + 1 : j + 1 + num]
        terms.append(prefix + suf)
        i = j + 1 + num
    return terms

```

```
# Ejemplo con las palabras solicitadas
palabras = ["algas", "algoritmo", "alguacil", "alguien", "alguno"]

encoded = front_encode_block(palabras)
decoded = front_decode_block(encoded)

print("Palabras originales:", palabras)
print("Encoded:", encoded)
print("Decoded:", decoded)
assert decoded == palabras, "La decodificación no coincide con las\
palabras originales"

# Estadísticas de compresión (bytes en UTF-8)
original_bytes = sum(len(p.encode("utf-8")) for p in palabras)
comprimido_bytes = len(encoded.encode("utf-8"))
print(f"\nTamaño original: {original_bytes} bytes")
print(f"Tamaño front coding: {comprimido_bytes} bytes")
print(f"Ratio: {original_bytes / comprimido_bytes:.2f}x")
```

4.3.3 Compresión de Listas de Postings

Las listas de postings contienen los IDs de documentos donde aparece cada término. La compresión de las listas de postings es crucial ya que es la que tiene mayor impacto en el tamaño total del índice.

4.3.3.1 Gap Encoding (Codificación de Diferencias)

En lugar de almacenar los IDs completos, almacenamos las diferencias (gaps) entre IDs consecutivos. Como los IDs están ordenados, los gaps suelen ser números pequeños.

```
# Ejemplo de gap encoding
doc_ids = [15478, 15874, 17950, 50123, 50234, 60001]

print("IDs originales:")
print(doc_ids)

# Convertir a gaps
gaps = [doc_ids[0]] # Primer ID se mantiene
for i in range(1, len(doc_ids)):
    gaps.append(doc_ids[i] - doc_ids[i - 1])

print("\nGaps (diferencias):")
print(gaps)

# Comparar tamaños (asumiendo que usamos el mínimo de bits necesarios)
import math

def bits_necesarios(numero):
    """Calcula bits necesarios para representar un número"""
    if numero == 0:
        return 1
    return math.ceil(math.log2(numero + 1))

bits_originales = sum(bits_necesarios(id) for id in doc_ids)
bits_gaps = sum(bits_necesarios(gap) for gap in gaps)

print(f"\nBits necesarios:")
print(f" IDs originales: {bits_originales} bits")
print(f" Con gaps: {bits_gaps} bits")
print(f" Ahorro: {100 * (1 - bits_gaps / bits_originales):.1f}%")
```

4.3.3.2 Variable Byte Encoding (VB)

Otra técnica popular es Variable Byte encoding, que usa uno o más bytes para representar un número, dependiendo de su tamaño, así la cantidad de bytes para codificar el gap entre el id de un documento y el siguiente varía según el valor del gap.

- Cada byte tiene 7 bits de datos y 1 bit de continuación
- Bit de continuación = 1: hay más bytes
- Bit de continuación = 0: es el último byte

Así por ejemplo la representación de los siguientes gaps [15478, 396, 2076, 32173, 111, 9767] sería:

- 15478 arrow.r 10000011 10111110 00101110 (3 bytes)
- 396 arrow.r 10000010 01100100 (2 bytes)
- 2076 arrow.r 10000010 00001000 00000100 (3 bytes)
- 32173 arrow.r 10000011 11111010 00101101 (3 bytes)
- 111 arrow.r 01101111 (1 byte)
- 9767 arrow.r 10000010 00101110 00000111 (3 bytes)

Se parte de la representación en binario del número y se divide en grupos de 7 bits, cada grupo se almacena en un byte. El bit más significativo, es decir el primer bit del un byte de 8 bits, se usa para indicar si hay más bytes (1) o si es el último (0).

Por ejemplo 15478 en binario es 11110001110110. Dividido en grupos de 7 bits desde la derecha:

- 0000011_0111100_0001110

Se utiliza el bit más significativo para indicar si hay más bytes:

- 10000011_10111100_00000011

```
def vb_encode(numero):
    """Codifica un número usando Variable Byte encoding"""
    if numero == 0:
        return [0]

    bytes_list = []
    while numero > 0:
        bytes_list.insert(0, numero % 128) # 7 bits de datos
        numero //= 128

    # El último byte tiene el bit de continuación en 0
    # Los demás tienen el bit en 1 (sumamos 128)
    for i in range(len(bytes_list) - 1):
        bytes_list[i] += 128

    return bytes_list

def vb_decode(bytes_list):
    """Decodifica una lista de bytes en Variable Byte encoding"""
    numero = 0
    for byte in bytes_list:
        if byte < 128:
            # Es el último byte
            numero = numero * 128 + byte
            break
        else:
            # Hay más bytes, quitar el bit de continuación
            numero = numero * 128 + (byte - 128)
    return numero
```

Ejemplos

```

numeros = [5, 127, 128, 130, 1000, 16383]

print("Variable Byte Encoding:")
print(f"{'Número':<10} {'Bytes VB':<25} {'Bits originales':<20} {'Bits VB'}")
print("-" * 75)

for num in numeros:
    encoded = vb_encode(num)
    bits_orig = 32 # Entero de 32 bits típico
    bits_vb = len(encoded) * 8

    # Mostrar en binario
    encoded_bin = " ".join(format(b, "08b") for b in encoded)
    print(f"{'num':<10} {'encoded_bin':<25} {'bits_orig':<20} {'bits_vb'}")

# Ejemplo de compresión de una lista completa
print("\n\nCompresión de lista de postings:")
postings = [3, 12, 15, 27, 35, 89, 142, 156, 299, 312]
gaps = [postings[0]]
gaps.extend(postings[i] - postings[i - 1] for i in range(1, len(postings)))

print(f"Postings originales: {postings}")
print(f"Gaps: {gaps}")

# Codificar todos los gaps
encoded_all = []
for gap in gaps:
    encoded_all.extend(vb_encode(gap))

print(f"\nBytes VB: {encoded_all}")
print(f"Tamaño original: {len(postings) * 4} bytes (enteros de 32 bits)")
print(f"Tamaño comprimido: {len(encoded_all)} bytes")
print(f"Ratio de compresión: {len(postings) * 4 / len(encoded_all):.2f}x")

```

4.3.4 Trade-offs de la Compresión

La compresión de índices implica compromisos:

Ventajas:

- Reducción significativa del espacio en disco
- Menos transferencia de datos disco-memoria
- Posible mejora en velocidad (menos I/O)

Desventajas:

- Overhead de CPU para comprimir/descomprimir
- Código más complejo
- No se puede acceder aleatoriamente sin descomprimir

En la práctica, técnicas como Variable Byte son muy populares porque ofrecen un buen balance entre compresión y velocidad de decodificación.

4.3.5 Resumen

La compresión de índices es esencial para manejar grandes colecciones de documentos. Técnicas como front coding para el diccionario y gap encoding combinado con Variable Byte para las listas de postings permiten reducir significativamente el tamaño del índice mientras mantienen un rendimiento aceptable en las consultas.

La elección de técnicas depende de:

- Tamaño de la colección
- Patrones de consulta
- Balance CPU vs espacio
- Requisitos de velocidad

En sistemas reales como Lucene/Elasticsearch, se combinan múltiples técnicas para lograr compresión de 3-5x mientras mantienen excelente rendimiento de búsqueda.

4.3.6 Referencias y Recursos Adicionales

4.3.6.1 Colecciones de Datos

- ClueWeb Dataset: Colección grande para experimentación
- TREC Collections: Colecciones estándar para IR
- Lemur Project: Herramientas y librerías para IR

4.3.6.2 Cursos y Tutoriales

- Information Retrieval - Stanford CS276: Incluye material sobre compresión
- Text Compression - University of Melbourne: Recursos de Alistair Moffat

4.3.6.3 Bibliografía Principal

- El Capítulo 5 del libro (Manning et al., 2008) presenta los conceptos de compresión de índices.
- El siguiente artículo estudia los índices para motores de búsqueda de texto.(Zobel & Moffat, 2006)
- En el artículo se presenta la compresión de índices e imágenes (Witten et al., 1999). Está disponible en internet.

4.4 Árboles B - Índices ordenados

Con los índices invertidos hemos visto cómo organizar la información para acelerar las búsquedas por términos. Sin embargo, en muchos casos es necesario realizar búsquedas con comodines o rangos, como por ejemplo:

- Buscar todos los documentos que contengan términos que empiecen con "comput*".
- Buscar documentos con fechas entre "2020-01-01" y "2020-12-31".
- Buscar productos con precios entre 100 y 500.
- Buscar nombres de usuarios que contengan la cadena "admin".

Para estos casos, los índices invertidos no son la mejor opción, ya que están optimizados para búsquedas exactas de términos. En su lugar, se utilizan **índices ordenados** basados en estructuras de datos como los **árboles B**.

4.4.1 ¿Qué es un Árbol B?

Un árbol B es una estructura de datos autoequilibrada que mantiene los datos ordenados y permite búsquedas, inserciones y eliminaciones en tiempo logarítmico. Los árboles B son especialmente útiles para sistemas de bases de datos y sistemas de archivos debido a su capacidad para manejar grandes cantidades de datos y minimizar las operaciones de lectura/escritura en disco.

Un árbol-B de orden M (el máximo número de hijos que puede tener cada nodo) es un árbol que satisface las siguientes propiedades:

- Cada nodo tiene como máximo M hijos.
- Cada nodo (excepto la raíz) tiene como mínimo $\lceil \frac{M}{2} \rceil$ claves.
- Si la raíz no es una hoja, entonces debe tener al menos 2 hijos.
- Todos los nodos hoja aparecen al mismo nivel.
- Un nodo no hoja con k hijos contiene $k - 1$ elementos o claves almacenados.
- Los hijos de un nodo con claves (k_1, \dots, k_m) tienen que cumplir ciertas condiciones:
 - El primer hijo tiene valores menores que k_1 .
 - El segundo tiene valores mayores o igual a k_1 y menores que k_2 , etc.
 - El último hijo tiene valores mayores que k_m .

Para construir índices usaremos árboles B+, una variante de los árboles B en la que:

- Todos los valores se almacenan en las hojas.
- Los nodos internos solo almacenan claves para guiar la búsqueda.
- Las hojas están enlazadas entre sí para facilitar recorridos secuenciales.

4.4.2 Ejemplo de árbol B+

Consideremos un árbol B+ de orden 3 (cada nodo puede tener hasta 3 hijos) que almacena las siguientes palabras: "PACO", "POCO", "PECA", "PICO", "PALA", "POLO", "PIEL" y "PIPA". El árbol se vería así:

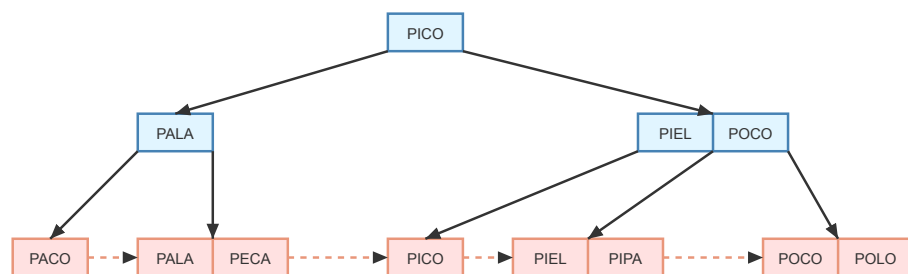


Figure 4.6: Árbol B+ de orden 3.

En la figura se observa que los nodos internos contienen las claves que guían la búsqueda, mientras que las hojas contienen las palabras completas. Además, las hojas están enlazadas entre sí para facilitar la búsqueda secuencial, lo que facilita la búsqueda con comodines o rangos.

En cada nodo pueden haber como máximo 2 claves (orden 3) y como mínimo 1 clave ($\lceil \frac{3}{2} \rceil - 1 = 1$) y las palabras dentro de cada nodo están ordenadas alfabéticamente.

En los nodos intermedios y en la raíz se repiten palabras que ya están en las hojas, de tal manera que las palabras estrictamente menores a una clave dada se encuentran en el subárbol izquierdo y las palabras mayores o iguales a esa clave se encuentran en su subárbol derecho.

En el ejemplo si se busca "PILA" como es mayor que la clave en la raíz, la búsqueda continúa en el subárbol derecho. El nodo intermedio contiene las palabras "PIEL"|"POCO", como "PILA" es mayor que "PIEL", pero menor que "POCO", la búsqueda continúa en el subárbol del medio. Al llegar a la hoja con las palabras "PIEL"|"PIPA", se determina que "PILA" no está en el árbol.

4.4.3 Inserción en un Árbol B+

La inserción de un nuevo valor en un árbol B+ sigue estos pasos:

1. **Buscar la hoja adecuada:** Se comienza en la raíz y se desciende por el árbol siguiendo las claves hasta llegar a la hoja donde debería insertarse el nuevo valor.
2. **Insertar el valor:** Si la hoja tiene espacio (menos de $M - 1$ claves), se inserta el nuevo valor en orden.
3. **Dividir la hoja si es necesario:** Si la hoja está llena (tiene $M - 1$ claves), se divide en dos hojas. La clave mediana se promueve al nodo padre.
4. **Actualizar el nodo padre:** Si el nodo padre también está lleno, se repite el proceso de división y promoción hacia arriba hasta llegar a la raíz.
5. **Crear una nueva raíz si es necesario:** Si la raíz se divide, se crea una nueva raíz con la clave promovida.

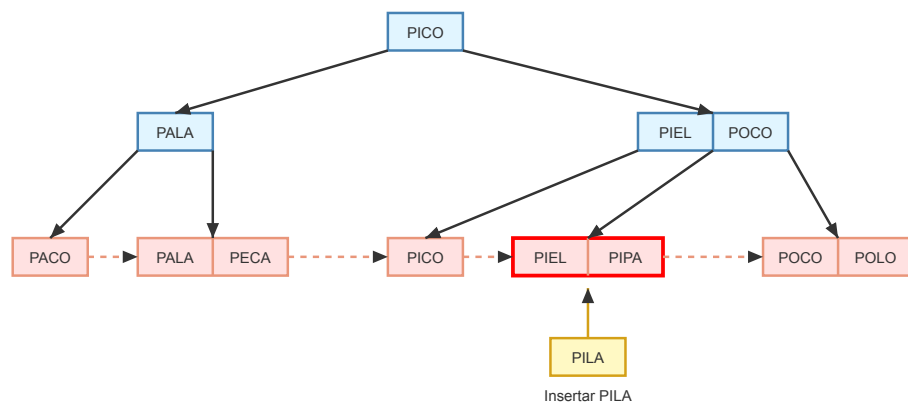


Figure 4.7: Inserción en hoja

Se encuentra la hoja donde se debe insertar "PILA".

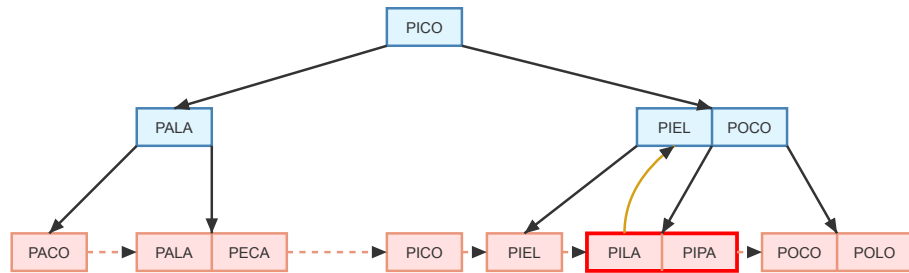


Figure 4.8: División de hoja

Al insertar "PILA", la hoja se divide en dos, la primera contiene "PIEL" y la segunda "PILA"|"PIPA".

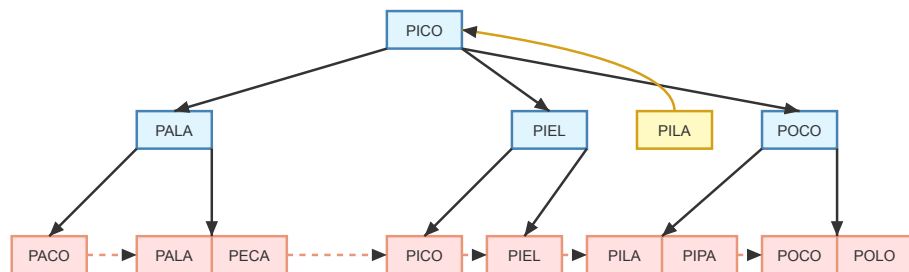


Figure 4.9: Promoción hacia arriba

Como "PILA", es la primera clave del segundo nodo que se partió, se promueve hacia arriba.

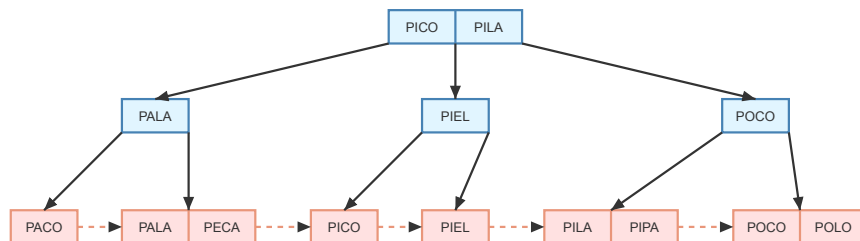


Figure 4.10: Nueva raíz

El nodo intermedio también se parte y finalmente se promueve "PILA" a la raíz.

Se puede usar un visualizador interactivo para observar cómo se realiza la inserción en un árbol B+.

4.4.4 Ejercicio interactivo

En la siguiente simulación presionar en que nodo insertar el nuevo valor y observar cómo se reestructura el árbol B cuando es necesario. Al finalizar con el botón grade se puede observar los puntos obtenidos.

4.4.5 Índices con árboles B+ en Python

Para implementar índices basados en árboles B+ en Python, utilizaremos la librería BTrees del proyecto ZODB (*Zope Object DataBase*). Esta librería proporciona implementaciones maduras y eficientes de árboles B+ que pueden persistirse en disco. BTrees ofrece varias variantes según el tipo de claves y valores:

00BTree Claves y valores como objetos Python

0IBTree Claves como objetos, valores como enteros

IOBTree Claves como enteros, valores como objetos

IIBTree Claves y valores como enteros

Para nuestros ejemplos usaremos 00BTree que permite usar *strings* tanto para claves como para valores.

A continuación se muestra un ejemplo básico de cómo crear un árbol B+, insertar valores y realizar búsquedas.

```
from BTrees.00BTree import 00BTree
import pickle

# Crear un árbol B+
# 00BTree mantiene las claves ordenadas automáticamente
btree = 00BTree()

# Insertar palabras
palabras = ["PACO", "POCO", "PECA", "PICO", "PALA", "POLO", "PIEL", "PIPA"]
for palabra in palabras:
    btree[palabra] = 1

# Persistir el árbol en disco usando pickle
with open(os.path.join(tmp_dir, "btree.pkl"), "wb") as f:
    pickle.dump(btree, f)

print("Árbol B+ creado y persistido en disco")
print(f"Total de palabras: {len(btree)}")

# Realizar búsquedas exactas
print("\nBúsquedas exactas:")
print(f"'PACO' está en el árbol: {'PACO' in btree}")
print(f"'PILA' está en el árbol: {'PILA' in btree}")
print(f"Valor de 'PACO': {btree.get('PACO')}")
print(f"Valor de 'PILA': {btree.get('PILA')}")

# Mostrar todas las palabras ordenadas
print("\nTodas las palabras (ordenadas por el árbol B+):")
for palabra in btree.keys():
    print(f"    {palabra}: {btree[palabra]}")
```

Ventajas de usar BTrees

- Implementación madura del proyecto ZODB (*Zope Object DataBase*).
- Mantiene las claves ordenadas automáticamente usando árboles B+.
- Interfaz similar a diccionarios de Python (*dict*).
- Eficiente para grandes volúmenes de datos.
- Puede persistirse fácilmente con pickle o integrarse con ZODB.

Características de los árboles B+ en BTrees

- Búsqueda: $O(\log n)$
- Inserción: $O(\log n)$
- Eliminación: $O(\log n)$
- Recorrido ordenado: $O(n)$

- Búsqueda por rango: $O(\log n + k)$ donde k es el número de resultados

4.4.5.1 Búsquedas con rangos y comodines

Los árboles B+ en BTrees permiten realizar búsquedas eficientes por rangos gracias a que mantienen los datos ordenados. Podemos usar los métodos `keys()`, `values()` e `items()` con parámetros de rango:

```
from BTrees.OOBTree import OOBTree
import pickle

# Cargar el árbol B+ desde disco
with open(os.path.join(tmp_dir, "btree.pkl"), "rb") as f:
    btree = pickle.load(f)

# Búsqueda exacta
print("Búsqueda exacta:")
print(f"'PACO' está en el árbol: {'PACO' in btree}")
print(f"'PILA' está en el árbol: {'PILA' in btree}")

# Búsqueda por rango usando keys() con min y max
print("\nBúsqueda con rango de 'PALA' a 'PICO' (inclusive):")
# keys(min, max) devuelve las claves en el rango [min, max)
# Para incluir 'PICO', usamos 'PICOZ' como límite superior
for palabra in btree.keys(min="PALA", max="PICOZ"):
    valor = btree[palabra]
    print(f" {palabra}: {valor}")

# Búsqueda con comodín (prefijo 'PI*')
# Buscamos desde 'PI' hasta 'PIZ' (siguiente prefijo)
print("\nBúsqueda con prefijo 'PI' (comodín 'PI*'):")
for palabra in btree.keys(min="PI", max="PIZ"):
    if palabra.startswith("PI"):
        valor = btree[palabra]
        print(f" {palabra}: {valor}")

# Búsqueda con comodín (prefijo 'P0*')
print("\nBúsqueda con prefijo 'P0' (comodín 'P0*'):")
for palabra in btree.keys(min="P0", max="P0Z"):
    if palabra.startswith("P0"):
        valor = btree[palabra]
        print(f" {palabra}: {valor}")
```

Eficiencia de las búsquedas por rango en árboles B+

1. **Búsquedas exactas** ("PACO" in btree): $O(\log n)$
2. **Búsquedas por rango** (keys(min="A", max="Z")): $O(\log n + k)$ donde k es el número de resultados
3. **Recorrido completo** (keys()): $O(n)$

El método `keys(min, max)` es muy eficiente porque:

- Usa el índice B+ para localizar el punto de inicio en $O(\log n)$
- Recorre secuencialmente las hojas enlazadas para obtener los k resultados
- No necesita cargar todo el árbol en memoria

Importante: El parámetro `max` es exclusivo, es decir, `keys(min="A", max="B")` devuelve claves "A" $\leq x < "B"$. Para incluir el límite superior, se puede usar un valor ligeramente mayor (como "PICOZ" para incluir "PICO").

4.4.5.2 Búsqueda con comodín "?" (un solo caracter)

Para buscar "P?C0" (donde "?" representa un solo caracter), podemos iterar sobre un rango amplio y filtrar las claves que coincidan con el patrón:

```

from BTrees.OOBTree import OOBTree
import pickle

# Cargar el árbol B+ desde disco
with open(os.path.join(tmp_dir, "btree.pkl"), "rb") as f:
    btree = pickle.load(f)

# Búsqueda con patrón "P?C0" (donde ? es un caracter cualquiera)
print("Búsqueda con patrón 'P?C0' (un caracter en la segunda posición):")
print("(Iterando sobre el rango 'P' a 'PZ' y filtrando por patrón)")

resultados = []
# Iterar sobre todas las palabras que empiezan con 'P'
for palabra in btree.keys(min="P", max="PZ"):
    # Verificar que:
    # 1. La clave tiene exactamente 4 caracteres
    # 2. Termina con "C0"
    if len(palabra) == 4 and palabra.endswith("C0"):
        valor = btree[palabra]
        resultados.append((palabra, valor))
        print(f"✓ {palabra}: {valor}")

print(f"\nTotal de resultados encontrados: {len(resultados)}")
palabras_encontradas = [r[0] for r in resultados]
print(f"Claves que coinciden con 'P?C0': {'', ' '.join(palabras_encontradas)}")

```

Estrategia para búsquedas con comodines en posiciones intermedias

1. Usar keys(min, max) para obtener un rango que incluya todas las posibles coincidencias
2. Aplicar filtros adicionales en Python (longitud, sufijo, regex, etc.)
3. El árbol B+ sigue siendo eficiente porque:
 - Solo lee los nodos necesarios del rango especificado
 - No carga todo el árbol en memoria
 - Aprovecha el orden lexicográfico de las claves

En este ejemplo, "P?C0" coincide con "PAC0", "PIC0" y "POC0", que son las únicas palabras de 4 caracteres que empiezan con "P" y terminan con "C0" en nuestro conjunto de datos.

Para patrones más complejos, se pueden usar expresiones regulares para el filtrado.

4.4.5.3 Búsqueda con comodín "*" al inicio de la palabra

Para búsquedas con comodines al inicio de la palabra (como "*C0"), los árboles B+ tradicionales no son eficientes ya que están optimizados para búsquedas que comienzan desde el inicio de la clave, es decir de prefijos.

Una estrategia efectiva es mantener un **índice adicional con palabras invertidas** en otro árbol B+. De esta manera, una búsqueda como "*C0" se transforma en una búsqueda por prefijo "0C*" en el índice invertido.

```

from BTrees.OOBTree import OOBTree
import pickle

# Cargar el árbol B+ normal
with open(os.path.join(tmp_dir, "btree.pkl"), "rb") as f:
    btree = pickle.load(f)

# Crear un árbol B+ adicional para palabras invertidas
btree_invertido = OOBTree()

# Poblar el índice con palabras invertidas
print("Creando índice con palabras invertidas:")

```

```

for palabra in btree.keys():
    palabra_invertida = palabra[::-1]
    # Almacenar la palabra invertida como clave y la original como valor
    btree_invertido[palabra_invertida] = palabra
    print(f" {palabra} → {palabra_invertida}")

# Persistir el árbol invertido
with open(os.path.join(tmp_dir, "btree_invertido.pkl"), "wb") as f:
    pickle.dump(btree_invertido, f)

print("\nÍndice con palabras invertidas creado y persistido")

# Búsqueda con patrón "*C0" (palabras que terminan en "C0")
print("\n" + "=" * 60)
print("Búsqueda con patrón '*C0' (palabras que terminan en 'C0'):")
print("=" * 60)
print("Estrategia: Buscar prefijo '0C' en el índice")
print("con palabras invertidas\n")

resultados = []
# Buscar en el índice invertido palabras que empiecen con "0C"
for palabra_invertida in btree_invertido.keys(min="0C", max="0CZ"):
    if palabra_invertida.startswith("0C"):
        # Recuperar la palabra original
        palabra_original = btree_invertido[palabra_invertida]
        resultados.append(palabra_original)
        print(f"Encontrado: '{palabra_invertida}' → '{palabra_original}'")

print(f"\nTotal de resultados: {len(resultados)}")
print(f"Palabras que terminan en 'C0': {'', '.join(sorted(resultados))}")

```

Nota

Ventajas del índice con palabras invertidas usando árboles B+:

- Convierte búsquedas por sufijo ("*C0") en búsquedas por prefijo ("0C*")
- Cada índice es un árbol B+ independiente con sus propias optimizaciones
- Ambos árboles pueden persistirse en disco de forma independiente
- Aprovecha la eficiencia de los árboles B+ para búsquedas por prefijo

Desventajas:

- Requiere espacio adicional (aproximadamente el doble de almacenamiento)
- Necesita mantener dos árboles sincronizados al insertar/eliminar
- Para comodines en posiciones intermedias ("P*C0"), se requieren técnicas más avanzadas

Alternativas para búsquedas más complejas:

- **N-gramas:** Dividir las palabras en secuencias de n caracteres e indexarlas
- **Árboles de sufijos (Suffix Trees):** Estructura especializada para búsquedas de subcadenas
- **Tries:** Para conjuntos de palabras con prefijos comunes
- **Índices invertidos con wildcards:** Como los vistos en el capítulo anterior

La elección de la estructura de datos depende del tipo de búsquedas más frecuentes en la aplicación.

4.4.5.4 Búsquedas con comodín "*" en posiciones intermedias

Para búsquedas con el comodín "*" en el medio de una palabra (como "P*C0"), se puede aprovechar tanto el árbol B+ normal como el árbol de palabras invertidas. La estrategia consiste en dividir la búsqueda en dos partes:

1. Buscar en el árbol normal las palabras que comienzan con el prefijo antes del "*" (en este caso "P").
2. Buscar en el árbol invertido las palabras que terminan con el sufijo después del "*" (en este caso "OC").
3. Intersectar los resultados de ambas búsquedas para obtener las coincidencias finales.

```

from BTrees.OOBTree import OOBTree
import pickle

# Cargar ambos árboles B+ desde disco
with open(os.path.join(tmp_dir, "btree.pkl"), "rb") as f:
    btree = pickle.load(f)

with open(os.path.join(tmp_dir, "btree_invertido.pkl"), "rb") as f:
    btree_invertido = pickle.load(f)

# Búsqueda con patrón "P*C0" (comodín en el medio)
print("Búsqueda con patrón 'P*C0' (comodín en el medio):")
print("Estrategia: Intersectar resultados de prefijo 'P' y sufijo 'OC'")
resultados_prefijo = set()

# Buscar en el árbol normal palabras que empiezan con 'P'
for palabra in btree.keys(min="P", max="PZ"):
    if palabra.startswith("P"):
        resultados_prefijo.add(palabra)

resultados_sufijo = set()

# Buscar en el árbol invertido palabras que empiezan con 'OC'
for palabra_invertida in btree_invertido.keys(min="OC", max="OCZ"):
    if palabra_invertida.startswith("OC"):
        palabra_original = btree_invertido[palabra_invertida]
        resultados_sufijo.add(palabra_original)

# Intersectar ambos conjuntos para obtener las coincidencias finales
resultados_finales = resultados_prefijo.intersection(resultados_sufijo)
print("\nResultados finales para 'P*C0':")
for palabra in resultados_finales:
    print(f"    ✓ {palabra}")

print(f"\nTotal de resultados: {len(resultados_finales)}")

```

4.4.6 Implementación completa

En ÍndiceOrdenado se encuentra una implementación completa de un índice basado en árboles B+ utilizando la librería BTrees de ZODB.

5. Aplicaciones

5.1 Recuperación de la Información de las Redes Sociales

Las redes sociales se han convertido en una de las fuentes más importantes de información en la actualidad. Plataformas como Facebook, Twitter e Instagram generan enormes volúmenes de datos cada segundo, que pueden ser analizados para obtener insights valiosos sobre comportamientos, tendencias y patrones sociales.

En este capítulo exploraremos diferentes técnicas y herramientas para recuperar y analizar información de redes sociales, centrándonos en dos casos de estudio principales:

1. **Facebook:** Modelado de redes sociales como grafos y recorrido mediante algoritmos de búsqueda en profundidad (DFS).
2. **Twitter:** Procesamiento de streams de tweets en tiempo real y análisis de datos históricos en formato JSON.

Nota

Es importante mencionar que al trabajar con datos de redes sociales, debemos respetar los términos de servicio de cada plataforma, las leyes de protección de datos personales, y considerar las implicaciones éticas del uso de información pública.

5.1.1 Caso de estudio: Facebook y Grafos de Redes Sociales

Las redes sociales pueden modelarse naturalmente como grafos, donde los nodos representan usuarios y las aristas representan relaciones de amistad o conexión. Este modelo nos permite aplicar algoritmos de teoría de grafos para analizar la estructura y propiedades de la red.

El primer paso es registrarse como desarrollador en la plataforma de Meta (Facebook) y obtener un token de acceso para usar la API Graph de Facebook. Este token es necesario para autenticar las solicitudes y acceder a los datos permitidos. Ver Anexo: Facebook para una guía detallada.

5.1.1.1 Instalación de la Librería Facebook SDK

Para trabajar con la API de Facebook en Python, vamos a usar la librería `facebook-sdk`:

```
pip install facebook-sdk
```

5.1.1.2 Ejemplo de Uso de Facebook Graph API

Copiar el siguiente fragmento de código en un archivo Python y reemplazar `USER_ACCESS_TOKEN` con el token de acceso obtenido

```
import facebook # Importamos la nueva librería

def get_all_likes_sdk(token):
    """
    Obtiene todas las páginas que le han gustado a un usuario usando el
    facebook-sdk. La paginación es manejada automáticamente por la librería.

    Args:
        token (str): El token de acceso de usuario con el permiso
        'user_likes'.

    Returns:
        list: Una lista de diccionarios que representan las páginas.
    """
    try:
        # 1. Creamos una instancia del objeto GraphAPI
        graph = facebook.GraphAPI(access_token=token)

        print("Obteniendo tus 'Me gusta' con el SDK de Facebook...")
```

```

# 2. Usamos get_all_connections para manejar la paginación
# automáticamente. La librería se encargará de hacer todas las
# llamadas necesarias.
pages_generator = graph.get_all_connections(
    id='me',
    connection_name='likes',
    fields='name,category'
)

# Convertimos el generador a una lista para tener todos los resultados
all_likes = list(pages_generator)

print("\nProceso completado.")
return all_likes

except facebook.GraphAPIError as e:
    print(f"Error de la API de Facebook: {e}")
    return []

# --- BLOQUE DE PRUEBA ---
if __name__ == "__main__":

    # --- CONFIGURACIÓN ---
    # Pegar aquí el Access Token de Usuario.
    USER_ACCESS_TOKEN = "USER_ACCESS_TOKEN"

    if "USER_ACCESS_TOKEN" in USER_ACCESS_TOKEN:
        print("Por favor, completar el TOKEN DE ACCESO DE USUARIO.")
    else:
        # Llamamos a la nueva función
        likes = get_all_likes_sdk(USER_ACCESS_TOKEN)

        if likes:
            print(f"\nSe encontraron un total de {len(likes)} páginas "
                  "que te gustan!")
            print("\n--- Ejemplo de los primeros 100 resultados: ---")
            for i, page in enumerate(likes[:100]):
                category = page.get('category', 'Sin categoría')
                print(f"{i+1}. Nombre: {page['name']} | Categoría: "
                      f"{category}")
        else:
            print("\nNo se encontraron 'Me gusta' o ocurrió un error "
                  "durante el proceso.")

```

5.1.1.3 Modelado de una Red Social como Grafo

En Facebook las relaciones de amistad son simétricas o bidireccionales, por lo tanto se pueden representar con un grafo no dirigido. A continuación, mostramos un ejemplo simple de una red social con varios usuarios y sus conexiones:

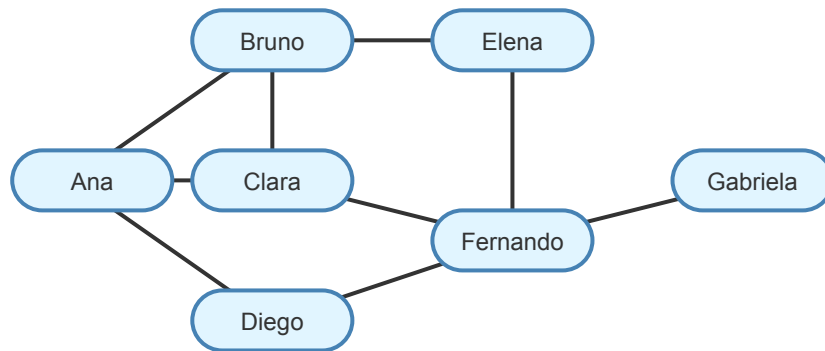


Figure 5.1: Grafo de amistades en una red social

En Facebook se puede acceder online a Graph API Explorer para probar consultas y explorar la estructura de datos: Graph API Explorer.

La siguiente consulta devuelve todos los likes e incluye las categorías:

```
GET /me/likes?fields=name,category
```

Para listar los amigos de un usuario:

```
GET /me/friends
```

Por motivos de seguridad, la API de Facebook no permite profundizar en las conexiones de amigos (amigos de amigos) sin permisos adicionales. En una aplicación no comercial sólo podemos obtener grafos egocéntricos (1 nivel de profundidad).

En una aplicación comercial se deben solicitar los permisos correspondientes y cumplir con las políticas de la plataforma para poder explorar el grafo en más profundidad.

5.1.2 Caso de estudio: Twitter/X y Procesamiento de JSON

Twitter (ahora X) es una plataforma de microblogging donde los usuarios publican mensajes cortos llamados “tweets”. La información de tweets se obtiene en formato JSON mediante la API oficial, tanto de streams en tiempo real como de datos históricos.

A diferencia de Facebook en X las relaciones no son simétricas, por lo tanto se modelan con un grafo dirigido, ya que una persona puede seguir a otra sin que la relación sea recíproca.

Una persona que sigue a otra recibe sus tweets en su timeline, pero la persona seguida no recibe automáticamente los tweets del seguidor a menos que también lo siga.

Antes de poder usar la API de Twitter, es necesario registrarse como desarrollador y crear una aplicación para obtener las credenciales necesarias (Bearer Token y Access Token). Ver Anexo: Twitter para una guía detallada.

Para consultar la API de Twitter en Python, usaremos la librería tweepy.

```
pip install tweepy
```

5.1.2.1 Ejemplo de Streaming de Tweets en Tiempo Real

Copiar el siguiente script en un archivo Python y ejecutar. Requiere un Bearer Token válido.

```
"""
```

```
Ejemplo de uso de la API de Twitter/X para análisis de datos
```

```
Requiere credenciales válidas de desarrollador de X
"""
```

```
import tweepy
import json
import os
from datetime import datetime
```

```
def verificar_credenciales():
    """Verifica que las credenciales estén configuradas"""
    bearer_token = os.getenv('TWITTER_BEARER_TOKEN')

    if not bearer_token:
        print("Error: TWITTER_BEARER_TOKEN no encontrado en variables de
entorno")
        print("\nPara configurar las credenciales:")
        print("1. Obtén un Bearer Token siguiendo el instructivo en "
              "contenidos/Anexos/Twitter.md")
        print("2. Ejecuta: export
TWITTER_BEARER_TOKEN='tu_bearer_token_aqui'")
        print("3. O crea un archivo .env con: "
              "TWITTER_BEARER_TOKEN=tu_bearer_token_aqui")
        return None

    return bearer_token
```

```
def buscar_tweets_recientes(query="python", max_results=10):
    """Busca tweets recientes sobre un tema"""

    bearer_token = verificar_credenciales()
    if not bearer_token:
        return

    try:
        # Crear cliente de Twitter API v2
        client = tweepy.Client(bearer_token=bearer_token)

        print(f"Buscando tweets sobre: '{query}'")

        # Buscar tweets recientes
        tweets = client.search_recent_tweets(
            query=f"{query} -is:retweet lang:es", # Excluir retweets, solo
español
            max_results=max_results,
            tweet_fields=["created_at", "public_metrics", "author_id",
"lang"],
            expansions=["author_id"],
            user_fields=["username", "name", "public_metrics"]
        )

        if not tweets.data:
            print("No se encontraron tweets. Verifica tu Bearer Token y cuota
"
                  "de API.")
            return

        print(f"Encontrados {len(tweets.data)} tweets")

        # Procesar y mostrar resultados
        archivo_salida = os.path.join(tmp_dir,
                                      f"tweets_{query.replace(' ',
'_' )}.jsonl")

        with open(archivo_salida, "w", encoding="utf-8") as f:
            for tweet in tweets.data:
                # Obtener información del autor
```

```

        author = None
        if tweets.includes and 'users' in tweets.includes:
            author = next(
                (user for user in tweets.includes["users"]
                 if user.id == tweet.author_id),
                None
            )

        # Mostrar información del tweet
        print(f"\nTweet: {tweet.text[:100]}...")
        if author:
            print(f"Autor: @{author.username} ({author.name})")
            print(f"Likes: {tweet.public_metrics['like_count']} | "
                  f"Retweets: {tweet.public_metrics['retweet_count']}")

        # Guardar en formato JSON
        tweet_dict = {
            "id": tweet.id,
            "created_at": str(tweet.created_at),
            "text": tweet.text,
            "author": {
                "username": author.username if author else "unknown",
                "name": author.name if author else "unknown",
                "followers": (author.public_metrics["followers_count"]
                             if author else 0)
            } if author else None,
            "metrics": tweet.public_metrics,
            "lang": tweet.lang,
            "fecha_extraccion": datetime.now().isoformat()
        }

        f.write(json.dumps(tweet_dict, ensure_ascii=False) + "\n")

    print(f"\n Tweets guardados en: {archivo_salida}")

except tweepy.errors.Forbidden as e:
    print("Error 403 - Acceso denegado:")
    print("    • Verifica que tu Bearer Token sea válido")
    print("    • Asegúrate de que tu aplicación esté asociada a un Proyecto")
    print("    • Revisa que tengas permisos para usar la API v2")
    print(f"    • Detalles del error: {e}")

except tweepy.errors.TooManyRequests as e:
    print("Error 429 - Demasiadas solicitudes:")
    print("    • Has excedido el límite de la API")
    print("    • Espera unos minutos antes de intentar de nuevo")
    print(f"    • Detalles del error: {e}")

except Exception as e:
    print(f"Error inesperado: {e}")
    print("    • Verifica tu conexión a internet")
    print("    • Asegúrate de que tweepy esté instalado: pip install tweepy")

def mostrar_estadisticas_api():
    """Muestra información sobre los límites de la API"""

    bearer_token = verificar_credenciales()
    if not bearer_token:
        return

    try:
        client = tweepy.Client(bearer_token=bearer_token)

        # Verificar autenticación haciendo una búsqueda mínima

```

```

        response = client.search_recent_tweets(
            query="test",
            max_results=10,
            tweet_fields=["created_at"]
        )

        print("Autenticación exitosa")
        print(f"• Resultados de búsqueda de prueba: "
              f"{response.meta.get('result_count', 0)} tweets encontrados")
        print("\nInformación de la API:")
        print("• API Version: v2")
        print("• Search Recent: Disponible")
        print("• Para información actualizada sobre límites y tiers de la
API,"
              " consulta la documentación oficial:")
        print("  https://developer.twitter.com/en/docs/twitter-api/rate-
limits")

    except Exception as e:
        print(f"Error al verificar API: {e}")

def main():
    """Función principal del programa"""

    print("Twitter/X API - Ejemplo de Análisis de Datos")
    print("=" * 50)

    # Verificar configuración
    print("\n1 Verificando configuración...")
    mostrar_estadisticas_api()

    # Buscar tweets sobre Python
    print("\n2 Buscando tweets sobre Python...")
    buscar_tweets_recientes("python", max_results=5)

    # Buscar tweets sobre programación
    print("\n3 Buscando tweets sobre programación...")
    buscar_tweets_recientes("programacion", max_results=5)

if __name__ == "__main__":
    # Intentar cargar variables de entorno desde .env si existe
    try:
        from dotenv import load_dotenv
        load_dotenv()
        print("Variables de entorno cargadas desde .env")
    except ImportError:
        print("Tip: Instala python-dotenv para usar archivos .env")
        print("  pip install python-dotenv")

    main()

```

5.1.2.2 Estructura de un Tweet en JSON (API v2)

La API v2 de Twitter devuelve tweets en formato JSON. Para más información sobre los campos que se pueden consultar la documentación oficial:

```

import json
from datetime import datetime

# Ejemplo de tweet en formato JSON (estructura real de API v2)
# Basado en la documentación oficial:
tweet_ejemplo_api_v2 = {
    "data": {
        "id": "1234567890123456789",
        "text": ("Las estructuras de datos son fundamentales en programación!")
    },
    "includes": {}
}

```

```

        "#EDD #Python"),
    "created_at": "2024-01-15T10:30:00.000Z",
    "author_id": "987654321",
    "lang": "es",
    "public_metrics": {
        "retweet_count": 15,
        "reply_count": 3,
        "like_count": 47,
        "quote_count": 2,
        "impression_count": 1523
    },
    "entities": {
        "hashtags": [
            {"start": 55, "end": 59, "tag": "EDD"},
            {"start": 60, "end": 67, "tag": "Python"}
        ]
    }
},
"includes": {
    "users": [
        {
            "id": "987654321",
            "name": "Maria Lopez",
            "username": "maria_dev",
            "created_at": "2020-05-10T12:00:00.000Z",
            "public_metrics": {
                "followers_count": 1523,
                "following_count": 342,
                "tweet_count": 2341,
                "listed_count": 12
            }
        }
    ]
}
}

print("=== Estructura de un Tweet (Twitter API v2) ===")
print(json.dumps(tweet_ejemplo_api_v2, indent=2, ensure_ascii=False))
print("\nDocumentación: "
      "https://developer.twitter.com/en/docs/twitter-api/data-dictionary/"
      "object-model/tweet"")

```

5.1.2.3 Procesamiento de Tweets Históricos

Cuando trabajamos con datos históricos de Twitter, comúnmente recibimos archivos en formato JSONL (JSON Lines), donde cada línea es un objeto JSON independiente. Esto facilita el procesamiento de grandes volúmenes de datos.

Nota

Datos de práctica: Los siguientes ejemplos usan datos simulados con la estructura real de la API de Twitter v2. Para trabajar con datos reales, sigue los pasos de registro y autenticación descritos anteriormente y usa los ejemplos de código con Tweepy.

```

import random
import json
import os # Assuming os is already imported or available in the environment

# Assuming tmp_dir is defined elsewhere, e.g., tmp_dir = "temp_data"
# For this example, let's define it if not present.
try:
    tmp_dir
except NameError:
    tmp_dir = "temp_data"
    if not os.path.exists(tmp_dir):

```

```

        os.makedirs(tmp_dir)

def generar_tweets_ejemplo(n=20):
    """
    Genera tweets de ejemplo usando la estructura real de Twitter API v2.
    Simula datos que se obtendrían usando tweepy.Client.search_recent_tweets()
    """
    usuarios = [
        {"id": "1001", "name": "Ana García", "username": "ana_tech",
         "followers": 2341},
        {"id": "1002", "name": "Bruno Silva", "username": "bruno_code",
         "followers": 1523},
        {"id": "1003", "name": "Clara Ruiz", "username": "clara_dev",
         "followers": 3421},
        {"id": "1004", "name": "Diego Mendoza", "username": "diego_data",
         "followers": 987},
        {"id": "1005", "name": "Elena Torres", "username": "elena_ai",
         "followers": 5432},
    ]

    temas = [
        ("Las estructuras de datos son fundamentales",
         ["EDD", "Programación"]),
        ("Python es un lenguaje muy versátil",
         ["Python", "Desarrollo"]),
        ("Los grafos tienen muchas aplicaciones prácticas",
         ["Grafos", "Algoritmos"]),
        ("El análisis de redes sociales es fascinante",
         ["RedesSociales", "DataScience"]),
        ("Machine learning está revolucionando el mundo",
         ["ML", "IA"]),
    ]

    tweets = []
    for i in range(n):
        usuario = random.choice(usuarios)
        tema, hashtags = random.choice(temas)

        # Formato real de Twitter API v2
        tweet = {
            "id": str(1000000000000000000 + i),
            "author_id": usuario["id"],
            "created_at": (f"2024-01-{random.randint(10,20):02d}T"
                          f"{random.randint(0,23):02d}:"
                          f"{random.randint(0,59):02d}:00.000Z"),
            "text": f"{tema} #{' #' .join(hashtags)}",
            "lang": "es",
            "public_metrics": {
                "retweet_count": random.randint(0, 50),
                "reply_count": random.randint(0, 20),
                "like_count": random.randint(0, 100),
                "quote_count": random.randint(0, 10),
                "impression_count": random.randint(100, 5000)
            },
            "entities": {
                "hashtags": [{"tag": tag} for tag in hashtags]
            },
            # Metadatos adicionales para procesamiento
            # No viene en API real, lo agregamos para simplificar ejemplos
            "_user": usuario
        }
        tweets.append(tweet)

    return tweets

# Generar tweets de ejemplo

```

```

tweets = generar_tweets_ejemplo(20)
print(f"Generados {len(tweets)} tweets de ejemplo "
      f"(estructura Twitter API v2)")
print(f"\nPrimer tweet:\n"
      f"{json.dumps(tweets[0], indent=2, ensure_ascii=False)}")

# Guardar tweets en formato JSONL
archivo_tweets = os.path.join(tmp_dir, "tweets_historicos.jsonl")

with open(archivo_tweets, 'w', encoding='utf-8') as f:
    for tweet in tweets:
        f.write(json.dumps(tweet, ensure_ascii=False) + '\n')

print(f"Tweets guardados en {archivo_tweets}")

# Verificar el contenido del archivo
with open(archivo_tweets, 'r', encoding='utf-8') as f:
    primeras_lineas = [f.readline() for _ in range(3)]

print(f"\nPrimeras 3 líneas del archivo:")
for i, linea in enumerate(primeras_lineas, 1):
    tweet = json.loads(linea)
    print(f"{i}. @{tweet['_user']['username']}: {tweet['text'][:50]}...")

```

5.1.2.4 Procesamiento y Análisis de Tweets

Ahora procesemos los tweets para extraer información útil:

```

def procesar_tweets_jsonl(archivo):
    """
    Procesa un archivo JSONL de tweets (formato Twitter API v2)
    y extrae estadísticas. Compatible con la estructura real que
    devuelve tweepy.
    """
    estadisticas = {
        "total_tweets": 0,
        "usuarios_unicos": set(),
        "hashtags": {},
        "total_retweets": 0,
        "total_likes": 0,
        "total_replies": 0,
        "tweets_por_usuario": {},
    }

    with open(archivo, 'r', encoding='utf-8') as f:
        for linea in f:
            tweet = json.loads(linea)
            estadisticas["total_tweets"] += 1

            # Usuario (de metadatos auxiliares _user)
            if "_user" in tweet:
                username = tweet["_user"]["username"]
                estadisticas["usuarios_unicos"].add(username)
                estadisticas["tweets_por_usuario"][username] = \
                    estadisticas["tweets_por_usuario"].get(username, 0) + 1

            # Hashtags (de entities)
            if "entities" in tweet and "hashtags" in tweet["entities"]:
                for hashtag_obj in tweet["entities"]["hashtags"]:
                    hashtag = hashtag_obj.get("tag", "")
                    if hashtag:
                        estadisticas["hashtags"][hashtag] = \
                            estadisticas["hashtags"].get(hashtag, 0) + 1

            # Métricas públicas (estructura real de API v2)
            if "public_metrics" in tweet:

```

```

        metrics = tweet["public_metrics"]
        estadisticas["total_retweets"] += metrics.get(
            "retweet_count", 0)
        estadisticas["total_likes"] += metrics.get("like_count", 0)
        estadisticas["total_replies"] += metrics.get(
            "reply_count", 0)

    # Convertir set a lista para serialización
    estadisticas["usuarios_unicos"] = list(estadisticas["usuarios_unicos"])

    return estadisticas

# Procesar los tweets
stats = procesar_tweets_jsonl(archivo_tweets)

print("=== Estadísticas de Tweets ===")
print(f"Total de tweets: {stats['total_tweets']}")
print(f"Usuarios únicos: {len(stats['usuarios_unicos'])}")
print(f"Total de retweets: {stats['total_retweets']}")
print(f"Total de likes: {stats['total_likes']}")
print(f"Total de replies: {stats['total_replies']}")

print("\n=== Hashtags más populares ===")
hashtags_ordenados = sorted(stats["hashtags"].items(),
                             key=lambda x: x[1], reverse=True)
for hashtag, count in hashtags_ordenados[:5]:
    print(f"#{hashtag}: {count} veces")

print("\n=== Usuarios más activos ===")
usuarios_ordenados = sorted(stats["tweets_por_usuario"].items(),
                             key=lambda x: x[1], reverse=True)
for usuario, count in usuarios_ordenados[:5]:
    print(f"@{usuario}: {count} tweets")

```

5.1.3 Aspectos Importantes de la Recuperación de Información en Redes Sociales

- Las redes sociales generan grandes volúmenes de datos que pueden ser analizados usando estructuras de datos y algoritmos apropiados.
- Los grafos son una representación natural de las redes sociales y permiten aplicar algoritmos de teoría de grafos.
- JSON y JSONL son formatos ideales para trabajar con datos de APIs y streams.
- La persistencia de datos es crucial para poder analizar y procesar la información posteriormente.

6. Anexos

6.1 Activación automática de entorno virtual en Git Bash

Para esta guía se asume que el sistema Windows cuenta con Python y Git Bash instalados.

6.1.1 Crear un entorno virtual

Parados sobre la carpeta donde queremos crear el entorno virtual, podemos hacer:

```
python -m venv 3 .venv
```

Donde 3 es la versión de Python a utilizar (podemos ser más específicos en la numeración como 3.13, o 3.13.1). `.venv` es el nombre del entorno virtual que vamos a crear, eso puede ser cualquier cosa que consideremos apropiado (también será el nombre de la carpeta que se creará donde ejecutamos `venv`).

6.1.2 Activar el entorno virtual

Para activar el entorno virtual, debemos usar los *scripts* que se crearon en la carpeta `.venv`.

```
source .venv/Scripts/activate
```

Si todo salió relativamente bien, es posible que en nuestro *prompt* aparezca el nombre del entorno virtual... por ejemplo el mío se ve así:

```
(.venv)
~/Projects/untref-edd/edd
$
```

(si, esas 3 líneas son todo el *prompt*)

6.1.3 Desactivar el entorno virtual

Simplemente ejecutando:

```
deactivate
```

6.1.4 Borrar el entorno virtual

Se debe borrar la carpeta que se había creado:

```
rm -r .venv
```

6.1.5 Activar el entorno virtual cuando entramos al directorio del “proyecto”

Primero debemos instalar una ayudita en nuestro archivo `~/.bashrc`:

```
# Helpers for `venv`
function set_local_venv {
    echo $1 > .python-version
}

function activate_venv {
    [ ! -f .python-version ] && return

    env_name=`cat .python-version`
    activate_script=$env_name/Scripts/activate

    [ ! -f $activate_script ] && return

    source $activate_script
}

# This will simulate chpwd hook effect on this bash context
export PROMPT_COMMAND=activate_venv
```

Tanto copiar esas líneas manualmente, como ejecutar el siguiente comando en Git Bash, habilita estas funciones.

```

echo -e "\n\n# Helpers for `venv`\nfunction set_local_venv {\n    echo \$1\n    > .python-version\n}\n\nfunction activate_venv {\n    [ ! -f .python-version ]\n    && return\n\n    env_name=`cat .python-version`\n    activate_script=\n    $env_name/Scripts/activate\n\n    [ ! -f $activate_script ] && return\n\n    source $activate_script\n}\n\n# This will simulate chpwd hook effect on this\nbash context\nexport PROMPT_COMMAND=activate_venv\n" >> $HOME/.bashrc

```

Cerramos todas las terminales y las volvemos a abrir.

6.1.5.1 Cómo usar la activación automática

Primero es necesario decirle a nuestro programita “qué entorno queremos activar”, para eso usamos un archivo oculto llamado `.python-version` (este es un nombre con convención que se usan en varias herramientas como pyenv). Una de las funciones creamos fue `set_local_venv`, que es utilidad que vamos a usar para “activar la activación automática en un determinado entorno virtual”.

Siguiendo con nuestro ejemplo, para prender el entorno `.venv`, nos paramos en la carpeta del entorno y desde ahí ejecutamos:

```
set_local_venv .venv
```

Y simplemente el entorno virtual `.venv` se activará cada vez que entremos en esa carpeta.

Importante: los entornos virtuales no se desactivaran automáticamente.

6.2 Registrarse como Desarrollador de Meta (Facebook)

Este instructivo detalla el proceso completo para registrarse como desarrollador de Meta (Facebook), crear una aplicación y generar el Token de Acceso de Usuario necesario para realizar análisis de datos de redes sociales, explorar grafos de amistades y acceder a información pública usando Python.

6.2.1 Como obtener un “Access Token” de Facebook para Análisis de Datos

El primer paso es crear una cuenta de desarrollador en la plataforma Meta.

1. Ir al Portal de Desarrolladores
 - Abrir un navegador web y acceder al portal oficial: <https://developers.facebook.com/>.
2. Iniciar Sesión
 - Hacer clic en el botón “**Empezar**” en la esquina superior derecha e iniciar sesión con una cuenta personal de Facebook.

Nota

Requisito importante: Necesitas tener una cuenta de Facebook activa y verificada para poder solicitar acceso como desarrollador.

1. Verificar la Cuenta
 - Seguir las instrucciones para completar el registro. Esto incluye:
 - ▶ Aceptar las **Condiciones de la plataforma** y las **Políticas para desarrolladores**.
 - ▶ Verificar la identidad proporcionando un número de teléfono o correo electrónico para recibir un código de confirmación.
 - ▶ Completar el perfil de desarrollador con información sobre el propósito de uso.
2. Proceso de Verificación
 - Meta puede solicitar información adicional sobre el uso previsto de la API.
 - La verificación puede ser inmediata o tomar hasta 24-48 horas.
 - Algunos casos de uso requieren verificación empresarial adicional.

6.2.2 Crear una Nueva Aplicación de Consumidor

Una vez aprobada tu cuenta de desarrollador, puedes crear aplicaciones para acceder a la Graph API.

1. Crear una Nueva Aplicación
 - Desde el panel de desarrolladores, hacer clic en el botón verde “**Crear aplicación**”.
2. Seleccionar el Tipo de Aplicación
 - Elegir la opción “**Otro**” para casos de uso personalizados.
 - En la pantalla siguiente, seleccionar “**Ninguno**”. Esto proporciona un “**lienzo en blanco**”, ideal para trabajar directamente con la API sin configuraciones predefinidas.
 - Alternativamente, puedes seleccionar:
 - ▶ “**Consumidor**”: Para aplicaciones que consumen datos de Facebook.
 - ▶ “**Empresa**”: Para herramientas internas de empresa.
 - ▶ “**Gaming**”: Para aplicaciones de juegos.
3. Configurar Detalles de la Aplicación

- En el campo “**Nombre de la aplicación**”, escribir un nombre descriptivo (ej: “Analizador de Grafos Académico”).
 - **Propósito de la aplicación**: Describir claramente el caso de uso (ej: “Análisis académico de redes sociales”).
 - **Correo de contacto**: Verificar que el correo de contacto sea correcto.
 - **Categoría de la aplicación**: Seleccionar la categoría más apropiada.
4. Crear y Configurar
 - Hacer clic en “**Crear aplicación**”. Se podría solicitar la contraseña de Facebook por seguridad.
 - Una vez creada, se abrirá el panel de control de la aplicación.

6.2.3 Generar y Configurar el Token de Acceso de Usuario

Este es el paso más importante: obtener la “*llave*” para acceder a los datos de la Graph API.

1. Abrir el Explorador de la **API Graph**
 - En el menú lateral izquierdo del panel de la aplicación, navegar a **Herramientas > Explorador de la API Graph**.
2. Configurar la Solicitud del Token
 - En la parte superior derecha de la pantalla, verificar los siguientes campos:
 - ▶ **Aplicación de Meta**: Asegurarse de que esté seleccionada la aplicación recién creada.
 - ▶ **Usuario o página**: Confirmar que esté elegida la opción “**Identificador de usuario**”.
 - ▶ **Versión de API**: Seleccionar la versión más reciente (ej: v18.0 o superior).
3. Configurar Permisos Necesarios
 - Hacer clic en la pestaña “**Permisos**”.
 - Se desplegará una lista de categorías de permisos:

Permisos básicos recomendados:

- **user_likes**: Para leer páginas que le han gustado al usuario.
- **user_posts**: Para acceder a las publicaciones del usuario.
- **user_friends**: Para obtener la lista de amigos (limitado).
- **email**: Para obtener el correo electrónico del usuario.

1. Generar el Token de Acceso
 - Hacer clic en el botón azul “**Generate Access Token**”.
 - Aparecerá una ventana emergente de Facebook solicitando confirmar los permisos. **Aceptar** para continuar.
 - El sistema generará un **Token de Acceso de Usuario** de corta duración (1-2 horas).
2. Copiar y Extender el Token
 - El campo “**Identificador de acceso**” contendrá una larga cadena de caracteres.
 - **Copiar inmediatamente** el token haciendo clic en el icono de copiar.

Advertencia

Los tokens de usuario tienen duración limitada. Para uso prolongado, considera generar tokens de larga duración o implementar renovación automática.

6.2.4 Explorar la Graph API con Consultas Útiles

El Explorador de Graph API permite probar diferentes consultas antes de implementarlas en código.

6.2.4.1 Consultas Básicas de Usuario

1. Información del Usuario Actual

`/me?fields=id,name,email,birthday,location`

1. Páginas que le Gustan al Usuario

`/me/likes?fields=name,category,fan_count,website`

1. Publicaciones del Usuario

`/me/posts?
fields=message,created_time,likes.summary(true),comments.summary(true)`

6.2.4.2 Consultas para Análisis de Redes

1. Lista de Amigos (limitada)

`/me/friends?fields=name,id`

Nota

Solo devuelve amigos que también usan la aplicación debido a restricciones de privacidad.

1. Información de una Página Específica

`/[{page-id}]?fields=name,fan_count,category,website,about,location`

1. Publicaciones de una Página Pública

`/[{page-id}]/posts?
fields=message,created_time,likes.summary(true),shares.summary(true)`

6.2.4.3 Consultas Avanzadas para Investigación

1. Eventos Públicos

`/search?type=event&q=tecnología&fields=name,description,start_time,place`

1. Lugares Cercanos

`/search?type=place¢er=lat,lng&distance=1000&fields=name,location,checkins`

1. Análisis de Engagement

`/[{post-id}]?
fields=reactions.summary(total_count).limit(0),comments.summary(total_count).limit(0),shares.summary(total_count).limit(0)`

6.2.5 Configurar Permisos Avanzados y Revisión de Aplicación

Para acceder a datos más sensibles, Meta requiere un proceso de revisión.

1. Permisos que Requieren Revisión

- `user_posts`: Publicaciones del usuario
- `user_photos`: Fotos del usuario
- `manage_pages`: Gestión de páginas
- `publish_pages`: Publicar en páginas

2. Proceso de Revisión de Aplicación

- Ir a **Revisión de aplicación** en el panel lateral.
- Seleccionar los permisos necesarios y proporcionar:
 - **Justificación detallada** del uso.
 - **Capturas de pantalla** de la funcionalidad.
 - **Video demostración** de cómo se usan los datos.

3. Configurar Webhook (Opcional)

- Para recibir actualizaciones en tiempo real:
 - Ir a **Productos > Webhooks**.
 - Configurar URL de endpoint y eventos de interés.

6.2.6 Tipos de Tokens y Gestión de Autenticación

Meta maneja diferentes tipos de tokens según el caso de uso:

6.2.6.1 Token de Acceso de Usuario

- **Duración:** 1-2 horas por defecto
- **Uso:** Acceso a datos del usuario autenticado
- **Extensión:** Puede extenderse a 60 días

6.2.6.2 Token de Acceso de Página

- **Duración:** No expira (mientras la aplicación tenga permisos)
- **Uso:** Gestión y publicación en páginas
- **Obtención:** A través del token de usuario con permisos `manage_pages`

6.2.6.3 Token de Aplicación

- **Duración:** No expira
- **Uso:** Acceso a datos públicos y gestión de aplicación
- **Formato:** `{app-id}|{app-secret}`

6.2.6.4 Extender la Duración del Token

```
# Extender token de usuario (60 días)
curl -i -X GET "https://graph.facebook.com/oauth/access_token?grant_type=fb_exchange_token&client_id={app-id}&client_secret={app-secret}&fb_exchange_token={short-token}"
```

6.2.7 Probar las Credenciales y Conectividad

Una vez obtenido el token, es importante verificar que funciona correctamente.

1. Verificar Token con curl

```
# Verificar información del usuario
curl -i -X GET "https://graph.facebook.com/me?access_token={your-token}"

# Verificar páginas que le gustan
curl -i -X GET "https://graph.facebook.com/me/likes?access_token={your-token}"
```

1. Usar el Debugger de Tokens

- Ir a Facebook Debugger
- Pegar el token para verificar:
 - Validez y expiración
 - Permisos otorgados
 - ID de aplicación y usuario

2. Verificar con Python

```
import requests

token = "YOUR_ACCESS_TOKEN"
response = requests.get(f"https://graph.facebook.com/me?access_token={token}")

if response.status_code == 200:
    print("✓ Token válido:", response.json())
else:
    print("✗ Error:", response.text)
```

6.2.8 Limitaciones y Restricciones Actuales

Es crucial entender las limitaciones de la Graph API tras los cambios de privacidad.

6.2.8.1 Restricciones de Datos de Usuario

- **Solo datos propios:** Acceso limitado a datos del usuario autenticado
- **Amigos limitados:** Solo amigos que también usan la aplicación
- **Revisión obligatoria:** Muchos permisos requieren proceso de revisión
- **Rate limiting:** Límites estrictos en número de llamadas por hora

6.2.8.2 Datos Públicos Disponibles

- **Páginas públicas:** Información básica y publicaciones públicas
- **Eventos públicos:** Eventos marcados como públicos
- **Lugares:** Información de ubicaciones y check-ins públicos
- **Grupos públicos:** Contenido de grupos públicos (limitado)

6.2.8.3 Rate Limits y Cuotas

- **200 llamadas por hora** por usuario para aplicaciones en desarrollo
- **Límites más altos** para aplicaciones verificadas
- **Throttling automático** cuando se exceden los límites

6.2.9 Consideraciones de Seguridad

6.2.9.1 Protección de Credenciales

1. **Nunca hardcodear** tokens en el código fuente
2. **Usar variables de entorno** para almacenar credenciales
3. **Implementar renovación automática** de tokens
4. **Monitorear el uso** de tokens en el panel de desarrollador

6.2.9.2 Ejemplo de Configuración Segura

```
import os
from dotenv import load_dotenv

# Cargar variables de entorno
load_dotenv()

# Obtener credenciales de forma segura
FACEBOOK_ACCESS_TOKEN = os.getenv("FACEBOOK_ACCESS_TOKEN")
FACEBOOK_APP_ID = os.getenv("FACEBOOK_APP_ID")
FACEBOOK_APP_SECRET = os.getenv("FACEBOOK_APP_SECRET")

# Verificar que las credenciales están disponibles
if not all([FACEBOOK_ACCESS_TOKEN, FACEBOOK_APP_ID, FACEBOOK_APP_SECRET]):
    raise ValueError("Faltan credenciales de Facebook en variables de entorno")
```

6.2.9.3 Archivo .env (ejemplo)

```
# Credenciales de Facebook
FACEBOOK_ACCESS_TOKEN=your_user_access_token_here
FACEBOOK_APP_ID=your_app_id_here
FACEBOOK_APP_SECRET=your_app_secret_here

# Configuración adicional
FACEBOOK_API_VERSION=v18.0
FACEBOOK_RATE_LIMIT_DELAY=1
```

6.2.9.4 Mejores Prácticas de Seguridad

1. **Rotación de tokens:** Renovar tokens periódicamente
2. **Monitoreo de uso:** Revisar logs de API en el panel de desarrollador
3. **Principio de menor privilegio:** Solo solicitar permisos necesarios
4. **Validación de entrada:** Sanitizar datos recibidos de la API
5. **HTTPS obligatorio:** Usar siempre conexiones seguras

6.2.10 Documentación y Recursos Útiles

6.2.10.1 Documentación Oficial

- Meta for Developers - Getting Started
- Graph API Reference
- Graph API Explorer
- Facebook SDK for Python

6.2.10.2 Librerías de Python Recomendadas

- **facebook-sdk**: SDK de terceros/comunidad para interactuar con Graph API en Python
- **requests**: Para llamadas HTTP directas (recomendado para máxima compatibilidad)
- **python-facebook-api**: Alternativa moderna mantenida por la comunidad

6.2.10.3 Herramientas de Desarrollo

- **Graph API Explorer**: Probar consultas interactivamente
- **Access Token Debugger**: Verificar tokens
- **Sharing Debugger**: Probar compartir contenido

6.2.10.4 Ejemplos de Endpoints Útiles

- **Usuario actual**: /me
- **Páginas favoritas**: /me/likes
- **Publicaciones**: /me/posts
- **Información de página**: /{page-id}
- **Publicaciones de página**: /{page-id}/posts
- **Eventos**: /me/events

Importante

Nota sobre privacidad y cambios recientes: La API de Facebook Graph ha limitado significativamente el acceso a datos de usuarios desde el escándalo de Cambridge Analytica en 2018. Actualmente, solo se puede acceder a datos del propio usuario autenticado y datos públicos. Muchas funcionalidades que antes estaban disponibles ahora requieren revisión de aplicación y justificación detallada del caso de uso.

6.3 Registrarse como Desarrollador de X (Twitter)

Este instructivo detalla el proceso completo para registrarse como desarrollador de X (anteriormente Twitter), crear una aplicación y generar las credenciales necesarias (Bearer Token y Access Token) para realizar análisis de datos y acceder tanto a tweets históricos como a streams en tiempo real usando Python.

6.3.1 Como obtener un “*Bearer Token*” y “*Access Token*” de X (Twitter) para Análisis de Datos

El primer paso es crear una cuenta de desarrollador en la plataforma X.

1. Ir al Portal de Desarrolladores
 - Abrir un navegador web y acceder al portal oficial: <https://developer.twitter.com/>.
2. Iniciar Sesión
 - Hacer clic en el botón “**Sign up**” en la esquina superior derecha e iniciar sesión con una cuenta personal de X (Twitter).

Nota

Requisito importante: Necesitas tener una cuenta de X/Twitter verificada con un número de teléfono válido para poder solicitar acceso como desarrollador.

1. Solicitar Acceso como Desarrollador
 - Hacer clic en “**Apply for a developer account**”.
 - Completar el formulario de solicitud:
 - ▶ **Propósito de uso:** Seleccionar entre Académico/Investigación, Comercial, Personal, o Estudiante.
 - ▶ **Descripción detallada:** Explicar específicamente cómo planeas usar la API (ej: “Análisis académico de tendencias en redes sociales para curso de Estructuras de Datos”).
 - ▶ **Casos de uso específicos:** Indicar si vas a analizar tweets, hacer streaming, publicar contenido, etc.
2. Verificar la Solicitud
 - Revisar toda la información y enviar la solicitud.
 - Aceptar las **Condiciones de Servicio para Desarrolladores** y la **Política de Uso de la API**.
 - La aprobación puede tomar desde minutos hasta 1-2 días hábiles.

6.3.2 Crear una Nueva Aplicación (App)

Una vez aprobada tu cuenta de desarrollador, puedes crear aplicaciones para acceder a la API.

1. Acceder al Panel de Desarrollador
 - Desde el Developer Portal, hacer clic en “**Projects & Apps**” en el menú lateral.
2. Crear un Nuevo Proyecto
 - Hacer clic en el botón “**+ Create Project**”.
 - Asignar un nombre descriptivo al proyecto (ej: “Analizador de Redes Sociales Académico”).
 - Seleccionar el caso de uso más apropiado:
 - ▶ “**Exploring the API**”: Para aprendizaje y experimentación.
 - ▶ “**Making a bot**”: Para crear bots automatizados.
 - ▶ “**Doing academic research**”: Para investigación académica.
 - ▶ “**Building internal tools**”: Para herramientas empresariales.
3. Crear una Aplicación dentro del Proyecto

- Después de crear el proyecto, X te pedirá crear una aplicación.
- Asignar un nombre único a la aplicación (ej: "TwitterDataAnalyzer2024").
- Proporcionar una descripción clara del propósito de la aplicación.

6.3.3 Obtener las Credenciales de Autenticación

Este es el paso más importante: generar las **"llaves"** para acceder a los datos de X.

1. Configurar Niveles de Acceso

- En el panel de tu aplicación, ir a la pestaña **"Settings"**.
- En **"App permissions"**, configurar los permisos necesarios:
 - **Read**: Para leer tweets, perfiles, y datos públicos.
 - **Write**: Para publicar tweets (opcional para análisis de datos).
 - **Direct Messages**: Para acceder a mensajes directos (raramente necesario).

Importante

Para análisis de datos, generalmente solo necesitas permisos de **"Read"**.

1. Generar el Bearer Token (API v2)

- Ir a la pestaña **"Keys and Tokens"**.
- En la sección **"Bearer Token"**, hacer clic en **"Generate"**.
- **Copiar y guardar** inmediatamente el Bearer Token en un lugar seguro.

Advertencia

El Bearer Token se muestra solo una vez. Si lo pierdes, deberás regenerar uno nuevo.

1. Generar Consumer Keys (API v1.1)

- En la sección **"Consumer Keys"**, hacer clic en **"Generate"**.
- Se generarán automáticamente:
 - **API Key** (Consumer Key)
 - **API Secret Key** (Consumer Secret)
- **Copiar y guardar** ambas credenciales de forma segura.

2. Generar Access Token y Access Token Secret

- En la sección **"Access Token and Secret"**, hacer clic en **"Generate"**.
- Se generarán:
 - **Access Token**
 - **Access Token Secret**
- **Copiar y guardar** ambas credenciales.

6.3.4 Configurar Permisos para Streaming y Búsqueda

Para acceder a diferentes funcionalidades de la API, necesitas configurar permisos específicos.

1. Configurar Permisos de la Aplicación

- En **"App permissions"**, seleccionar el nivel apropiado:
 - **Read**: Suficiente para leer tweets históricos y hacer streaming.
 - **Read and Write**: Si planeas publicar tweets desde tu aplicación.
- Guardar los cambios.

2. Verificar Nivel de Acceso a la API

- X ofrece diferentes niveles de acceso:
 - **Free**: 500,000 tweets por mes, acceso básico a API v2.

- ▶ **Basic** (\$100/mes): 10 millones de tweets por mes, acceso completo a API v2.
 - ▶ **Pro** (\$5,000/mes): 50 millones de tweets por mes, acceso a datos históricos completos.
 - ▶ **Enterprise**: Acceso personalizado y soporte premium.
3. Configurar Webhook URLs (Opcional)
 - Si planeas usar webhooks para recibir datos en tiempo real:
 - ▶ Ir a “**Settings**” > “**Webhook URLs**”.
 - ▶ Agregar la URL de tu servidor que recibirá los datos.

6.3.5 Probar las Credenciales

Una vez obtenidas las credenciales, es importante verificar que funcionan correctamente.

1. Verificar Bearer Token
 - Usar el **Postman** o **curl** para hacer una petición de prueba:

```
curl -H "Authorization: Bearer YOUR_BEARER_TOKEN" \
      "https://api.twitter.com/2/tweets/search/recent?query=python"
```

1. Verificar Access Tokens
 - Para OAuth 1.0a, las credenciales se usan en conjunto para firmar las peticiones.
 - La librería tweepy de Python maneja automáticamente la autenticación.

6.3.6 Niveles de Acceso y Limitaciones

Es importante entender las limitaciones de cada nivel de acceso:

6.3.6.1 Free Tier (Gratuito)

- **500,000 tweets** por mes
- **1 aplicación** por proyecto
- **Acceso a API v2** básica
- **Search Recent** (últimos 7 días)
- **Filtered Stream** básico
- **Sin acceso a datos históricos** completos

6.3.6.2 Basic Tier (\$100/mes)

- **10 millones de tweets** por mes
- **3 aplicaciones** por proyecto
- **Acceso completo a API v2**
- **Search Recent y Archive** (histórico completo)
- **Filtered Stream** avanzado
- **Soporte estándar**

6.3.6.3 Pro Tier (\$5,000/mes)

- **50 millones de tweets** por mes
- **Aplicaciones ilimitadas**
- **Acceso a Full Archive Search**
- **Datos históricos** desde 2006
- **Analytics y métricas** avanzadas
- **Soporte prioritario**

6.3.7 Documentación y Recursos Útiles

6.3.7.1 Documentación Oficial

- X API Documentation
- API v2 Quick Start Guide
- Authentication Guide
- Rate Limits and Pricing

6.3.7.2 Librerías de Python Recomendadas

- **Tweepy**: Librería de terceros (muy popular) para Python
- **Python-Twitter**: Alternativa robusta
- **TwitterAPI**: Acceso directo a endpoints

6.3.7.3 Ejemplos de Endpoints Útiles

- **Search Recent Tweets**: /2/tweets/search/recent
- **Search Historical**: /2/tweets/search/all (requiere Academic Research o Basic+)
- **User Timeline**: /2/users/:id/tweets
- **Filtered Stream**: /2/tweets/search/stream
- **Sample Stream**: /2/tweets/sample/stream

6.3.8 Consideraciones de Seguridad

6.3.8.1 Protección de Credenciales

1. **Nunca hardcodear** las credenciales en el código fuente.
2. **Usar variables de entorno** o archivos de configuración privados.
3. **Agregar archivos de credenciales** al `.gitignore`.
4. **Regenerar tokens** periódicamente por seguridad.

6.3.8.2 Ejemplo de Configuración Segura

```
import os
from dotenv import load_dotenv

# Cargar variables de entorno
load_dotenv()

# Obtener credenciales de forma segura
BEARER_TOKEN = os.getenv("TWITTER_BEARER_TOKEN")
API_KEY = os.getenv("TWITTER_API_KEY")
API_SECRET = os.getenv("TWITTER_API_SECRET")
ACCESS_TOKEN = os.getenv("TWITTER_ACCESS_TOKEN")
ACCESS_TOKEN_SECRET = os.getenv("TWITTER_ACCESS_TOKEN_SECRET")
```

Importante

Nota sobre cambios recientes: Desde la adquisición por Elon Musk en 2022, X ha modificado significativamente sus políticas de API. El acceso gratuito está más limitado y muchas funcionalidades requieren suscripciones pagadas. Siempre verificar la documentación oficial para conocer las limitaciones actuales.

7. Referencias

7.1 Bibliografía

(Baeza-Yates & Ribeiro-Neto, 2011; Büttcher et al., 2010; Manning et al., 2008; Mitchell, 2024; Pilgrim, 2009; Python Software Foundation, 2024a; 2024b; 2024c; Witten et al., 1999; Zobel & Moffat, 2006)

Bibliografía

- Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology behind Search* (2nd ed.). Addison-Wesley.
- Büttcher, S., Clarke, C. L. A., & Cormack, G. V. (2010). *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press.
- Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. <https://nlp.stanford.edu/IR-book/>
- Mitchell, R. (2024). *Web Scraping with Python: Data Extraction from the Modern Web* (3rd ed.). O'Reilly Media. <https://www.oreilly.com/library/view/web-scraping-with/9781098145347/>
- Pilgrim, M. (2009). *Dive Into Python 3*. Apress. <https://diveintopython3.net/>
- Python Software Foundation. (2024b,). *Documentación de Python*. Python Software Foundation. <https://docs.python.org/es/3.13/>
- Python Software Foundation. (2024a,). *El tutorial de Python*. Python Software Foundation. <https://docs.python.org/es/3.13/tutorial/index.html>
- Python Software Foundation. (2024c,). *PEP 8 – Guía de estilo para código Python*. Python Software Foundation. <https://peps.python.org/pep-0008/>
- Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd ed.). Morgan Kaufmann. <https://sigmodrecord.org/publications/sigmodRecord/0406/RB2.Nagaraj.pdf>
- Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 6–es. <https://doi.org/10.1145/1132956.1132959>