Parte I: Programación en un lenguaje orientado a objetos

1. Introducción a los lenguajes de programación
2. Datos y expresiones
3. Clases
4. Estructuras algorítmicas
5. *Estructuras de Datos*
   - Tablas. Algoritmos de recorrido. Algoritmos de búsqueda. Conjuntos. Tablas multidimensionales. El paquete NumPy. Diccionarios. Tipos enumerados.
6. Tratamiento de errores
7. Entrada/salida
8. Herencia y polimorfismo

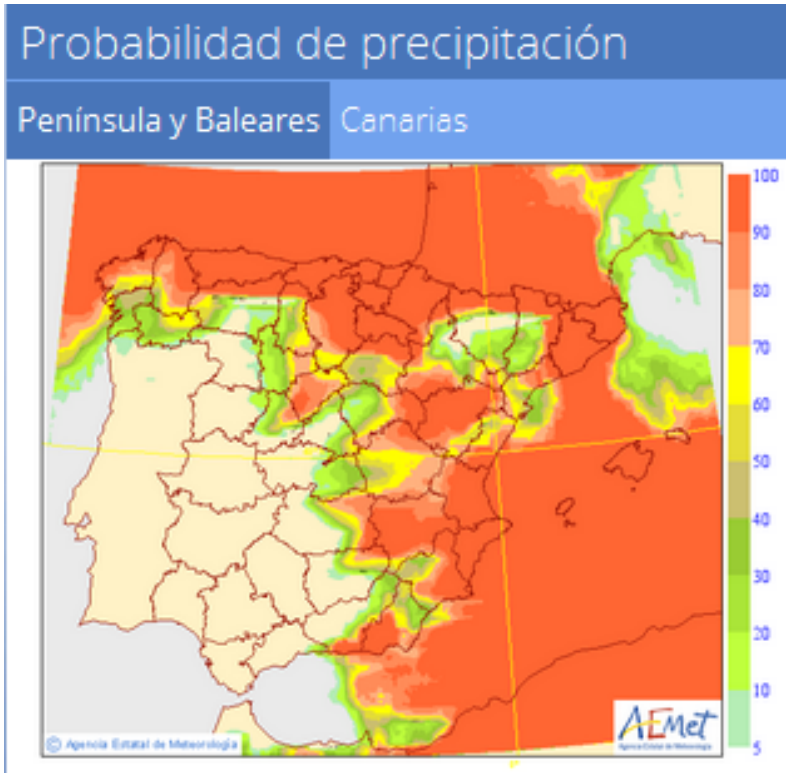
5.1 Tablas

Muchos programas deben manejar numerosos datos en forma de tablas

Listado de estaciones de esquí

Pirineo Catalán	Estado	Km. abiertos	Nieve	Meteo
Baqueira Beret	Cerrada	- / 155	-	
Boí Taüll	Cerrada	- / 48	-	
Espot Esquí	Cerrada	- / 25	-	
La Molina	Abierta	19 / 67	60	
Masella	Abierta	48 / 74,5	100	
Port Ainé	Cerrada	- / 26,7	-	
Port del Comte	Cerrada	- / 50	-	
Tavascán	Cerrada	- / 6	-	
Vall de Nuria	Cerrada	- / 7,6	-	
Vallter 2000	Abierta	13 / 18,73	60	
Pirineo Aragonés	**Estado**	**Km. abiertos**	**Nieve**	**Meteo**
Astún	Cerrada	- / 50	-	
Candanchú	Abierta	29,1 / 50,6	210	
Cerler	Abierta	oct-79	220	
Formigal	Abierta	83 / 137	260	
Panticosa	Cerrada	- / 39	-	
Cordillera Cantábrica	**Estado**	**Km. abiertos**	**Nieve**	**Meteo**
Alto Campoo	Abierta	10,16 / 27,7	120	
Fuentes de Invierno	Abierta	8,5 / 8,76	210	

Imagen: tabla bidimensional de píxeles, cada uno con un valor numérico de color

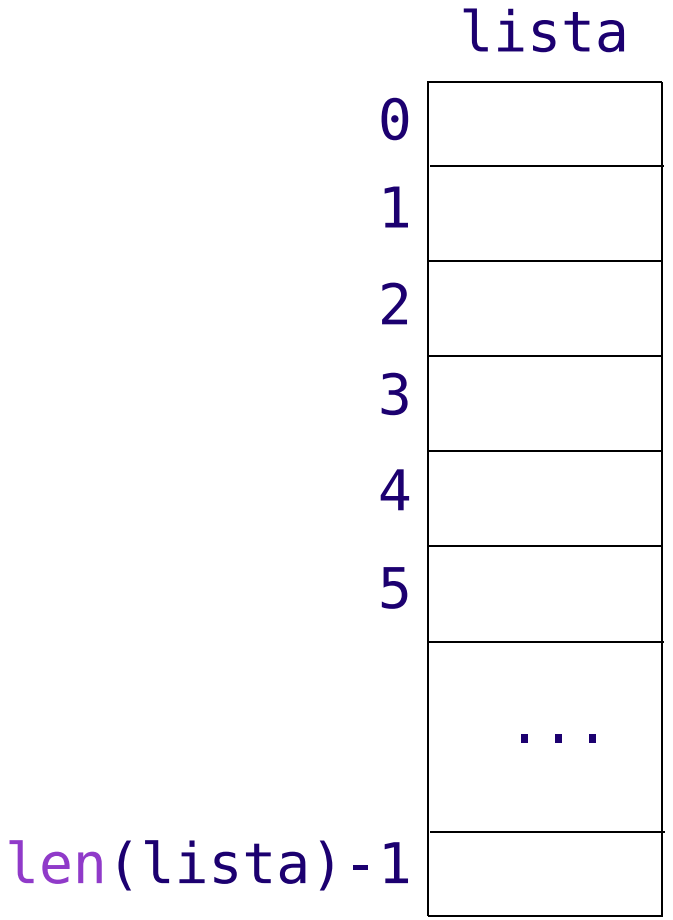


Fuente: aemet

Construcción de tablas mediante listas y tuplas

Las listas y tuplas (ver sección 2.8) permiten guardar muchos datos del mismo tipo, como en un casillero

- se agrupan bajo un nombre común
- se utiliza un *índice* numérico para referirse al dato individual
  - en Python el índice comienza por `0`
- los datos son referencias a objetos
- el tamaño se obtiene con `len()`



Listas y tuplas

Son dos de las *secuencias* definidas en Python

tuplas	inmutables
listas	mutables

En las secuencias se dispone de la operación de concatenación

```
lista_completa = lista_parcial_1 + lista_parcial_2
```

Hay operaciones para invertir y ordenar los elementos:

	in situ, modifica la lista	**para hacer un recorrido o crear una nueva secuencia, dejando la secuencia original sin cambiar**
Se aplica a	**listas**	**listas y tuplas**
Ordenar	`l.sort()`	`for x in sorted(l):`
Dar la vuelta	`l.reverse()`	`for x in reversed(l):`

Listas y tuplas (cont.)

Recordamos también otros usos de las secuencias

- Elemento individual:

```
lista[i]    # elemento i
lista[0]    # primer elemento
lista[-1]   # último elemento
```

- Rodaja:

```
lista[i:j]  # del i (incluido) al j (excluido)
lista[i:]   # del i al último, ambos incluidos
lista[i:-1] # del i (incluido) al último (excluido)
lista[:j]   # del 0 (incluido) al j (excluido)
```

Ejemplos con listas

Considerar una lista con 5 elementos

```
mi_lista = [0, 5, 10, 15, 20]
```

Podemos obtener una rodaja con los tres elementos centrales

```
mi_lista[1:-1]  # Vale [5, 10, 15]
```

También así

```
mi_lista[1:4]  # Vale [5, 10, 15]
```

Ahora queremos cambiar esos tres elementos centrales por los valores 6, 7 y 8:

```
mi_lista[1:4] = [6, 7, 8]  # mi_lista vale [0, 6, 7, 8, 20]
```

Ejemplos con listas (cont.)

Si el número de elementos reemplazados es distinto, la lista se encoje o amplía

```
frase =  ['Yo', 'estoy', 'muy', 'enfadado', 'con', 'Andrés',
          'y', 'Elena']

# Cambio dos palabras por una
frase[2:4] = ['contento']
# Ahora frase = ['Yo', 'estoy', 'contento', 'con', 'Andrés',
#                'y', 'Elena']
# Cambio una palabra por tres
frase[5:6] = [',', 'Laura', 'y']
# Ahora frase = ['Yo', 'estoy', 'contento', 'con', 'Andrés',
#                ',', 'Laura', 'y', 'Elena']
```

Observar que si reemplazamos por un único elemento hay que ponerlo entre `[]` para que sea una lista

Ejemplos con listas (cont.)

Al hacer asignación con elementos individuales de la lista no se puede cambiar su tamaño:

```
# Reemplazamos el elemento 'dos'
lis = ['uno', 'dos', 'tres', 'cuatro']
lis[1] = ['one', 'two', 'three']
# Ahora lis vale
# ['uno', ['one', 'two', 'three'], 'tres', 'cuatro']
```

En cambio, si usamos una rodaja es distinto

```
# Insertamos entre los elementos 'uno' y 'dos'
lis = ['uno', 'dos', 'tres', 'cuatro']
lis[1:1] = ['one', 'two', 'three']
# Ahora lis vale
# ['uno', 'one', 'two', 'three', 'dos', 'tres', 'cuatro']
```

Operaciones con listas

Vimos en el capítulo 2 la operación para añadir al final de una lista: `append()`

Hay una operación llamada `pop()` para sacar elementos del final

```
elemento = lista.pop()
```

En las listas, añadir o quitar elementos al principio es ineficiente

- para ello hay otra estructura de datos llamada `collections.deque`
  https://docs.python.org/3/library/collections.html#collections.deque

5.2 Recorrido de tablas

El recorrido para hacer algo con todas las casillas es un algoritmo muy frecuente en tablas

```
para cada val en tabla
    trabajar con val
fin para
```

La implementación en Python sería:

```
for val in tabla:
    #trabajar con val
```

Ejemplo de recorrido, con salida anticipada

```
def main():
    """
    Creamos una lista de comestibles y la recorremos mostrando
    en pantalla los que nos gustan, pero terminamos si alguno
    no nos gusta
    """

    # Obtenemos la lista de comestibles
    comida: list[str] = ["queso", "jamón", ...]
    for alimento in comida:
        if alimento == "apio":
            print("¡No me gusta el apio!")
            break
        print("Me encanta el " + alimento)
    print("Ya he terminado de comer")
```

Variantes de recorridos

Recorrido en orden inverso:

```
for val in reversed(tabla):
    #trabajar con val
```

Recorrido en orden:

```
for val in sorted(tabla):
    #trabajar con val
```

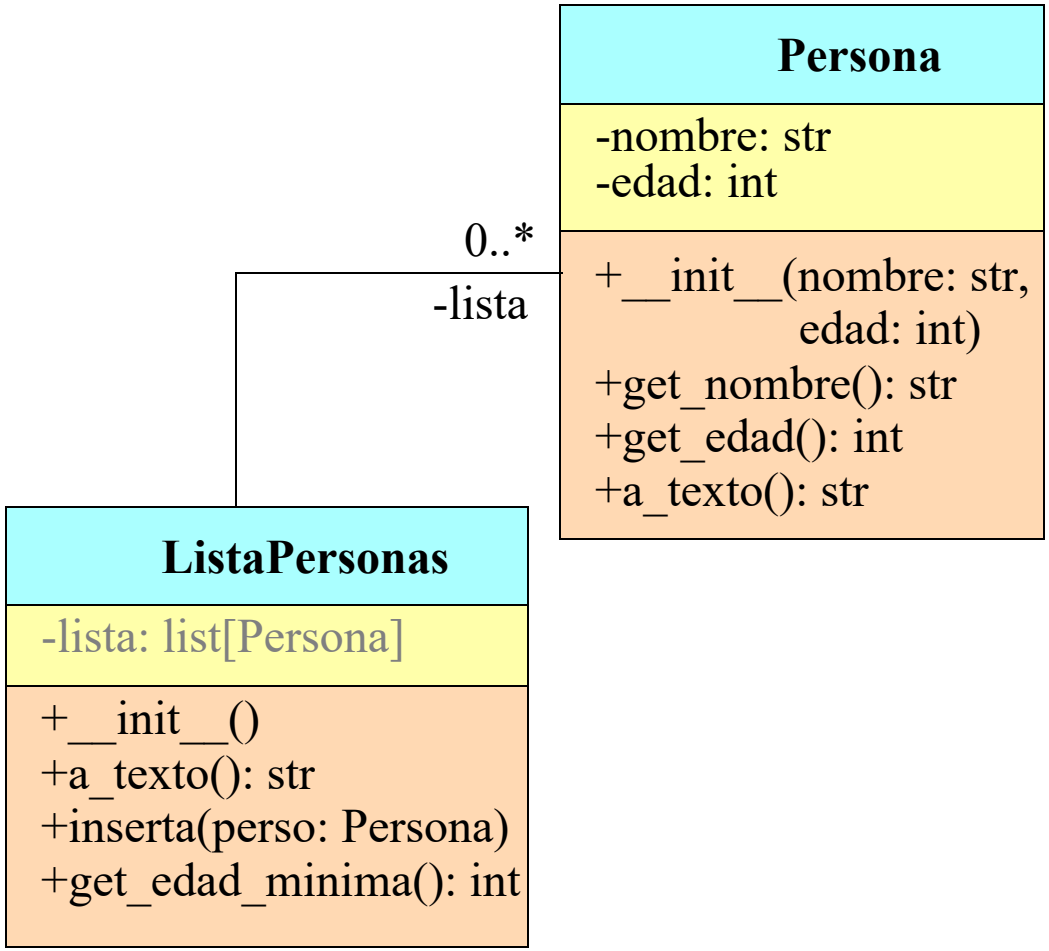
Recorrido parcial, usando una rodaja de la tabla:

```
for val in tabla[inicio:final]:
    #trabajar con val
```

Ejemplo de recorrido parcial, para calcular un mínimo

Queremos crear las siguientes clases:

- `Persona`: contiene los datos de una persona y operaciones asociadas
- `ListaPersonas`: contiene una lista de personas y operaciones asociadas



Calcular el mínimo con recorrido completo y parcial

En el capítulo 4 vimos un algoritmo de cálculo de máximo con un recorrido completo (el de mínimo es simétrico).

Ahora vemos uno ligeramente diferente. Ponemos aquí los dos para observar las diferencias:

Con recorrido completo	Con recorrido parcial

`minimo = infinito`
`para cada num en lista:`
`  si num < minimo entonces`
`  minimo = num`
`  fin si`
`fin para`
`# minimo contiene el resultado final`
	`minimo = lista[0]`
	`para cada num en lista[1:]:`
	`  si num < minimo entonces`
	`  minimo = num`
	`  fin si`
	`fin para`
	`# minimo contiene el resultado final`

Ejemplo (cont.)

```
# -*- coding: utf-8 -*-
"""
Módulo con operaciones sobre listas de personas

@author: Michael
@date  : mar 2023
"""

class Persona:
    """
    Clase que contiene los datos de una persona

    Attributes:
        __nombre : nombre de la persona
        __edad   : edad de la persona
    """ __
```

Ejemplo (cont.)

```
    def __init__(self, nombre: str, edad: int):
        """
        Constructor que da valor al nombre y la edad a
        partir de los argumentos
        """
        self.__nombre: str = nombre
        self.__edad: int = edad

    def get_nombre(self) -> str:
        """
        Obtiene el nombre de la persona
        """
        return self.__nombre

    def get_edad(self) -> int:
        """
        Obtiene la edad de la persona
        """
        return self.__edad
```

Ejemplo (cont.)

```
    def a_texto(self) -> str:
        """
        Retorna una representación textual de la persona
        con el nombre y la edad entre paréntesis
        separados por :
        """
        return f"({self.__nombre}:{self.__edad})"
```

Ejemplo (cont.):

```
class ListaPersonas:
    """
    Representa una lista de objetos de la clase Persona

    Attributes:
        __lista: una lista de personas
    """

    def __init__(self):
        """
        Crea la lista vacía
        """
        self.__lista: list[Persona] = []
```

Ejemplo (cont.)

```
    def a_texto(self) -> str:
        """
        Retorna una representación en texto de la lista, con
        los elementos separados por comas
        """
        texto: str = "["
        # recorremos todas las personas menos la última
        for perso in self.__lista[:-1]:
            texto += perso.a_texto()+", "
        # y ahora añadimos la última persona
        texto += self.__lista[-1].a_texto()+"]"
        return texto

    def inserta(self, perso: Persona):
        """
        Añade la persona perso al final de la lista
        """
        self.__lista.append(perso)
```

Ejemplo (cont.)

```
def get_edad_minima(self) -> int:
    """
    Retorna la edad mínima de las personas de la lista
    """
    # mini contiene la edad mínima encontrada hasta ahora
    # empezamos por la primera persona
    mini: int = self.__lista[0].get_edad()
    # comparamos con el resto de personas
    for perso in self.__lista[1:]:
        edad = perso.get_edad()
        if edad < mini:
            mini = edad
    return mini
```

Ejemplo: Uso de las clases

```
def main():
    """
    Programa que ilustra el uso de una lista de personas
    """

    # Creamos la lista vacía y luego añadimos varias personas
    lista: ListaPersonas = ListaPersonas()
    lista.inserta(Persona("pepe", 20))
    lista.inserta(Persona("ana", 19))
    lista.inserta(Persona("luis", 21))
    lista.inserta(Persona("laura", 20))
    lista.inserta(Persona("maría", 19))

    # Trabajar con la lista
    print(lista.a_texto())
    minima: int = lista.get_edad_minima()
    print(f"Edad mínima: {minima}")
```

Recorrido con índices

En ocasiones interesa recorrer una tabla usando los índices de sus casillas

- por ejemplo, para trabajar con los índices y los valores

Implementación del recorrido con índices en Python

```
for i, val in enumerate(tabla):
    #trabajar con i (índice) y val (valor)
```

Ejemplo: mostrar en pantalla el índice y valor de una lista de nombres

```
nombres: list[str] = ["Pepe", "Juan", "Ana",
                      "Laura"]
for i, nom in enumerate(nombres):
    print(f"{i}- {nom}")
```

Tablas paralelas

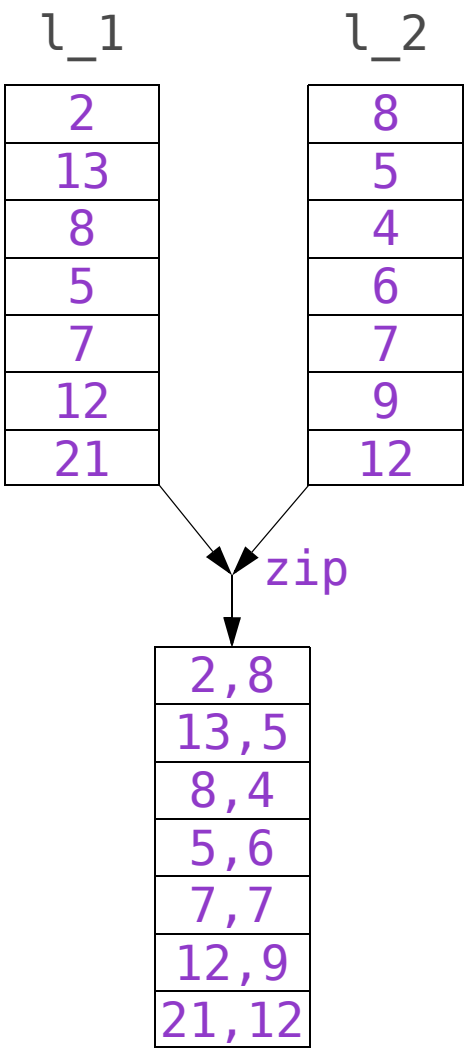
En ocasiones tenemos dos tablas del mismo tamaño con datos relacionados

- El dibujo muestra un ejemplo con dos tablas con las coordenadas *X* e *Y* de unos puntos

La función predefinida `zip()` permite hacer un iterador en el que se unen dos (o más) listas en una secuencia de tuplas, a modo de *cremallera*

- es muy eficiente, pues no crea en memoria una nueva lista; aprovecha el espacio de memoria de las listas ya existentes
- habitualmente se usa para un recorrido

```
for val1, val2 in zip(l_1, l_2):
    print(val1, val2)
```



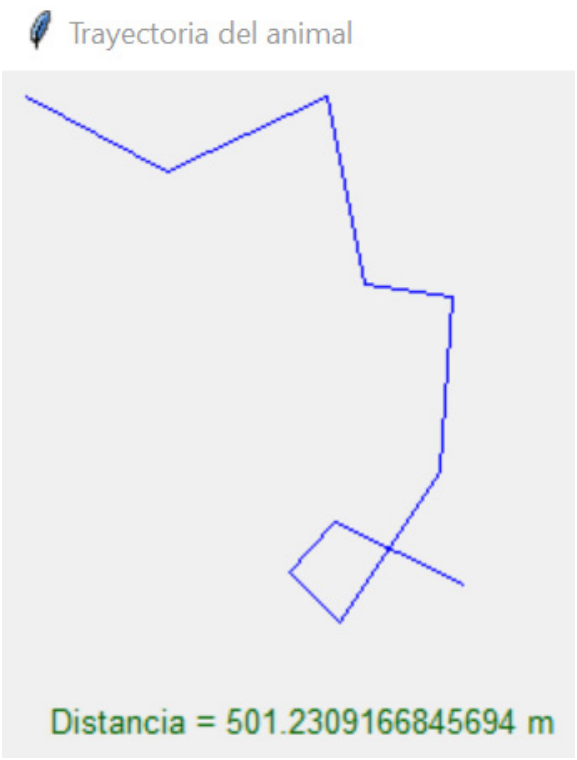
Ejemplo con tablas paralelas

Disponemos de dos listas con las coordenadas x e y de un recorrido de un animal en un recinto

Queremos calcular la distancia recorrida. Sumaremos la distancia de cada tramo, desde el punto i-1 al i, calculada así

$$dist_{i-1,i} = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Siendo (x_i, y_i) las coordenadas del punto i



Necesitamos un recorrido parcial

- cada tramo va de un punto al siguiente
- hay un tramo menos que el número de puntos

Ejemplo

```
# -*- coding: utf-8 -*-
"""
Ejemplo de cálculo de distancia de una trayectoria de un animal

@author: Michael
@date  : mar 2023
"""

import math

class Trayectoria:
    """
    La clase trayectoria contiene la trayectoria de un animal

    Atributos:
        __coord_x : lista de las coordenadas x de los puntos de
                    la trayectoria, en m
        __coord_y : lista de las coordenadas y de los puntos de
                    la trayectoria, en m
    """
```

Ejemplo (cont.)

```
    def __init__(self, coord_x: list[float],
                 coord_y: list[float]):
        """
        Constructor que copia en los atributos las listas que
        se pasan como parámetros

        Args:
            coord_x: lista de coordenadas x, en m
            coord_y: lista de coordenadas y, en m
        """

        # con list(x) creamos una copia de x
        # los atributos serán copias de las listas originales
        self.__coord_x: list[float] = list(coord_x)
        self.__coord_y: list[float] = list(coord_y)
```

Ejemplo (cont.)

```
    def distancia(self) -> float:
        """
        Calcula la distancia recorrida por el animal, en m

        Returns:
            La distancia recorrida, en m
        """
        # Aquí anotamos las coordenadas del punto anterior
        c_x_ant: float = self.__coord_x[0]
        c_y_ant: float = self.__coord_y[0]
        total: float = 0
        # Recorremos todas las parejas de puntos
        # excepto la primera
        for c_x, c_y in zip(self.__coord_x[1:],
                            self.__coord_y[1:]):
            total += math.sqrt((c_x-c_x_ant)**2+
                               (c_y-c_y_ant)**2)
            # Refrescamos el punto anterior
            c_x_ant = c_x
            c_y_ant = c_y
        return total
```

Ejemplo: programa de prueba

```
def main():
    """
    Programa que prueba la clase trayectoria
    """

    tray = Trayectoria(
        [10, 66, 130, 145, 180, 175, 135, 115, 133, 185],
        [10, 40, 10, 85, 90, 160, 220, 200, 180, 205])

    dis: float = tray.distancia()
    print(f"Distancia = {dis} m")
```

5.3. Algoritmos de búsqueda en tablas

La búsqueda del primer elemento que cumple una determinada propiedad es otro algoritmo muy habitual en tablas

- Ejemplo: ¿Existe en una lista un elemento mayor que 100?

Se parece al recorrido

- pero cuando encontramos el elemento buscado *cesamos el recorrido*
- Hay que prever el caso de que *no encontremos* lo buscado

En Python la búsqueda se suele basar en el *filtrado*

- El filtrado de una tabla consiste en obtener otra tabla con *todos* los elementos que cumplen una propiedad
- Luego bastará obtener el *primer* elemento de la tabla filtrada, teniendo en cuenta la posibilidad de que no haya ninguno

Filtrar una lista

Deseamos obtener una lista conteniendo los elementos de otra que cumplen una condición

- Es una operación relacionada con la búsqueda, pero diferente

Para hacer un filtrado, Python dispone de comprensiones de listas (list comprehensions)

```
cumplen = [x for x in lista if condición]
```

- Donde condición es una expresión booleana sobre el valor `x`

Ejemplo: obtener todos los elementos mayores que 30

```
mayores_que_30 = [x for x in lista if x > 30]
```

Filtrar una lista (cont.)

Similarmente se puede hacer con una expresión generadora con condiciones

```
generador_cumplen = (x for x in lista if condición)
```

- Donde condición es una expresión booleana sobre el valor x
- observar que la expresión generadora va entre ( )

Un generador es un objeto capaz de generar sobre la marcha una secuencia de objetos, sin que tenga que estar almacenada en memoria

- es muy eficiente

Al igual que los iteradores, se puede usar para un recorrido

Ejemplo: obtener los elementos mayores que 30

```
gen_mayores_que_30 = (x for x in lista if x > 30)
```

Esquemas de búsqueda en Python

Python dispone de varias utilidades para facilitar la búsqueda en listas, basadas en el filtrado con una expresión generadora

Búsqueda del primer elemento que cumple una condición

```
elemento = next((x for x in lista if condición), None)
```

- donde condición es una expresión booleana que usa el elemento `x`
- si el elemento no se encuentra se retorna `None`

Si nos interesa obtener el índice en lugar del elemento:

```
indice = next((i for i, x in enumerate(lista) if ...), -1)
```

- si el elemento no se encuentra se obtiene `-1`

Ejemplo: Buscar el primer elemento >30 y su índice

```
elemento = next((x for x in lista if x > 30), None)
indice = next((i for i, x in enumerate(lista) if x > 30), -1)
```

Ejemplo de búsqueda

Añadimos a la clase `ListaPersonas` una operación para buscar una persona en la tabla de personas

- La búsqueda es por el nombre

```
def busca_por_nombre(
        self, nombre: str) -> Persona | None:
    """
    Retorna la primera persona cuyo nombre es igual
    a nombre, o None si no la encuentra
    """
    return next((perso for perso in self.__lista
                 if perso.get_nombre() == nombre), None)
```

- La notación `Persona | None` indica que el valor de retorno puede ser indistintamente un objeto de la clase Persona o el valor None
- Versiones de Python anteriores usaban: Optional[Persona]

Ejemplo, uso del método de búsqueda

Desde el `main` u otra función:

```
    # Obtener la edad de luis

    # Primero buscamos a luis en la lista
    perso: Persona | None = lista.busca_por_nombre("luis")

    # Según el resultado de la búsqueda, actuamos
    if perso is None:
        print("Luis no se encuentra")
    else:
        edad: int = perso.get_edad()
        print(f"Edad de Luis {edad}")
```

5.4. Conjuntos

Un conjunto es una colección de elementos sin secuencia definida y no repetidos

Se forma poniendo los elementos entre {}. Ejemplo

```
equipo = {'pedro', 'ana', 'carla', 'andrés'}
```

Comprobar si un elemento pertenece al conjunto:

```
'carla' in equipo # True
```

Diferencia de conjuntos: -

```
elegidos = equipo - {'andrés'} # pedro, ana y carla
```

Unión de conjuntos: |

```
especial = elegidos | {'gelo'} # pedro, ana, carla
                               # y gelo
```

Conjuntos (cont.)

Intersección de conjuntos: &

```
comunes = equipo & especial # pedro, ana, carla
```

Crear un conjunto vacío

```
v = set()
```

Obtener una lista a partir de un conjunto

```
l = list(v)
```

Obtener un conjunto a partir de una lista

```
s = set(l)
```

Ejemplo con números aleatorios

En este ejemplo corregimos el programa del capítulo 2 que produce una apuesta aleatoria para la bonoloto

- son seis números *no repetidos* del 1 al 49

```
# -*- coding: utf-8 -*-
"""
Producimos una apuesta aleatoria para la bonoloto

Son seis números aleatorios del 1 al 49
Eliminamos repeticiones con un conjunto

@author: Michael
@date  : ene 2023
"""

import random
from typing import Set
```

Ejemplo (cont.)

```
def main():
    """
    Muestra en pantalla seis números aleatorios entre 1 y 49
    """

    # Establecemos una semilla aleatoria
    random.seed()

    # Creamos el conjunto vacío
    conjunto: Set[int] = set()

    #Añadimos a la secuencia hasta que su tamaño sea 6
    while len(conjunto) < 6:
        # añadimos al conjunto un nuevo número aleatorio
        conjunto = conjunto | {random.randint(1, 49)}


    # Mostramos el conjunto
    print(conjunto)
```

5.5 Tablas multidimensionales

Las tablas multidimensionales como matrices o tensores se pueden construir con listas de listas (es decir, listas anidadas)

	0	1	2
0	'a'	'b'	'c'
1	1	2	3

Ejemplo de creación de una lista heterogénea

```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n] # x vale [['a', 'b', 'c'], [1, 2, 3]]
```

Ejemplo de uso de una fila

```
x[0] vale ['a', 'b', 'c']
```

Ejemplo de uso de un elemento individual

```
x[0][1] vale 'b'
```

Matrices

Organizada como una lista de filas, y cada fila una lista de números. Ejemplo:

```
matriz = [
     [1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12],
]
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Uso de un elemento

```
matriz[2][3] vale 12
```

Sin embargo, el uso de elementos individuales de matrices muy grandes es ineficiente

- Para manipular vectores y matrices se recomienda usar el paquete `numpy`, que es una librería muy eficiente por estar hecha en C

5.6. El paquete numpy

Es una librería muy eficiente para manejar vectores, matrices y tablas multidimensionales con números

Su elemento central es el *array*

- es un objeto de la clase `numpy.ndarray`
- en su manejo sencillo es como una lista o matriz de Python
- dispone de numerosas operaciones que lo hacen cómodo y eficiente

Es habitual importar `numpy` dándole el nombre abreviado `np`:

```
import numpy as np
```

Crear arrays

Usar la función `np.array()` y pasarle los datos

`arr = np.array([1, 2, 3, 4, 5])`	`[1 2 3 4 5]`
`my_array = np.array([[1, 2, 3, 4],` `[5, 6, 7, 8]])`	`[[1 2 3 4]` ` [5 6 7 8]]`

O crear un array relleno de unos, o ceros

`arr_1 = np.ones((7))`	`[1. 1. 1. 1. 1. 1. 1.]`
`arr_0 = np.zeros((2, 3))`	`[[0. 0. 0.]` ` [0. 0. 0.]]`

Array a partir de un rango entero, o números reales distribuidos uniformemente

`arr_range = np.arange(10, 25, 2)`	`[10 12 14 16 18 20 22 24]`
`arr_lin = np.linspace(0, 2, 9)`	`[0. 0.25 0.5 0.75 1.` ` 1.25 1.5 1.75 2.]`

Inspeccionar arrays y cambiar su forma

Podemos mostrar y cambiar la forma del array (número de filas, columnas, ...) con `shape`:

`print(arr.shape)`	`(5,)`

`arr.shape = (5, 1)`	`[[1]`
	` [2]`
	` [3]`
	` [4]`
	` [5]]`

Podemos mostrarlos con `print()`

```
print(arr_lin)
```

Se puede iterar con ellos

```
for x in arr:
    # usar x
```

Inspeccionar arrays (cont.)

Se puede obtener un elemento con dos índices entre `[]`, como con las listas, o poniendo una *tupla* de índices

`print(arr_0[1][2])` o `print(arr_0[1,2])`	`0.0`
`arr_lin[0] = 3`	`[3. 0.25 0.5 0.75 1. 1.25 1.5 1.75 2.]`

Se puede obtener la longitud (número de filas) con

`print(len(arr_1))`	`9`
`print(len(arr_0))`	`2`

Se pueden usar rodajas, pero no encoger o estirar el tamaño

`arr_1[1:3] = [2., 3.]  # solo dos elementos`	`[1. 2. 3. 1. 1. 1. 1.]`
`arr_0[:,2] = [4., 5.]  # 2ª columna`	`[[0. 0. 4.] [0. 0. 5.]]`

Operaciones con arrays

Se admiten las operaciones aritméticas habituales con vectores y matrices: suma (+), resta (-), producto elemento a elemento (*), producto escalar o matricial (@), producto vectorial (`np.cross`)

- obviamente solo si los tamaños son compatibles

`a_1 = np.array([[1, 2], [0, 1]])`	`[[1 2]` ` [0 1]]`
`a_2 = np.array([[4, 5], [7, 8]])`	`[[4 5]` ` [7 8]]`
`a_3 = a_1 + a_2`	`[[5 7]` ` [7 9]]`
`a_4 = a_1 @ a_2`	`[[18 21]` ` [ 7  8]]`

`v_1 = np.array([1, 2, 3])`
`v_2 = np.array([-1, 1, 2])`
`v_3 = np.cross(v_1, v_2)`
	`[1, 2, 3]`
	`[-1, 1, 2]`
	`[ 1, -5, 3]`

Operaciones con arrays (cont.)

Operaciones con escalares, operando sobre todos los elementos: suma, resta, producto, división, elevar a

`a_5 = a_1 + 10.0`	`[[11. 12.]` ` [10. 11.]]`
`a_6 = a_5 * 2.0`	`[[22. 24.]` ` [20. 22.]]`

Funciones matemáticas

Es posible aplicar funciones matemáticas similares a las del paquete `math`, tales como logaritmos, trigonométricas, etc.

Se aplican elemento a elemento

```
np.sin(np.array((0., 30., 45., 60., 90.))*
       np.pi / 180.)
```

```
[0.         0.5
 0.70710678 0.8660254 1.]
```

También las funciones como `sum`, `max`, `mean`, `std`, ... que actúan sobre todos los elementos

```
np.mean(arr) # promedio
```

```
3.0
```

Operaciones con matrices

Matriz inversa, transpuesta, determinante, identidad

`arr = np.array([[1, 2], [3, 4]])`	`[[1 2]` ` [3 4]]`
`inverse = np.linalg.inv(arr)`	`[[-2.   1. ]` ` [ 1.5 -0.5]]`
`transpuesta = arr.transpose()`	`[[1 3]` ` [2 4]]`
`np.linalg.det(arr)`	`-2.`
`id = np.identity(2)`	`[[1. 0.]` ` [0. 1.]]`

Ejemplo con numpy

Vamos a trabajar con una imagen como un array de numpy con filas y columnas formadas por píxeles

- leer la imagen y pintarla
- darle la vuelta de arriba a abajo y pintarla
- darle la vuelta de izquierda a derecha y pintarla
- mostrar un fragmento

```
# -*- coding: utf-8 -*-
"""
Ejemplo de tratamiento de imágenes con numpy
@author: Michael
@date:   mar 2023
"""

import numpy as np
import imageio                          # para leer la imagen
import matplotlib.pyplot as plt         # para mostrar la imagen
```

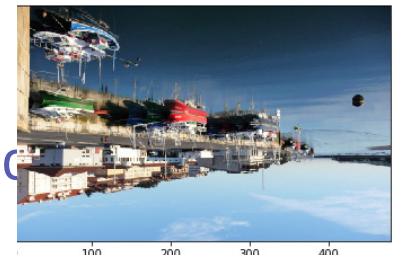
Ejemplo (cont.)

```
def main():
    """
    Prueba con una foto de San Vicente
    """
    # Leer y mostrar la imagen
    img = imageio.imread('san-vicente-small.jpg')
    plt.imshow(img)
    plt.show()

    # Mostrar la imagen invertida de arriba a abajo
    plt.imshow(img[::-1, :])
    plt.show()

    # Mostrar la imagen en espejo
    plt.imshow(img[:, ::-1])
    plt.show()


    # Mostrar un fragmento de la imagen
    plt.imshow(img[130:200, 150:280])
    plt.show()
```



Apéndices

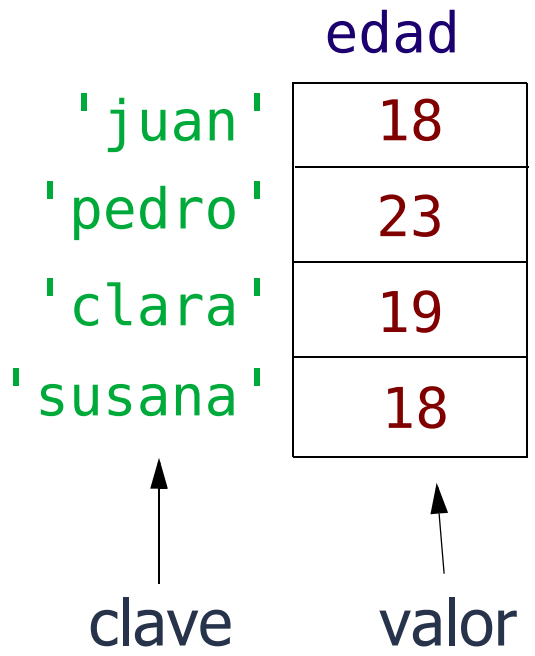
- A.1 Diccionarios
- A.2 Tipos enumerados

A.1. Diccionarios

El diccionario sigue el modelo de una lista en la que el índice no es un número entero, sino un texto u otro tipo de dato inmutable llamado *clave*

Creación usando `{}` y parejas de clave/valor. Las claves no se pueden repetir. Ejemplo

```
edad = {'juan': 18, 'pedro': 23,
        'clara': 19, 'susana': 18}
```



Elemento de un diccionario:

```
edad['pedro'] # vale 23
```

Pertenencia:

```
'ana' in edad # vale False
```

Recorrido de un diccionario

Para recorrer un diccionario se usa el método `items()`, que nos da una secuencia de sus parejas clave-valor.

Si hacemos el recorrido con un bucle `for`, se usa una tupla de variables de control, para las parejas clave-valor. Ejemplo:

```
for key, val in edad.items():
    print(f"clave={key}, valor={val}")
```

Ejemplo con diccionarios

```
# -*- coding: utf-8 -*-
"""
Este módulo define operaciones para trabajar con el código Morse

Define un diccionario de código morse para pasar con comodidad
letras individuales a código Morse, así como una función para
convertir textos completos

@author: Michael
@date:   mar 2023
"""

# Diccionario de código Morse
MORSE = {
    "A" : ".-", "B" : "-...", "C" : "-.-.", "D" : "-..",
    "E" : ".", "F" : "..-.", "G" : "--.", "H" : "....",
    "I" : "..", "J" : ".---", "K" : "-.-", "L" : ".-..",
    "M" : "--", "N" : "-.", "O" : "---", "P" : ".--.",
    "Q" : "--.-", "R" : ".-.", "S" : "...", "T" : "-",
```

Ejemplo con diccionarios (cont.)

```
    "U" : "..-", "V" : "...-", "W" : ".--", "X" : "-..-",
    "Y" : "-.--", "Z" : "--..", "0" : "-----", "1" : ".----",
    "2" : "..---", "3" : "...--", "4" : "....-", "5" : ".....",
    "6" : "-....", "7" : "--...", "8" : "---..", "9" : "----.",
    "." : ".-.-.-", "," : "--..--"
}

def pasar_a_morse(texto: str) -> str:
    """
    Devuelve un string con el texto convertido a código morse

    Se utiliza un formato en el que aparece cada palabra
    seguida de su código morse

    Args:
        texto: El texto a convertir
    Returns:
        El código morse
    """
```

Ejemplo con diccionarios (cont.)

```
    # Partir el texto inicial en palabras
    palabras: list[str] = texto.strip().split(" ")
    # Variable para ir recogiendo el resultado
    resul: str = ""

    # Recorremos todas las palabras
    for palabra in palabras:
        # Solo nos interesan las palabras no vacías
        if palabra != "":
            # Variable para ir recogiendo el código morse
            palabra_morse: str = ""
            # Recorremos cada carácter de la palabra
            for carac in palabra:
                # Añadimos el carácter Morse y unos espacios
                palabra_morse += MORSE[carac.upper()]+"   "
            # Añadimos la palabra y la palabra morse
            resul += palabra+":  "+palabra_morse+"\n"
    return resul
```

A.2 Tipos enumerados

Un tipo enumerado es aquel en el que los posibles valores son un conjunto finito de palabras

- por ejemplo un color (`ROJO`, `VERDE`, `AZUL`)
- o un día de la semana (`LUNES`, `MARTES`, ...)

En lenguajes sin enumerados se usan constantes enteras

- pero esta solución puede dar errores si se usan enteros no previstos
- además, los tipos enumerados tienen facilidades para trabajar con sus valores

Declaración de clases enumeradas

Declarar una clase enumerada:

```
from enum import Enum

class Color(Enum):
    """
    Describe los colores permitidos
    """
    ROJO = 1
    VERDE = 2
    AZUL = 3
```

Elementos de una clase enumerada

- Constantes:
  ```
  Color.ROJO
  ```
- Un valor enumerado se puede convertir directamente a texto
  ```
  col = Color.VERDE
  print(col)
  ```
- u obtener su nombre mediante su atributo `name`
  ```
  print(col.name)
  ```
- Se puede obtener un valor enumerado a partir del valor numérico
  ```
  mi_color = Color(3)  # Es AZUL
  ```
- o del nombre expresado como string:
  ```
  otro_color = Color["VERDE"]
  ```
- Se pueden comparar para igualdad u orden
- Se puede iterar sobre la clase enumerada para obtener sucesivamente todos sus valores, tal como muestra el ejemplo

Ejemplo

```
from enum import Enum

class DiaSemana(Enum):
    """
    Describe los días de la semana
    """
    LUNES = 1
    MARTES = 2
    MIERCOLES = 3
    JUEVES = 4
    VIERNES = 5
    SABADO = 6
    DOMINGO = 7

dia_1 = DiaSemana.LUNES       # constante
dia_2 = DiaSemana(2)          # obtener a partir del valor
dia_3 = DiaSemana["SABADO"]   # obtener a partir del nombre
print(dia_1)
print(dia_2.name)
print(dia_3)
```

Ejemplo (cont.)

```
# Iterar sobre todos los valores

for dia in DiaSemana:
    print(dia.name)
```