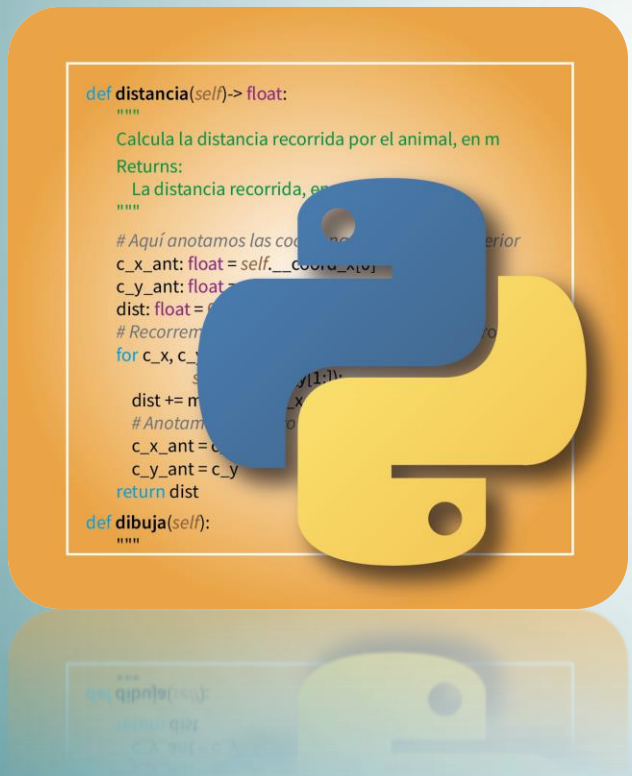
Programación

5. Estructuras de datos



Michael González Harbour

José Javier Gutiérrez García

José Carlos Palencia Gutiérrez

José Ignacio Espeso Martínez

Adolfo Garandal Martín

Departamento de Ingeniería Informática y Electrónica

Programación en Python

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

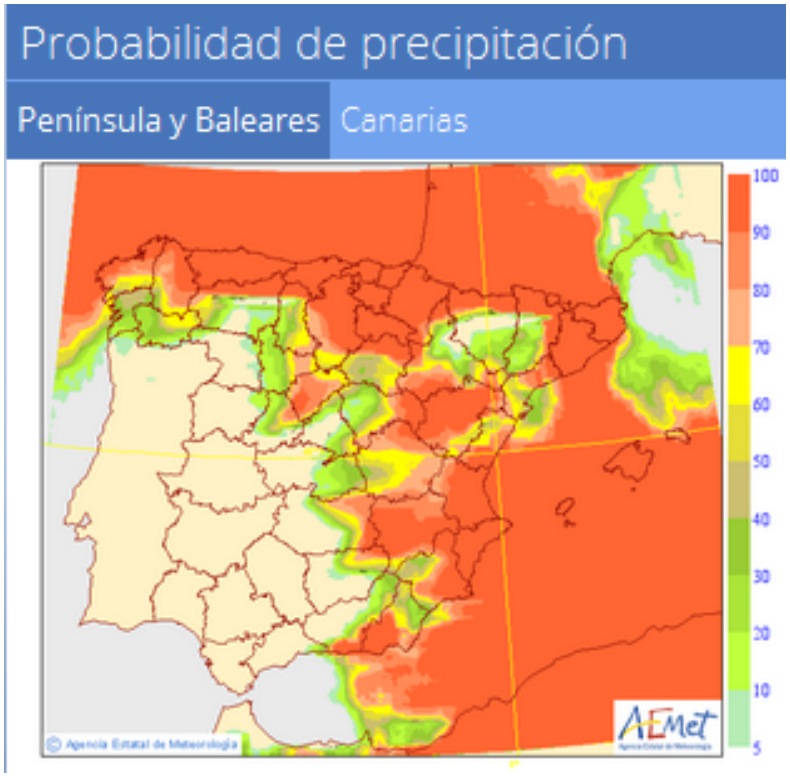
5.1 Tablas

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

Notas:

Aunque conocemos la forma de guardar datos en variables individuales, cuando queramos manejar centenares, miles o incluso millones de datos no es práctico hacerlo en variables individuales y manejarlas por separado

Por ello todos los lenguajes de programación disponen de estructuras de datos para manejar tablas de datos. Por ejemplo listas, matrices, etc. Estas estructuras se suelen llamar arrays o arreglos. En Python las estructuras básicas para manejar tablas de datos son las tuplas y las listas

En el primer ejemplo que se muestra arriba se guardan datos de un conjunto de estaciones de esquí. Cada estación tiene datos como su nombre, el estado, los kilómetros abiertos, el espesor de nieve y un icono de la meteorología actual. Los datos de una estación podrían guardarse en una tupla, o podrían ser atributos de un objeto. Posteriormente, como tenemos muchas estaciones, podríamos crear una lista de estaciones

En el segundo ejemplo se muestra una imagen meteorológica. Cualquier imagen, al digitalizarse, se almacena como una tabla bidimensional de píxeles, o puntos de la imagen. Cada punto de una imagen en color se suele representar por una terna de intensidades de los colores fundamentales, por ejemplo rojo, verde y azul. Una imagen sencilla, por ejemplo una fotografía de un móvil, puede tener millones de estos píxeles (megapíxeles)

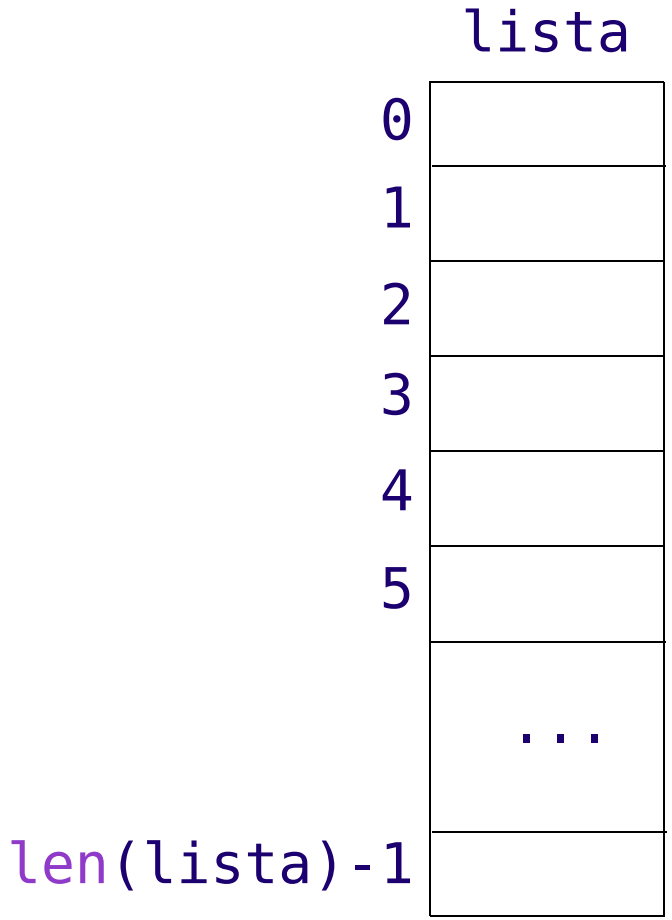
Ambos ejemplos nos muestran la necesidad de disponer de estructuras de datos para manejar tablas de datos

En capítulos anteriores hemos hablado ya de las tuplas y listas y en este capítulo profundizaremos en ellas y veremos algunas otras estructuras de datos

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, 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):`

Notas:

La diferencia fundamental entre la lista y la tupla es su propiedad de ser mutable o inmutable

Son secuencias (al igual que los strings y los rangos) y por tanto tienen definida la operación de concatenación, que forma una secuencia nueva conteniendo los elementos de la primera secuencia seguidos de los de la segunda

- esta operación se representa con el símbolo `"+"`

Existen operaciones para invertir una secuencia (*reverse*) o para ordenar sus elementos (*sort*), siempre que tengan definida una relación de orden

La tabla muestra que hay dos formas de realizar estas operaciones:

- `sort()` y `reverse()`: se aplican a listas, que son mutables, y modifican la propia lista sin crear una nueva
- `sorted()` y `reversed()`: obtienen un iterador, que es un objeto que permite recorrer la secuencia original en otro orden; son muy eficientes en términos de memoria porque no crean una nueva secuencia; se pueden usar para hacer un recorrido (por ejemplo en un bucle `for`) o para crear una nueva secuencia, haciendo un cambio de tipo, por ejemplo con `list()` o `tuple()`

  - como no modifican la secuencia original se pueden aplicar a cualquier secuencia, tanto mutable como inmutable

Notas:

Ejemplos con `sorted`

Código	Resultado

`lista: List[str] =["uno", "dos", "tres"]`
`for num in sorted(lista):`
`    print(num)`
	dos
	tres
	uno

`lista: List[str]=["uno", "dos", "tres"]`
`ordenada = tuple(sorted(lista))`
`print(ordenada)`
	`('dos', 'tres', 'uno')`

Listas y tuplas (cont.)

Recordamos también otros usos de las secuencias

- Elemento individual:

```
lista[i]
```

- 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

Notas:

Los métodos `append()` y `pop()`, aplicados sobre una lista, permiten respectivamente añadir un elemento al final y quitar un elemento del final de una lista

Ejemplo

```
# Crea la lista vacía
lista: List[str]  = []
# Añade tres elementos
lista.append("uno")
lista.append("dos")
lista.append("tres")
# Muestra la lista. Se obtiene ['uno', 'dos', 'tres']
print(lista)

# Saca los elementos uno por uno desde el final, mientras haya elementos,
# mostrándolos. Se obtiene sucesivamente: tres, luego dos y finalmente uno
while len(lista) > 0:
    num = lista.pop()
    print(num)
# Muestra la lista. Como está vacía se obtiene []
print(lista)
```

Para trabajar con listas que se modifican por el principio se recomienda usar `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
```

Notas:

El recorrido se muestra mediante plantillas de pseudocódigo y código Python

En la plantilla se debe sustituir la parte marcada en amarillo por el código que se desea ejecutar sobre cada elemento de la tabla

Ejemplo de recorrido

```
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")
```

Notas:

En este ejemplo se hace un recorrido de una lista de alimentos, mostrándolos en pantalla

- Sin embargo, si nos encontramos uno que no nos gusta (el `"apio"` en este caso) finalizamos el recorrido anticipadamente

Puede observarse que la variable `alimento` va tomando uno por uno los sucesivos valores almacenados en la lista `comida`

- en este caso `"queso"`, `"jamón"`, ...

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
```

Notas:

Ejemplos de recorrido inverso y parcial

Código	Resultado

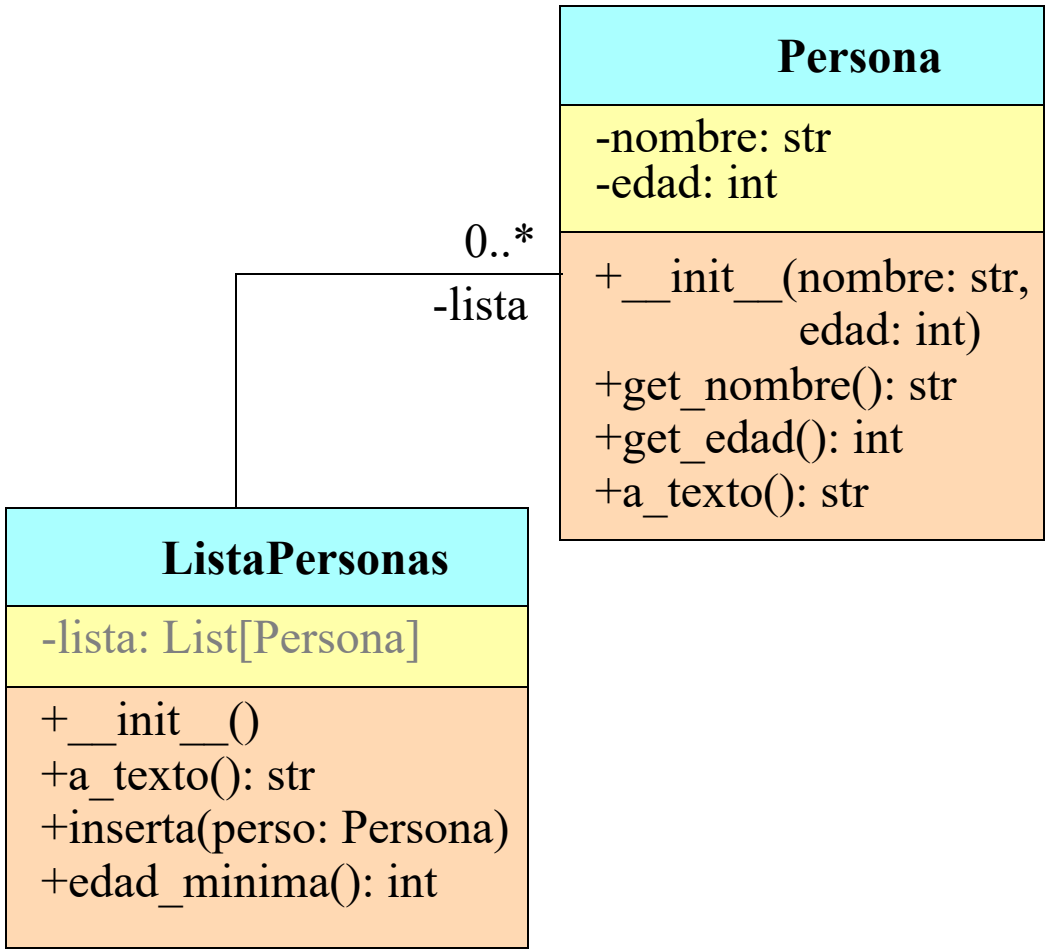
`dias_semana: List[str] = ["lunes", "martes", "miércoles",`
`  "jueves", "viernes", "sábado",`
`  "domingo"]`
 	
`for dia in reversed(dias_semana):`
`  print(dia)`
	domingo
	sábado
	viernes
	jueves
	miércoles
	martes
	lunes

`for dia_laborable in dias_semana[0:5]:` `  print(dia_laborable)`	lunes martes miércoles jueves viernes

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



Notas:

En el diagrama de clases que se muestra arriba, y que usa notación UML (https://creately.com/blog/diagrams/uml-diagram-types-examples/), se indica que la clase `ListaPersonas` contiene un atributo privado llamado `lista` que es una lista de 0 a más objetos de la clase `Persona`

Hay dos formas alternativas de indicar esto. Hay que usar una de las dos, pero en el diagrama de arriba se muestran ambas a modo ilustrativo:

1. En la línea que representa la asociación entre las clases `ListaPersonas` y `Persona`. Se pone `0..*` que significa de cero a más, y en este caso indica que una lista de personas puede tener entre 0 y más objetos de la clase `Persona`. El nombre `-lista` significa esta asociación es un atributo privado de la clase `ListaPersonas`, llamado `lista`, que es el que contiene de 0 a más personas
2. En el recuadro de atributos de la clase `ListaPersonas`. Con el texto `-lista: List[Persona]`

Como hemos dicho, en este caso se muestra de las dos formas, pero en la práctica hay que usar solo una. El color gris de la forma 2) trata de recordarnos esto.

Notas:

En el capítulo 4 vimos un algoritmo de cálculo de máximo (el de mínimo es simétrico). Ahora vemos uno ligeramente diferente. Ponemos aquí los dos para observar las diferencias

Con recorrido completo	Con recorrido parcial

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

En la versión con recorrido completo inicializamos la variable mínimo al mayor valor posible (infinito si es un número)

En la versión con recorrido parcial inicializamos la variable mínimo al primer elemento de la lista (`lista[0]` o a un valor asociado a ese elemento) y luego recorremos todos los elementos de la lista excepto el primero (`lista[1:]`), puesto que el primer elemento ya ha sido tenido en cuenta

- este algoritmo solo funciona si en la lista hay al menos un elemento
- por tanto, si la lista puede estar vacía se debe elegir el primer algoritmo

Ejemplo (cont.)

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

@author: Michael
@date  : mar 2019
"""

from typing import List

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 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.edad_minima()
    print(f"Edad mínima: {minima}")
```

Notas:

En el ejemplo, el algoritmo de mínimo se implementa en el método `edad_minima()`

Se ha elegido el algoritmo con recorrido parcial

- puede verse que el valor inicial es la edad del primer elemento de la lista `lista[0].get_edad()`

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}")
```

Notas:

La operación `enumerate(lista)` permite obtener un iterador que, al recorrerlo, nos va dando tuplas de índice y valor asociado en la lista

El ejemplo de arriba muestra cómo utilizar esta tupla de índice y valor en un recorrido con bucle **`for`**

En este otro ejemplo escribimos un algoritmo que nos da respuesta a la siguiente pregunta: ¿en qué casilla de la lista `nombres` está `"Ana"`?

```
casilla_de_ana: int = -1
for i, nom in enumerate(nombres):
    if nom == "Ana":
        casilla_de_ana = i
        break
# se muestra el resultado; -1 si Ana no está en la lista
print(f"Casilla de Ana: {casilla_de_ana}")
```

Observar que al encontrar lo que buscábamos cesamos el recorrido saliendo del bucle con un **`break`**

Observar también que un bucle normal de recorrido de la lista (sin `enumerate`) no nos podría dar directamente esta respuesta, a no ser que implementásemos un contador de casillas

Tablas paralelas

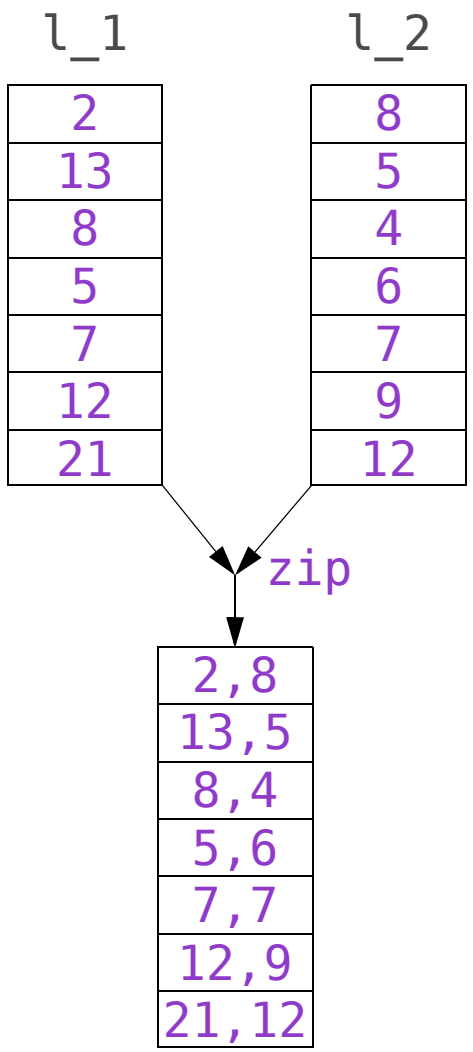
En ocasiones tenemos dos tablas del mismo tamaño con datos relacionados

- 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)
```



Notas:

Otro uso de la operación `zip()` si no se desea hacer un recorrido es para unir varias listas y obtener una nueva lista de tuplas. La tupla i-ésima contendrá las casillas i-ésimas de cada lista original

- la conversión del iterador a lista se hace con la conversión de tipo, `list()`

Ejemplo: convertir tres listas a una lista de tuplas a modo de "cremallera":

```
nombres: List[str] = ["Pepe", "Juan", "Ana"]
comidas_preferidas: List[str] = ["paella", "garbanzos", "pizza"]
musica_preferida: List[str] = ["rock", "electrónica", "hip hop"]

preferencias: List[Tuple[str, str, str]] = \
    list(zip(nombres, comidas_preferidas, musica_preferida))
print(preferencias)
```

El resultado obtenido es:

```
[('Pepe', 'paella', 'rock'),
 ('Juan', 'garbanzos', 'electrónica'),
 ('Ana', 'pizza', 'hip hop')]
```

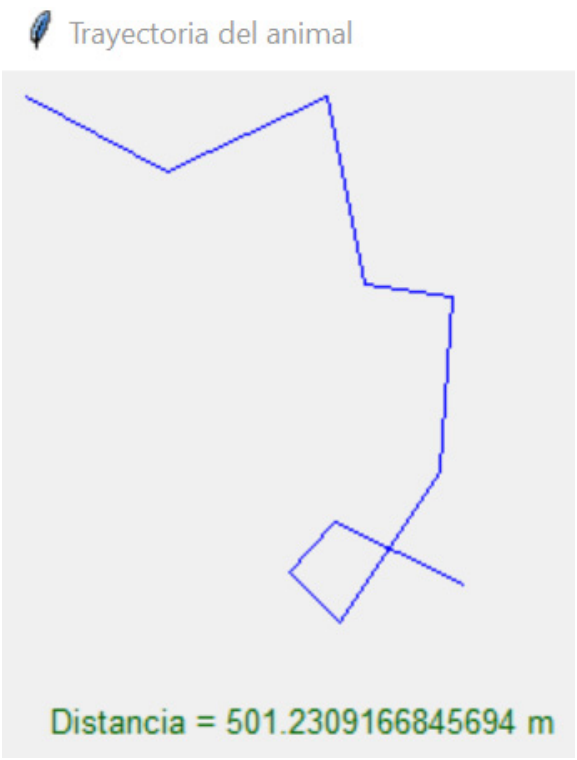
Ejemplo con tablas paralelas

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

Notas:

El algoritmo que nos piden es un sumatorio, siendo n el número de puntos. Tenemos siempre en cuenta que las casillas de la lista se numeran desde 0 hasta n-1. En este caso empezamos en la casilla 1:

$$total = \sum_{i=1}^{n-1} dist_{i-1,\,i}$$

Como para calcular la distancia $dist_{i-1,\,i}$ necesitamos las coordenadas x e y del punto i-1 y del punto i, haremos un bucle recorriendo las tablas paralelas con las coordenadas x e y

- además siempre guardaremos las coordenadas del punto i-1 (que es el anterior al i) en sendas variables llamadas `c_x_ant` y `c_y_ant`
- estas variables se inicializan a las coordenadas x e y del primer punto

El algoritmo queda así:

```
c_x_ant, c_y_ant = coord_x[0], coord_y[0]
total = 0
para cada c_x, c_y en coord_x, coord_y desde la 2ª casilla
    total = total + sqrt((c_x-c_x_ant)**2+(c_y-c_y_ant)**2)
    # Refrescamos el punto anterior
    c_x_ant, c_y_ant = c_x, c_y
# El resultado está en la variable total
```

Ejemplo

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

@author: Michael
@date  : mar 2019
"""

import math
from typing import List

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

Notas:

Hay que destacar las dos diferencias entre la búsqueda y el recorrido:

- cesar la búsqueda al encontrar lo buscado
- tener en cuenta el caso de no encontrar lo buscado

En relación a la evaluación: si se pide un algoritmo de búsqueda y no se tienen en cuenta estas diferencias se considerará una mala implementación, incluso aunque funcione o funcione parcialmente

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]
```

Notas:

Vimos las comprensiones de listas en el capítulo 4

Aquí les añadimos condiciones, escritas tras el bucle `for`

- la comprensión de listas incluye solo los elementos que cumplen la condición expresada en el `if`

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)
```

Notas:

Las expresiones generadoras son similares a las comprensiones de listas, pero producen un generador, no una lista

El generador no guarda elementos en memoria. Por ello es eficiente, pero requiere usarlo para hacer un recorrido o usar una función capaz de trabajar con la secuencia, como `sum()` en el caso de abajo

Por ejemplo, calcular la suma de los cuadrados de los 100 primeros números naturales:

- mediante una comprensión de listas

```
suma_cuadrados = sum([i**2 for i in range(1,101)])
```

- mediante una expresión generadora

```
suma_cuadrados = sum((i**2 for i in range(1,101)))
```

Es mucho más eficiente la segunda versión, pues no se necesita crear la lista en memoria

Entonces, ¿por qué tenemos comprensiones de listas?

- porque a veces necesitamos crear una lista y guardar sus elementos para hacer otras cosas con ellos

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 retorna `-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)
```

Notas:

La operación `next()` obtiene el próximo elemento de un generador o iterador y, aunque tiene más usos, es ideal para implementar el algoritmo de búsqueda sobre una expresión generadora

Además, el segundo parámetro que le pasamos es el valor a retornar si no existe ningún elemento en el generador, es decir, si no hemos encontrado lo buscado

Es habitual usar estos valores para indicar que algo no se ha encontrado:

- None: indica "nada" es decir una referencia a ningún objeto
- `-1`, en caso de que busquemos un número positivo
- `math.nan`, en caso de que busquemos un número real

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)-> Optional[Persona]:
    """
    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)
```

- Se requiere importar la anotación de tipo Optional, que indica que el valor de retorno puede ser None

```
from typing import Optional
```

Notas:

En este ejemplo añadimos un nuevo método a la clase `ListaPersonas` vista en la página 26

El método busca una persona en la lista, por su nombre. Se debe buscar un objeto cuyo método `get_nombre()` retorne un valor igual al parámetro `nombre`

Se retorna `None` si no se encuentra

En la anotación del tipo retornado por el método `busca_por_nombre()` ponemos `Optional[Persona]`

- Esto debe entenderse como que el método, opcionalmente, puede retornar una persona o el valor `None`, es decir, nada

Ejemplo, uso del método de búsqueda

Desde el `main` u otra función:

```
# Obtener la edad de luis
perso = lista.busca_por_nombre("luis")
if perso is None:
    print("Luis no se encuentra")
else:
    edad: int = perso.get_edad()
    print(f"Edad de Luis {edad}")
```

Notas:

Observar que `busca_por_nombre()` retorna o bien una persona (en el objeto llamado `perso`) o nada (`None`)

La comparación habitual para ver si un objeto vale `None` es `"objeto is None"`, como puede verse en el ejemplo aplicado al objeto `perso`

Si se obtiene una persona, entonces le podemos aplicar cualquiera de los métodos de la clase `Persona` al objeto `perso`

- en este caso el método `get_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)
```

Notas:

El conjunto es una estructura donde podemos introducir datos sin ordenar. Podemos imaginarla inicialmente como una bolsa donde meter información. Pero a diferencia de una bolsa, no permite tener datos repetidos. Si intentamos meter un dato que ya está, la operación no hará nada

Los conjuntos son mutables (pueden cambiar), aunque los elementos contenidos en ellos deben ser inmutables

Si para las listas usamos los delimitadores `[]`, y para las tuplas `()`, en los conjuntos usamos `{}` (también se usará este delimitador para los diccionarios, que aparecen en el apéndice A.1)

Una operación frecuente en los conjuntos es la pertenencia de un elemento al conjunto, que se hace con el operador in, que ya hemos visto para las listas y tuplas

Los conjuntos tienen operaciones especiales tales como unión (juntar dos conjuntos formando un tercero), intersección (obtener un nuevo conjunto con los elementos comunes) y diferencia (obtener un nuevo conjunto que resulta de quitar de un conjunto los elementos comunes con otro)

Operación	Operador
Unión	|
Intersección	&
Diferencia	-

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 2019
"""

import random
```

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()

    #Obtenemos 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)
```

Notas:

En este ejemplo usamos la propiedad del conjunto de no poder tener elementos repetidos para meter en él números evitando estas repeticiones

En el ejemplo debemos obtener una apuesta para la bonoloto, formada por seis números no repetidos. Iremos metiendo números en el conjunto hasta que haya seis elementos. Cuando metamos un número repetido no ocurrirá nada, pero el bucle continúa porque solo se acaba cuando en el conjunto haya 6 elementos

- El número de elementos del conjunto se obtiene con la operación `len()`, que ya vimos para las listas, tuplas y strings
- Para añadir un elemento al conjunto usamos la unión de conjuntos. Existe también un método `add()` que añade un elemento a un conjunto. Si el elemento ya pertenece al conjunto, no hace nada. Ejemplo:

```
conjunto.add(random.randint(1, 49))
```

- Similarmente, existe el método `discard()` para eliminar un elemento del conjunto. Si no existe, no hace nada. Ejemplo:

```
conjunto.discard(37)
```

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'
```

Notas:

En Python, así como en muchos lenguajes de programación, las listas multidimensionales siguen el modelo de listas de listas

Podemos imaginar una lista de filas con datos, de modo que la columna i-ésima de una tabla bidimensional estaría formada por las casillas i-ésimas de cada fila. Por ejemplo, la columna de índice 1 de la tabla que se muestra arriba estaría formada por los elementos de índice 1 de cada fila, es decir `'b'` y 2

Del mismo modo se pueden crear tablas de más dimensiones

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

Notas:

Para la manipulación de matrices y tablas de más de 2 dimensiones que contengan números se recomienda el uso de la librería `numpy`, que veremos a continuación

- es mucho más eficiente

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
```

Notas:

El paquete `numpy` define una clase, `numpy.ndarray`, cuyos objetos llamaremos *arrays*, y nos servirán para hacer operaciones numéricas muy eficientes con vectores, matrices y tablas multidimensionales de números

No hay una forma estandarizada de hacer anotaciones de tipo para `numpy.ndarray`, por lo que en general no las pondremos, excepto para anotar el valor de retorno de una función. En este caso, si no se pone se puede dar la falsa impresión de que es una función que no retorna nada.

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.]`

Notas:

Para crear arrays no lo hacemos con el constructor de `ndarray`, sino que disponemos de diferentes funciones especializadas en la creación

- `np.array()`: crea arrays a partir de listas unidimensionales o multidimensionales
- `np.zeros()`: crea arrays rellenos de ceros
- `np.ones()`: igualmente, crea arrays rellenos de unos
- `np.arange()`: parecido a `range()`, crea un array conteniendo un rango de números enteros
- `np.linspace()`: similarmente al anterior, crea un array conteniendo una secuencia de números reales distribuidos uniformemente

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
```

Notas:

El atributo `shape` nos indica la forma de un array, es decir cuántas filas tiene, cuántas columnas, cuántas dimensiones

Este atributo es muy potente porque no solo permite consultar la forma sino cambiarla, manteniendo los mismos datos

- en el ejemplo de arriba hemos pasado de un array de una sola dimensión a otro de dos dimensiones, con 5 filas y una sola columna: forma `(5,1)`
- otro ejemplo: podemos pasar de un array de 2 filas y 3 columnas a otro de 3 filas y 2 columnas

Código	Resultado
`mat = np.array([[1, 2, 3], [4, 5, 6]])` `print(mat)`	`[[1 2 3]` ` [4 5 6]]`

`mat.shape = (3, 2)`
`print(mat)`
	`[[1 2]`
	` [3 4]`
	` [5 6]]`

El recorrido de las filas de un array de dos dimensiones nos muestra que cada fila es a su vez un array

Código	Resultado
`for fila in mat:` `    print(fila)`	`[1 2 3]` `[4 5 6]`

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.]]`

Notas:

Para obtener un elemento cualquiera puede hacerse con una tupla de índices. Dos en el caso de un array bidimensional, tres para uno tri-dimensional, etc.

La operación `len()` nos da el número de filas, o el número de elementos si es un array unidimensional

En las rodajas, usamos también la notación con una tupla de rangos separados por `","` como en estos ejemplos para un array bidimensional

Índices	Filas incluidas	Columnas incluidas
`[:, :]`	Todas	Todas
`[:, 2]`	Todas	2
`[:, 2:4]`	Todas	De la 2 a la 3[1]
`[1:, 3]`	De la 1 a la última	3
`[:2, :]`	De la 0 a la 1[1]	Todas
`[3:5, 1:3]`	De la 3 a la 4[1]	De la 1 a la 2[1]
`[5:3:-1, ::2]`	De la 3 a la 4[1], pero tomadas en orden inverso[2]	Todas las pares; se toman de dos en dos[2]

1. Se excluye la última del rango. Por ejemplo, en el rango `i:j` incluye de la `i` a la `j-1`
2. Cuando en un rango se pone un tercer valor éste se usa como paso para ir incrementando el índice. Por ejemplo i:j:k incluye los índices entre i (incluido) y j (excluido) pero tomados de k en k

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]`

Notas:

Las operaciones entre vectores y matrices también se admiten, siempre que las dimensiones sean las apropiadas

Operación	Resultado
`v_4 = np.array([10, 20])`	`[10, 20]`
`v_5 = a_1 @ v_4`	`[50, 20]`

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
```

Notas:

En el primer ejemplo formamos un vector con 5 ángulos expresados en grados

Luego los convertimos a radianes multiplicando escalarmente todos los elementos del vector por π y dividiendo entre 180

- Para el valor π usamos la constante `np.pi`

Finalmente calculamos el seno del vector, que se hace elemento a elemento

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 2019
"""

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

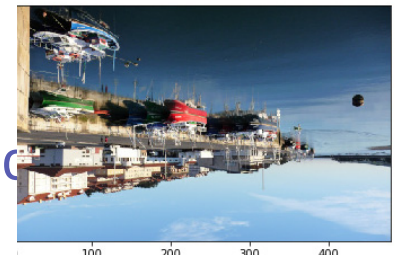
Ejemplo (cont.)

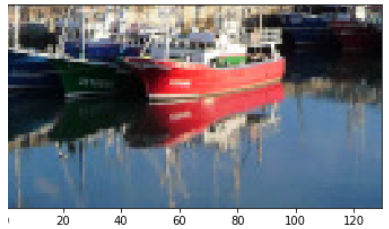
```
def main():
    """
    Prueba con una foto de San Vicente
    """
    # Leer y mostrar la imagen
    img = imageio.imread('san-vicente-small.jpg')
    plt.imshow(img)
    plt.show()

    # 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()
```





Notas:

En este ejemplo se muestra cómo podemos aprovechar las capacidades numéricas de `numpy` para hacer tratamiento de imágenes

- Una imagen se puede almacenar en un array de `numpy` con la función `imageio.imread()`
- Este array es de tres dimensiones: tiene filas y columnas correspondientes a cada píxel de la pantalla. Además, cada casilla es un array con tres valores correspondientes a las intensidades de cada color básico (rojo, verde, azul)
- Con la pareja de funciones `imshow()` y `show()` podemos mostrar una imagen

Se muestran cuatro imágenes:

- La imagen original
- La imagen invertida de arriba a abajo. En este caso observar que el índice de las filas (dimensión vertical) es `::-1` e incluye todas las filas, pero las recorre con paso -1 (en orden inverso). El índice horizontal es `:` e incluye todas las columnas
- La imagen invertida de izquierda a derecha (en espejo). Es como el caso anterior pero intercambiando los índices de las filas y las columnas
- Un fragmento de la imagen, comprendiendo en sentido vertical los píxeles de 130 a 199, y en sentido horizontal los píxeles 150 a 279

Apéndice

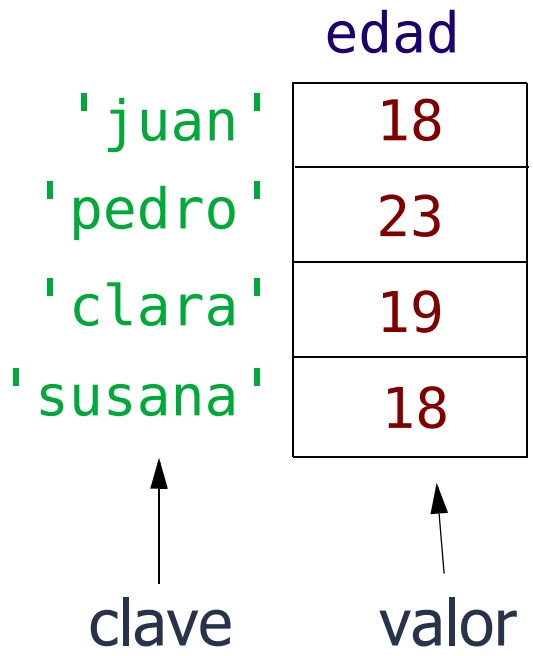
- A.1 Diccionarios
- A.2 Tipos enumerados

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

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
```

Notas:

El modelo de un diccionario es como el de una lista pero con la particularidad de que los índices no se limitan a valores numéricos enteros, y pueden ser otros datos como por ejemplo textos

- en este caso el índice se llama clave, y no se puede repetir
- el diccionario por tanto almacena parejas clave-valor
- aunque las claves deben ser inmutables, los valores pueden ser cualquier cosa
- el diccionario en sí es mutable; se pueden:
  - añadir nuevas parejas clave-valor, mediante asignación: `edad['elisa'] = 20`
  - borrarlas con la operación del: `del edad['pedro']`
  - o cambiar el valor asociado a una clave, también mediante asignación: `edad['susana'] = 19`

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 2019
"""

from typing import List

# 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 un tabulador
                palabra_morse += MORSE[carac.upper()]+"\t"
            # Añadimos la palabra y la palabra morse
            resul += palabra+":\t"+palabra_morse+"\n"
    return resul
```

Notas:

En este ejemplo se crea una constante que es un diccionario cuyas claves son caracteres, y los valores son los correspondientes códigos morse formados por puntos y rayas

La función `pasar_a_morse()` hace dos bucles

- El primer bucle recorre todas las palabras del texto. Las palabras se obtienen por medio del método `split()`, que parte el texto original en palabras que guardamos en la variable `palabras`
- El segundo bucle recorre cada letra de la palabra y obtiene su código morse pasando la letra a mayúscula con el método `upper()` y consultando el código en el diccionario

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

Notas:

Es muy habitual tener datos que pueden tomar valores de un conjunto de palabras. Además de los dos ejemplos mencionados podemos pensar muchos otros:

- facultades y escuelas de una universidad
- marcas de coches
- tipos de automóviles (coches, motos, autocaravanas, ...)

Una forma de guardar estos datos sería mediante strings, pero

- en comparación con los números enteros son poco eficientes tanto en términos de memoria como de lentitud de las operaciones con ellos
- sería fácil equivocarse y usar valores no previstos en el conjunto de valores válidos. Por ejemplo, cometer una falta de ortografía y usar como día de la semana la palabra "dmingo"

Otra forma más eficiente sería usar números enteros, pero

- también podría haber errores si usamos números no previstos
- es difícil recordar qué palabra corresponde a cada número ¿Cuál es la facultad 17?

Por estos motivos casi todos los lenguajes de programación, Python incluido, tienen tipos de datos especiales llamados enumerados

- guardan palabras en forma de números y por tanto son muy eficientes
- evitan errores y hacen el programa más fácil de entender que si se usan números directamente

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
```

Notas:

En Python los tipos enumerados se definen mediante clases enumeradas, cuya particularidad es que en la cabecera llevan la especificación `(Enum)`

En esta clase basta con definir los valores enumerados mediante constantes, tal como se muestra arriba con el ejemplo de los colores

Al estar estas constantes definidas directamente en la clase (no en el constructor) son atributos estáticos o de clase

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)
```
- 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

Notas:

Arriba se muestra cómo se usa una clase enumerada

- los valores de la clase se usan mediante sus atributos: `NombreClase.atributo`
- estos valores se pueden imprimir en pantalla, obteniéndose su nombre (no su valor entero)
  - en el ejemplo que se muestra, se pondría la palabra `VERDE` en pantalla
- también se puede obtener directamente el nombre con el atributo `name`
  - así, col.name vale `"VERDE"`
- se pueden obtener valores enumerados directamente
  - a partir de su valor numérico, con `NombreClase(valor_numerico)`
  - o de su nombre almacenado en un string, con `NombreClase[nombre_del_valor]`

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)
```

Notas:

En el ejemplo se ha creado una clase enumerada conteniendo los días de la semana

Posteriormente se muestra cómo trabajar con valores enumerados

Finalmente se muestra cómo recorrer todos los valores de tipo enumerado, es decir, todos los días de la semana