

# INTRODUCCIÓN A LA PROGRAMACIÓN

CON  python

LISTAS

## Estructuras de datos

Una **estructura de datos** es la **colección de datos** que se caracterizan por su **organización** y las **operaciones** que se definen en ella.

# Estructuras de datos

En general tenemos:

- cadenas
- **listas**
- tuplas
- diccionarios
- conjuntos

Nombre	Tipo	Ejemplo	Descripción
Lista	<code>list</code>	<code>[1, 2, 3]</code>	Secuencia heterogénea de datos mutable
Tupla	<code>tuple</code>	<code>1, 2, 3</code> o <code>(1, 2, 3)</code>	Secuencia heterogénea de datos inmutable
Rango	<code>range</code>	<code>range(1, 20, 2)</code>	Secuencia de enteros inmutable
Cadena de caracteres	<code>str</code>	<code>'Hola'</code>	Secuencia de caracteres inmutable
Diccionario	<code>dict</code>	<code>{'a':1, 'b':2, 'c':3}</code>	Tabla asociativa de valores únicos (clave, valor)
Conjunto	<code>set</code>	<code>{1, 2, 3}</code>	Colección sin orden de valores únicos

## Listas

Una lista es una colección **ordenada** en la que los elementos pueden ser de **distintos tipo**.

```
animales = ["gato", "perro", "conejo", "elefante"]
```

```
mi_lista = [ 10, "hola", "Pepe", 25]
```



## Listas. Características

---

Características de las listas:

- Puede contener elementos de **diferente tipo**: números, cadenas, booleanos, ... y también listas.
- Es una estructura dinámica y puede cambiar de tamaño, es decir, podemos añadir y quitar elementos. Es una **estructura de datos mutable**
- Los elementos están **ordenados, en cuanto a su posición en la lista**. No es que por defecto se les aplique un orden al insertarlos. Además dado que pueden ser de distintos tipos, no siempre se puede establecer un orden según su valor.

## Listas. Creación

---

**Crear** una lista es tan sencillo como indicar entre corchetes, y separados por comas, los valores que queremos incluir en la lista:

```
animales = ["gato", "perro", "conejo", "elefante"]
```

```
mi_lista = [ 10, "hola", "Pepe", 25]
```

Lista vacía:

```
animales = []
```

## Listas. Acceso a un elemento

---

Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes.

```
animales = ["gato", "perro", "conejo", "elefante"]  
           |         |         |         |  
animales[0] animales[1] animales[2] animales[3]
```

Ten en cuenta sin embargo que el índice del primer elemento de la lista es 0, y no 1.

Así pues animales[0] es "gato", animales[1] es "perro", etc.

## ¡Pruébalo!

---

```
>>>animales = ["gato","perro","conejo","elefante"]
```

```
>>>"hola " + animales[3]
```

```
???
```

```
>>> "el " + animales[3] + " se comió al " + animales[1]
```

```
???
```

## Listas. Acceso a un elemento

---

Los índices positivos comienzan en 0 pero también se pueden utilizar enteros negativos como índices. Por ejemplo, -1 se refiere al último elemento de la lista, -2 se refiere al anteúltimo, etc.

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> animales[-1]
'elefante'
```

```
>>> animales[-3]
'perro'
```

## Listas. Acceso a un elemento

---

Python dará un mensaje de error de tipo **IndexError** si se usa un índice que exceda el número de valores en la lista

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> animales[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

## Listas. Obtener una porción. Slicing (rebanar)

---

Así como un índice me permite obtener un valor de una lista, existe un mecanismo para obtener varios valores de una lista.

Si en lugar de un número escribimos dos números inicio y fin separados por dos puntos [inicio:fin], Python interpretará que queremos una lista que vaya desde la posición inicio a la posición fin, **sin incluir este último**.

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales[1:3]
['perro', 'conejo']
```

## Listas. Obtener una porción. Slicing (rebanar)

---

Hay que mencionar así mismo que no es necesario indicar el principio y el final, sino que, si estos se omiten, se usarán por defecto las posiciones de inicio y fin de la lista, respectivamente:

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales[:2]
['gato', 'perro']
```

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales[1:]
['perro', 'conejo', 'elefante']
```

## Listas. Obtener una porción. Slicing (rebanar)

---

En forma general

<code>a[inicio:fin]</code>	# desde <code>inicio</code> hasta <code>fin-1</code>
<code>a[inicio:]</code>	# desde <code>inicio</code> hasta el <code>final de la cadena</code>
<code>a[:fin]</code>	# desde el <code>principio de la cadena</code> hasta <code>fin-1</code>
<code>a[:]</code>	# Todos los elementos
<code>a[inicio:fin:pasos]</code>	# desde <code>inicio</code> hasta <code>fin-1</code> , en los <code>pasos</code> indicados por pasos

## Listas. Obtener una porción. Slicing (rebanar)

---

En el troceado de listas, a diferencia de lo que ocurre al acceder a los elementos, no debemos preocuparnos por acceder a índices inválidos (fuera de rango) ya que Python los restringirá a los límites de la lista:

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> animales[2:100]
['conejo', 'elefante']
```

## Listas. Longitud de una lista

---

La función **len()** devolverá la cantidad de elementos de la lista pasada como argumento.

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> len(animales)
4
```

## Listas. Utilizar **range** para hacer una lista de números

---

Si quieres hacer una lista de números, puedes convertir los resultados de **range()** directamente a una lista utilizando la función **list()** que tome como argumento el resultado de **range**. Esto devuelve una lista con los números generados por la función **range()**

```
numeros = list(range(1, 6))
print(numeros)
```

La ejecución del código producirá el siguiente resultado:

```
[1, 2, 3, 4, 5]
```



## Listas. Utilizar `range` para hacer una lista de números

---

Podemos usar `range()` para decirle a Python saltee números en una secuencia. Por ejemplo,

```
numeros = list(range(2, 11, 2))  
print(numeros)
```

La ejecución de este código producirá el siguiente resultado

```
[2, 4, 6, 8, 10]
```

## Listas. Máximo, mínimo y suma de sus elementos

---

Para obtener el máximo y mínimo se puede utilizar la función `max()` y `min()`, respectivamente

```
>>> lista = [1, 0, -1, 6, 2]  
>>> max(lista)  
6  
>>> min(lista)  
-1
```

Para obtener la suma de los elemento se utiliza la función `sum()`

```
>>> lista = [1, 0, -1, 6, 2]  
>>> sum(lista)  
8
```

## Listas. Cambiar elementos

---

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales[1] = "rata"
>>> animales
['gato', 'rata', 'conejo', 'elefante']
```

Significa que asigna en la posición cuyo índice es 1 el elemento “rata”

```
>>> animales[-1] = "mono"
>>> animales
['gato', 'rata', 'conejo', 'mono']
```

Significa que asigna en la última posición el elemento “mono”

## Listas. Concatenación y repetición

---

El operador + se puede utilizar para concatenar dos listas creando una nueva lista (tal como se utiliza para concatenar cadenas). Y el operador \* se puede utilizar con una lista y un entero para repetir la lista.

```
>>> [1,2] + ["a","z","f"]
[1, 2, 'a', 'z', 'f']
```

```
>>> ["*","-"]*3
['*', '-', '*', '-', '*', '-']
```

```
>>> [1,"hola"]*2
[1, 'hola', 1, 'hola']
```

## Listas. Pertenencia de un elemento

---

Para determinar si un elemento está o no en una lista se utilizan los operadores **in** y **not in**. Obtendremos como resultado un booleano, siendo True si el elemento está en la lista y False en caso contrario.

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> "perro" in animales
True
>>> "mono" in animales
False
>>> "perro" not in animales
False
>>> "mono" not in animales
True
```

## Listas. Encontrar un elemento

---

La estructura de datos lista tiene métodos útiles para encontrar, añadir, remover valores en una lista. Para encontrar un valor en una lista se utiliza el método **index(elemento)**

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> animales.index("gato")
0
>>> animales.index("elefante")
3
>>> animales.index("mono")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'mono' is not in list
```

## Listas. Agregar un elemento

Para agregar nuevos valores a una lista se utilizan los métodos `append()` e `insert()`.

El método `append()` añade un elemento al final de la lista

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales.append("mono")
>>> animales
['gato', 'perro', 'conejo', 'elefante', 'mono']
```

El método `insert()` añade un elemento en una posición de la lista

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales.insert(1,"mono")
>>> animales
['gato', 'mono', 'perro', 'conejo', 'elefante']
```

Al igual que ocurría con el troceado de listas, en este tipo de inserciones no obtendremos un error si especificamos índices fuera de los límites de la lista. Estos se ajustarán al principio o al final en función del valor que indiquemos

## Listas. Eliminar un elemento

Para eliminar un elemento de una lista se utiliza el método `remove()`

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales.remove("conejo")
>>> animales
['gato', 'perro', 'elefante']
```

Si el valor aparece varias veces en la lista solo la primer instancia se elimina.

```
>>> animales = ["gato","perro","conejo","gato","elefante"]
>>> animales.remove("gato")
>>> animales
['perro', 'conejo', 'gato', 'elefante']
```

## Listas. Eliminar un elemento

---

Para eliminar un elemento mediante la función **del()** indicando su índice:

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> del(animales[2])
>>> animales
['gato', 'perro', 'elefante']
```

## Listas. Eliminar todos los elementos

---

El método **clear()** borra todos los elementos de una lista

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales.clear()
>>> animales
[]
```

Otra forma de borrar una lista es reiniciándola con vacío

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales = []
>>> animales
```

## Listas. Acceso + borrado

---

La función `del()` y el método `remove()` efectivamente borran el elemento indicado de la lista, pero no «devuelven» nada. Sin embargo, Python nos ofrece la función **`pop()`** que además de borrar, nos «recupera» el elemento; algo así como una extracción. Lo podemos ver como una combinación de acceso + borrado:

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales.pop()
'elefante'
>>> animales
['gato', 'perro', 'conejo']

>>> animales = ["gato","perro","conejo","elefante"]
>>> animales.pop(1)
'perro'
>>> animales
['gato', 'conejo', 'elefante']
```

## Listas. Ordenar elementos

---

### Conservando la lista original

Mediante la función `sorted()` que devuelve una nueva lista ordenada (no modifica la original):

```
>>> lista = [2,5,3.14,1,-7]
>>> sorted(lista)
[-7, 1, 2, 3.14, 5]
```

## Listas. Ordenar elementos

### Modificando la lista original

Se pueden ordenar listas de elementos numéricos o cadenas con el método `sort()` y `reverse()`

```
>>> lista=[2,5,3.14,1,-7]
>>> lista.sort()
>>> lista
[-7, 1, 2, 3.14, 5]

>>> animales = ["zorro","gato","perro","foca", "elefante"]
>>> animales.sort()
>>> animales
['elefante', 'foca', 'gato', 'perro', 'zorro']

>>> lista=[2,5,3.14,1,-7]
>>> lista.sort(reverse=True)
>>> lista
[5, 3.14, 2, 1, -7]
```

Una vez ordenada sabremos que el primero o el último serán el mínimo o el máximo valor.

Si tenemos una lista que mezcla distintos tipos, por ejemplo números y letras nos dará una excepción de tipo **TypeError**.

## Listas. Desordenar elementos

La función **shuffle()** en el módulo **random** toma una secuencia, como una lista, y reorganiza el orden de los elementos.

```
>>> from random import shuffle
>>> animales = ["gato","perro","conejo","elefante"]
>>> shuffle(animales)
>>> animales
['elefante', 'perro', 'conejo', 'gato']
>>> shuffle(animales)
>>> animales
['elefante', 'gato', 'perro', 'conejo']
```

## Listas. Invertir una lista

---

### Conservando la lista original

Mediante slicing

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales[::-1]
['elefante', 'conejo', 'perro', 'gato']
```

Mediante la función reversed()

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> list(reversed(animales))
['elefante', 'conejo', 'perro', 'gato']
```

## Listas. Invertir una lista

---

### Modificando la lista original

Utilizando el método **reverse()**

```
>>> animales = ["gato","perro","conejo","elefante"]
>>> animales.reverse()
>>> animales
['elefante', 'conejo', 'perro', 'gato']
```



## Listas. Contar elemento

Se pueden contar las ocurrencias de un elemento en una lista con **count(elemento)**

```
>>> animales = ["zorro", "gato", "perro", "foca", "elefante", "foca"]
>>> lista.count("foca")
2
```

## Listas. Copia

El método **copy()** devuelve una copia de la lista. Podemos usarlo para crear una nueva si no queremos alterar la original.

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> mis_animales = animales.copy()
>>> mis_animales
['gato', 'perro', 'conejo', 'elefante']
```

Otra forma de hacerlo

```
>>> animales = ["gato", "perro", "conejo", "elefante"]
>>> mis_animales = animales[:]
>>> mis_animales
['gato', 'perro', 'conejo', 'elefante']
>>> animales.remove("conejo")
>>> animales
['gato', 'perro', 'elefante']
>>> mis_animales
['gato', 'perro', 'conejo', 'elefante']
```

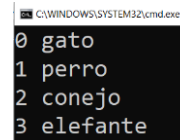
## Listas. Recorrer

---

```
animales = ["gato", "perro", "conejo", "elefante"]
for animal in animales:
    print(animal, end=" ")
```

A este proceso de recorrer una lista se le denomina enumerar o iterar. A veces necesitamos además de iterar el indicar la posición del elemento. Podemos hacerlo con la palabra reservada `enumerate`, obteniendo tanto el elemento como su posición.

```
animales = ["gato", "perro", "conejo", "elefante"]
for posicion, animal in enumerate(animales):
    print(posicion, animal)
```



```
C:\WINDOWS\SYSTEM32\cmd.exe
0 gato
1 perro
2 conejo
3 elefante
```

## Listas. Recorrer

---

Recorrer una lista con índices (**cambiar por un ejemplo más útil**)

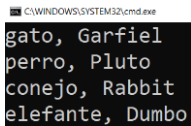
```
animales = ["gato", "perro", "conejo", "elefante"]
for i in range(len(animales)):
    print(animales[i], end=" ")
```

## Listas. Recorrer múltiples listas

Python ofrece la posibilidad de iterar sobre múltiples listas en paralelo utilizando la función `zip()`

```
animales = ["gato", "perro", "conejo", "elefante"]
nombres = ["Garfiel", "Pluto", "Rabbit", "Dumbo"]

for animal, nombre in zip(animales, nombres):
    print(animal + ", " + nombre)
```



```
C:\WINDOWS\SYSTEM32\cmd.exe
gato, Garfiel
perro, Pluto
conejo, Rabbit
elefante, Dumbo
```

## Listas. Creación a partir de otra

En ocasiones necesitamos generar una lista a partir de otra, realizando una operación sobre sus elementos. Podemos hacerlo con lo que ya sabemos, recorriendo nuestra lista y añadiendo los elementos a una nueva lista:

```
import math
radios = [2.0, 35, 7.1]
circunsferencias = []
for r in radios:
    circunsferencias.append(r*2*math.pi)
print(circunsferencias)
```

Pero como esto es algo muy frecuente, Python incorpora una manera muy cómoda de hacerlo, donde especificamos dentro de la lista la manera en la que calculamos cada elemento a partir de un bucle for:

```
radios = [2.0, 35, 7.1]
circunsferencias = [2*math.pi*r for r in radios]
print(circunsferencias)
```

## Listas. Listas por comprensión

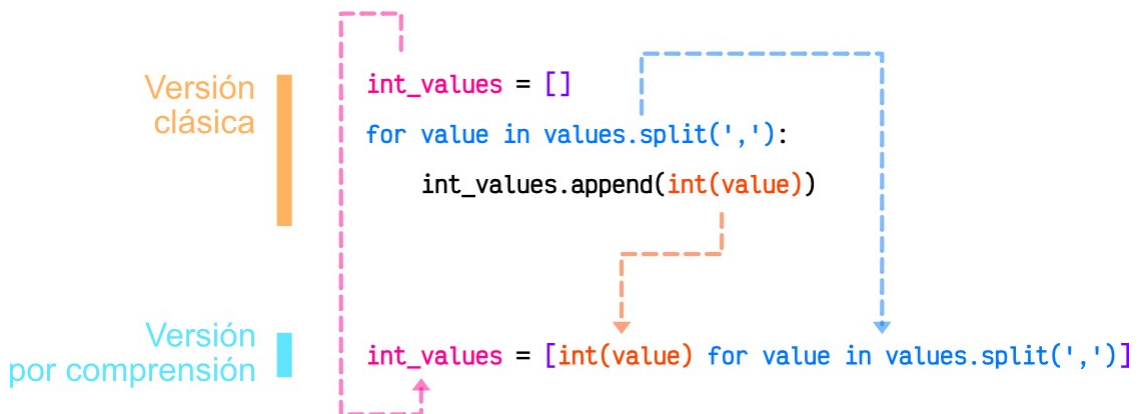
La forma de creación vista en la diapositiva anterior se denomina **listas por comprensión**.

Las listas por comprensión establecen una técnica para crear listas de forma más compacta basándose en el concepto matemático de conjuntos definidos por comprensión.



```
>>> valores = '32,45,11,87,20,48'
>>> valores_enteros = [int(valor) for valor in valores.split(',')]
>>> valores_enteros
[32, 45, 11, 87, 20, 48]
```

## Listas. Listas por comprensión



## Listas. Listas por comprensión

---

También existe la posibilidad de incluir condiciones en las listas por comprensión.

Continuando con el ejemplo anterior, supongamos que sólo queremos crear la lista con aquellos valores que empiecen por el dígito 4:

```
>>> valores = '32,45,11,87,20,48'
>>> valores_enteros = [int(v) for v in valores.split(',') if v.startswith("4")]
>>> valores_enteros
[45, 48]
```

## Listas. Conversión

---

Para convertir otros tipos de datos en una lista podemos usar la función `list()`:

```
>>> list("python")
['p', 'y', 't', 'h', 'o', 'n']
```

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto Python se ha creado una lista con 6 elementos, donde cada uno de ellos representa un carácter de la cadena. Podemos extender este comportamiento a cualquier otro tipo de datos que permita ser iterado (iterables).

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Listas. Conversión

---

Existe una manera particular de usar `list()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en una lista, con lo que obtendremos una lista vacía:

```
>>> list()  
[]
```

Para crear una lista vacía, se suele recomendar el uso de `[]` frente a `list()`, no sólo por ser más pitónico sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.


## Listas. Actividad

---

1. Dada la lista `[3,-2,5,7,0,1,5]` extrae la sublista `[5,7,0]` , añade el número 3.14 y elimina el 7 y muestra la sublista resultante por pantalla.
2. Escribe un programa que permita crear una lista de 5 nombres de alumnos. Para ello, el programa tiene que pedir por pantalla ese número de nombres para crear la lista. Por último, el programa tiene que mostrar la lista resultante. Nota: utiliza la sentencia `alumnos=[]` para crear una lista vacía
3. Modifica el programa anterior para que pida un nombre y diga cuántas veces aparece ese nombre en la lista.
4. Escribe un programa que permita crear una lista de 5 nombres de alumnos y que, a continuación, pida dos nombres y sustituya el primero por el segundo en la lista. Por último, el programa tiene que mostrar la lista resultante.
5. Escribe un programa que permita crear una lista de 5 nombres de alumnos y que, a continuación, pida un nombre y elimine ese nombre de la lista. Por último, el programa tiene que mostrar la lista resultante.
6. Escribe un programa que permita crear una lista de 5 nombres de alumnos y que, a continuación, la ordene pero en orden inverso. Por último, el programa tiene que mostrar la lista resultante.

## Listas. Autoevaluación


---

1. ¿Qué es []?
  2. Si lista=[1,4,6,8,10], cómo puedo asignar "hello" como tercer valor de la lista?
  3. Supongamos que lista=["a","b","c","d"] ¿Cuál es el resultado de evaluar lista[int("3"\*2)/11]?
  4. ¿Cuál es el resultado de evaluar lista[-1]?
  5. ¿Cuál es el resultado de evaluar lista[-2]?
  6. Supongamos que lista=[3.14, "gato",11,"gato",True] ¿Cuál es el resultado de evaluar lista.index("gato")?
  7. ¿Cuáles son los operadores de concatenación y reproducción de listas?
  8. ¿Cuál es la diferencia entre los métodos append() e insert()?
  9. ¿Cuáles son las dos formas de remover valores de una lista?
- 

## Mutabilidad en Python

---

Antes de ver el apartado de **paso de parámetros** recordemos que los diferentes tipos de Python u otros objetos en general, pueden ser clasificados atendiendo a su mutabilidad:

- **Mutable**s: Si permiten ser modificados una vez creados.
  - **Inmutable**s: Si no permiten ser modificados una vez creado
- 


# Mutabilidad en Python

---

Son **mutables** los siguientes tipos:

- Listas
- Diccionarios
- Sets
- Clases definidas por el usuario

Y son **inmutables**:


- Booleanos
  - Enteros
  - Float
  - Cadenas
  - Tuplas
  - Range
- 

# Mutabilidad en Python

---

Recordada esta clasificación, tal vez te preguntes porqué es esto relevante. Pues bien, Python trata de manera diferente a los tipos mutables e inmutables, y si no entiendes bien este concepto, puedes llegar a tener comportamientos inesperados en tus programas.

La mutabilidad de los objetos es una característica muy importante cuando tratamos con funciones, ya que Python los tratará de manera distinta.





# Mutabilidad en Python

En muchos lenguajes de programación existen los conceptos de paso por **valor** y por **referencia** que aplican a la hora de como trata una función a los parámetros que se le pasan como entrada. Su comportamiento es el siguiente:

- Si usamos un parámetro pasado por **valor**, se creará una copia local de la variable, lo que implica que cualquier modificación sobre la misma no tendrá efecto sobre la original.
- Con una variable pasada como **referencia**, se actuará directamente sobre la variable pasada, por lo que las modificaciones afectarán a la variable original. Lo que realmente hace es copiar en los parámetros la dirección de memoria de las variables que se usan como argumento. Esto implica que realmente hagan referencia al mismo elemento y cualquier modificación del valor en el parámetro afectará a la variable externa correspondiente.

Los tipos inmutables son **pasados por valor** y los tipos mutables son **pasados por referencia**.

## Función. Paso de parámetro por valor o por referencia

Las variables en Python son identificadores que referencian a objetos almacenados en bloques de memoria. Si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función. Sin embargo si pasamos un objeto de un tipo mutable podremos cambiar su valor:

Tenemos una función que modifica dos variables:

```
#tenemos una función que modifica dos parámetros
def mi_funcion(a, b):
    a = 10
    b[0] = 10

# x es un entero
x = 5
# y es una lista
y = [1,2,3]
```

Si llamamos a la función con ambas variables, vemos como el valor de **x** no ha cambiado, pero el de **y** sí.

```
mi_funcion(x, y)

print(x)
print(y)
```

El resultado será

```
5
[10, 2, 3]
```

## Función. Paso de parámetro por valor o por referencia

Otro ejemplo: 

```
def mi_funcion(mi_lista):
    mi_lista.append(5)
    return mi_lista

lista = [1,2]
nueva_lista = mi_funcion(lista)
print(lista)
print(nueva_lista)
```

```
[1, 2, 5]
[1, 2, 5]
```

La ejecución de este código da por resultado la lista pasada como argumento modificada

**En general, se espera que una función que recibe parámetros mutables, no los modifique, ya que si se los modifica se podría perder información valiosa. En el caso en que por una decisión de diseño o especificación se modifiquen los parámetros recibidos, esto debe estar claramente documentado.**

## Función. Paso de parámetro por valor o por referencia

Para evitar que una función modifique una lista puedes enviar una copia de la lista a la función.

```
def mi_funcion(mi_lista):
    mi_lista.append(5)
    return mi_lista

lista = [1,2]
nueva_lista = mi_funcion(lista[:])
print(lista)
print(nueva_lista)
```

La ejecución de este código da por resultado la lista pasada como argumento sin modificar

```
[1, 2]
[1, 2, 5]
```

## Función. if `__name__ == "__main__"`

---

Es común encontrarnos código con esta forma:

```
def hacer_algo():  
    print("algo")  
  
if __name__ == "__main__":  
    hacer_algo()
```

En lugar de, por ejemplo:

```
def hacer_algo():  
    print("algo")  
  
hacer_algo()
```

## Función. if `__name__ == "__main__"`

---

Cuando un intérprete de Python lee un archivo de Python, primero establece algunas variables especiales. Luego ejecuta el código desde el archivo.

Una de esas variables se llama `__name__`

Los archivos de Python se llaman módulos y se identifican mediante la extensión de archivo `.py`. Un módulo puede definir funciones, clases y variables.

## Función. if `__name__ == "__main__"`

---

Entonces, cuando el intérprete ejecuta un programa, el variable `__name__` se establecerá como `__main__` si el módulo que se está ejecutando es el programa principal.

La variable `__name__` para el archivo/módulo que se ejecuta será siempre `__main__`. Pero la variable `__name__` para todos los demás archivos/módulos que se importan se establecerá en el nombre de su módulo.

## Función. if `__name__ == "__main__"`

---

La forma habitual de usar `__name__` y `__main__` se ve así:

```
def media(x,y):
    resultado = (x+y)/2
    return resultado

if __name__ == "__main__":
    a = 2
    b = 3
    m = media(a,b)
    print(m)
```

Explicación completa en el este [enlace](#)

## Estructura de un programa

Puedes estructurar tu programa de cualquiera de estas dos formas

```
def media(x,y):
    resultado = (x+y)/2
    return resultado

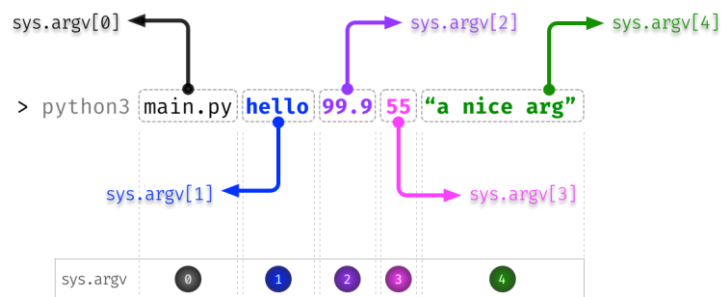
if __name__ == "__main__":
    a = 2
    b = 3
    m = media(a,b)
    print(m)
```

```
def media(x,y):
    resultado = (x+y)/2
    return resultado

#programa principal
a = 2
b = 3
m = media(a,b)
print(m)
```

## Listas. Argumentos en la línea de comandos

Cuando queramos ejecutar un programa Python desde línea de comandos, tendremos la posibilidad de acceder a los argumentos de dicho programa. Para ello se utiliza una lista que la encontramos dentro del módulo `sys` y que se denomina **argv** y cuyos elementos son cadenas.



## Listas. Argumentos en la línea de comandos

Veamos una aplicación de lo anterior en un programa que convierte un número decimal a una determinada base, ambos argumentos pasados por línea de comandos. Ejemplo,

***python convertir.py 10 2***

```
import sys

numero = int(sys.argv[1])
a_base = int(sys.argv[2])

match a_base:
    case 2:
        resultado = f'{numero:b}'
    case 8:
        resultado = f'{numero:o}'
    case 16:
        resultado = f'{numero:x}'
    case _:
        resultado = None

if result is None:
    print(f'¡Base {a_base} no implementada!')
else:
    print(resultado)
```

## Argumentos en la línea de comandos. Actividad

1. Escribe un programa llamado **suma\_digitos.py** con una función **suma()** para sumar los dígitos individuales de un entero positivo, dado en la línea de comando. La salida debe ser:

```
python suma_digitos.py 12345
```

La suma de los dígitos: 1 + 2 + 3 + 4 + 5 = 15

2. Escribe un programa llamado **media\_edades.py** con una función **media()** que reciba los nombres y edades de varias personas como argumentos en la línea de comandos de la siguiente forma:

```
python media_edades.py Mónica 12 Daniel 34 Carlos 23
```

y que calcule la media de las edades recibidas y la muestre por pantalla

La media de las edades es: 23

## Listas. Listas bidimensionales

---

¿Qué es una matriz?

## Listas. Listas bidimensionales

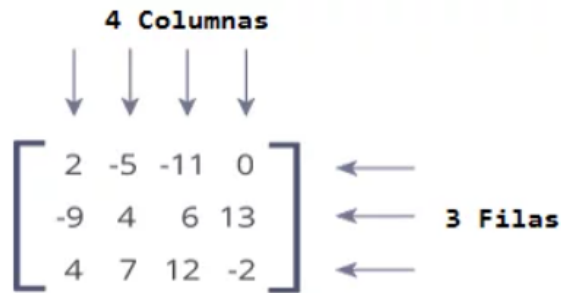
---

Una matriz es una colección de elementos dispuestos en **filas y columnas**.

En los campos de la ingeniería, la física, la estadística y los gráficos, las matrices se utilizan ampliamente para expresar rotaciones de imágenes y otros tipos de transformaciones.

## Listas. Listas multidimensionales

Las **matrices** son una estructura de datos **bidimensional**. Esta **matriz** es una matriz de 3x4 porque tiene 3 filas y 4 columnas



## Matrices: creación

Python **no tiene un tipo de dato incorporado para trabajar con matrices**, sin embargo, hemos dicho que una lista puede contener cualquier cosa. De hecho podemos hacer una **lista que contenga listas**, es lo que llamamos una matriz. Veamos un ejemplo

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

```
print("Matriz =", matriz)
```

La ejecución de este código producirá:

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



## Matrices: acceso

```
matriz = [[1, 3, 4], [9, 3, 5], [7, 3, 8]]
print(matriz)
print(matriz[1])          #segunda fila
print(matriz[1][2])       #elemento en la segunda fila y tercer columna
print(matriz[0][-1])      #último elemento de la primer fila

columna = []
for fila in matriz:
    columna.append(fila[2])

print("La tercer columna es:",columna)
```

```
[[1, 3, 4], [9, 3, 5], [7, 3, 8]]
[9, 3, 5]
5
4
La tercer columna es: [4, 5, 8]
```

Como hemos visto para acceder a una celda de la matriz hay que usar **dos índices** fila y columna

## Matrices: creación e inicialización

```
#inicializa una matriz de nxm con ceros
n = 5
m = 7
matriz = [[0] * m for _ in range(n)]
print(matriz)
```

```
#inicializa una matriz de nxm con números aleatorios
matriz = []
for i in range(n):
    fila = [randint(1,10) for _ in range(m)]
    matriz.append(fila)
```

## Matrices: recorrido

Podemos recorrer una matriz utilizando un bucle for. Si queremos imprimir todos los elementos de una matriz, podemos hacerlo de la siguiente manera:

```
matriz = [[1,2],[5,3]]
for f in range(len(matriz)):
    for c in range(len(matriz[0])):
        print(matriz[f][c])
```

1  
2  
5  
3

Que producirá la siguiente salida:

Una forma **más intuitiva** que produce la misma salida es

```
matriz = [[1,2],[5,3]]
for fila in matriz:
    for elemento in fila:
        print(elemento)
```

## Matrices: recorrido

**[5, 7], [7, 6]**

Supongamos que queremos sumar dos matrices. Podemos hacerlo primero iterando por filas y luego por columnas de ambas matrices a sumar:

$$\begin{pmatrix} 2 & 5 & 4 \\ 3 & 6 & 0 \\ 9 & 1 & 7 \end{pmatrix} + \begin{pmatrix} 8 & 7 & 3 \\ 7 & 2 & 1 \\ 4 & 5 & 9 \end{pmatrix} = \begin{pmatrix} 2+8 & 5+7 & 4+3 \\ 3+7 & 6+2 & 0+1 \\ 9+4 & 1+5 & 7+9 \end{pmatrix} = \begin{pmatrix} 10 & 12 & 7 \\ 10 & 8 & 1 \\ 13 & 6 & 16 \end{pmatrix}$$

O podemos recorrer las dos matrices a la vez, obteniendo cada fila de cada matriz y sumando cada elemento de cada fila:

```
matriz1 = [[1,2],[5,3]]
matriz2 = [[4,5],[2,3]]
matriz3 = [[0,0],[0,0]]

#itero a través de las filas
for f in range(len(matriz1)):
    # itero a través de las columnas
    for c in range(len(matriz1[0])):
        matriz3[f][c] = matriz1[f][c] + matriz2[f][c]
print(matriz3)

matriz1 = [[1,2],[5,3]]
matriz2 = [[4,5],[2,3]]
matriz3 = []
for fila1, fila2 in zip(matriz1,matriz2):
    fila = []
    for n1, n2 in zip(fila1,fila2):
        n = n1 + n2
        fila.append(n)
    matriz3.append(fila)
print(matriz3)
```

# Matrices: librería

Con la librería **numpy** (hay que instalarla previamente) . La suma de matrices es directa

```
from numpy import *
import numpy as np

matriz1 = [[1,2],[5,3]]
matriz2 = [[4,5],[2,3]]

filas = len(matriz1)
columnas = len(matriz1[0])

suma = zeros((filas,columnas))
suma = np.array(matriz1) + np.array(matriz2)
print(suma)
```

C:\WINDOWS\SYSTEM32\cmd.exe

```
[[5 7]
 [7 6]]
```

## Matrices. Impresión

```
for fila in matriz:
    for valor in fila:
        print(f"{valor:4}", end=" ")
    print()
```

```
15    7
 7    6
```

```
for fila in matriz:
    print(' '.join([str(elem) for elem in fila]))
```

```
15 7
7 6
```

## Matrices. Actividad

1. Crea una función **suma(matriz1,matriz2)** que sume dos matrices y devuelva la matriz resultante. Crea una función **imprime(matriz)** para que muestre la matriz resultado

$$\begin{pmatrix} 2 & 5 & 4 \\ 3 & 6 & 0 \\ 9 & 1 & 7 \end{pmatrix} + \begin{pmatrix} 8 & 7 & 3 \\ 7 & 2 & 1 \\ 4 & 5 & 9 \end{pmatrix} =$$

$$= \begin{pmatrix} 2+8 & 5+7 & 4+3 \\ 3+7 & 6+2 & 0+1 \\ 9+4 & 1+5 & 7+9 \end{pmatrix} = \begin{pmatrix} 10 & 12 & 7 \\ 10 & 8 & 1 \\ 13 & 6 & 16 \end{pmatrix}$$

## Matrices. Copia

Para copiar una matriz **no se puede usar** la asignación. Veamos el siguiente código:

```
matriz1 = [[1,1,1],[1,1,1],[1,1,1]]
matriz2 = matriz1
matriz2[0][0]=9
print(matriz1)
print(matriz2)
```

Al cambiar un valor en la segunda matriz también se modifica la primera

```
[[9, 1, 1], [1, 1, 1], [1, 1, 1]]
[[9, 1, 1], [1, 1, 1], [1, 1, 1]]
```

## Matrices. Copia

**Tampoco** se puede utilizar el método `copy()`. Veamos el siguiente código:

```
matriz3 = [[1,1,1],[1,1,1],[1,1,1]]
matriz4 = matriz3.copy()
matriz4[0][0]=9
print(matriz3)
print(matriz4)
```

Al cambiar un valor en la segunda matriz también se modifica la primera

```
[[9, 1, 1], [1, 1, 1], [1, 1, 1]]
[[9, 1, 1], [1, 1, 1], [1, 1, 1]]
```

## Matrices. Copia

Si funciona el siguiente código **copiando elemento por elemento**

```
matriz5 = [[1,1,1],[1,1,1],[1,1,1]]
matriz6 = [fila[:] for fila in matriz5]
matriz6[0][0]=9
print(matriz5)
print(matriz6)
```

Usando la función **deepcopy** en el módulo `copy` (form `copy import deepcopy`) o utilizando el método **copy()** para cada fila también funciona

```
matriz7 = [[1,1,1],[1,1,1],[1,1,1]]    matriz9 = [[1,1,1],[1,1,1],[1,1,1]]
matriz8 = deepcopy(matriz7)             matriz10 = [fila.copy() for fila in matriz9]
matriz8[0][0]=9                          matriz10[0][0]=9
print(matriz7)                           print(matriz5)
print(matriz8)                           print(matriz6)
```

Al cambiar un valor en la segunda matriz **NO** se modifica la primera

```
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]
[[9, 1, 1], [1, 1, 1], [1, 1, 1]]
```