


# match

En Python 3.10, se incorpora la coincidencia de patrones estructurales mediante las declaraciones `match` y `case`. Esto permite asociar acciones específicas basadas en las formas o patrones de tipos de datos complejos.

```
match objeto:
    case <patron_1>:
        <accion_1>
    case <patron_2>:
        <accion_2>
    case <patron_3>:
        <accion_3>
    case _:
        <accion_comodin>
```



*El caracter `_` es un comodín que actúa como coincidencia si la misma no se produce en los casos anteriores.*

Es posible detectar y deconstruir diferentes estructuras de datos: esto quiere decir que los patrones no son únicamente valores literales (strings o números), sino también estructuras de datos, sobre los cuales se buscan coincidencias de construcción.

# comprensión de listas

La comprensión de listas ofrece una sintaxis más breve en la creación de una nueva lista basada en valores disponibles en otra secuencia. Vale la pena mencionar que la brevedad se logra a costo de una menor interpretabilidad.

*cada elemento del iterable*      *tuplas, sets, otras listas...*

```
nueva_lista = [expresion for item in iterable if condicion == True]
```

*fórmula matemática*      *operación lógica*

Caso especial con else:

```
nueva_lista = [expresion if condicion == True else otra_expresion  
               for item in iterable]
```

Ejemplo:

```
nueva_lista = [num**2 for num in range(10) if num < 5]  
print(nueva_lista)  
>> [0, 1, 4, 9, 16]
```

# random

Python nos facilita un módulo (un conjunto de funciones disponibles para su uso) que nos permite generar selecciones pseudo-aleatorias\* entre valores o secuencias.

Nombre del módulo

**from random import \***

\* = Todos los métodos

*También pueden importarse de manera independiente aquellos a utilizar.*

**randint**(*min, max*): devuelve un **integer** entre dos valores dados (ambos **límites** incluidos)

**uniform**(*min, max*): devuelve un **float** entre un **valor mínimo y uno máximo**

**random**(*sin parámetros*): devuelve un **float** entre 0 y 1

**choice**(*secuencia*): devuelve un **elemento al azar de una secuencia** de valores (listas, tuples, rangos, etc.)

**shuffle**(*secuencia*): toma una **secuencia** de valores mutable (como una lista), y la retorna **cambiando el orden de sus elementos aleatoriamente**.

*\* La mecánica en cómo se generan dichos valores aleatorios viene en realidad predefinida en "semillas". Si bien sirve para todos los usos habituales, no debe emplearse con fines de seguridad o criptográficos, ya que son vulnerables.*

# min() & max()

La función `min()` retorna el item con el valor más bajo dentro de un iterable. La función `max()` funciona del mismo modo, devolviendo el valor más alto del iterable. Si el iterable contiene strings, la comparación se realiza alfabéticamente.

```
ciudades_habitantes = {"Tijuana":1810645, "León":1579803}
```

```
lista_valores = [5**5, 12**2, 3050, 475*2]
```

```
print(min(ciudades_habitantes.keys()))  
>> León
```

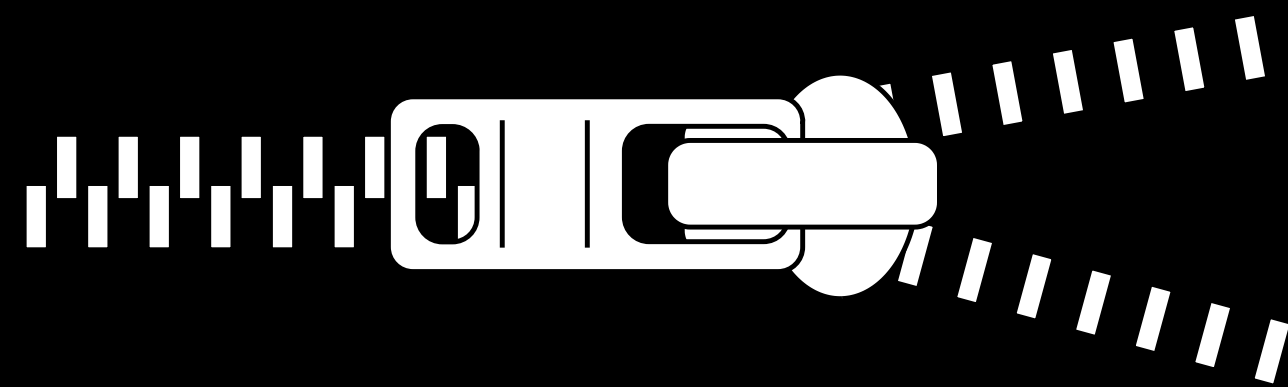
```
print(max(ciudades_habitantes.values()))  
>> 1810645
```

```
print(max(lista_valores))  
>> 3125
```



# zip()

La función `zip()` crea un iterador formado por los elementos agrupados del mismo índice provenientes de dos o más iterables. Zip deriva de *zipper* (cremallera o cierre), de modo que es una analogía muy útil para recordar.



La función se detiene al cuando se agota el iterable con menor cantidad de elementos.

```
letras = ['w', 'x', 'c']
numeros = [50, 65, 90, 110, 135]
for letra, num in zip(letras, numeros):
    print(f'Letra: {letra}, y Número: {num}')
```

```
>> Letra: w, y Número: 50
>> Letra: x, y Número: 65
>> Letra: c, y Número: 90
```

# enumerate()

La función `enumerate()` nos facilita llevar la cuenta de las iteraciones, a través de un contador de índices de un iterable, que se puede utilizar de manera directa en un loop, o convertirse en una lista de tuplas con el método `list()`.

Cualquier objeto que pueda ser iterado



```
enumerate(iterable, inicio)
```

Valor [int] de inicio del índice (por defecto iniciado en 0)

```
print(list(enumerate("Hola")))
>> [(0, 'H'), (1, 'o'), (2, 'l'), (3, 'a')]
```

```
for indice, numero in enumerate([5.55, 6, 7.50]):
    print(indice, numero)
```

```
>> 0 5.55
>> 1 6
>> 2 7.5
```

# range()

La función `range()` devuelve una secuencia de números dados 3 parámetros. Se utiliza fundamentalmente para controlar el número de ejecuciones de un loop o para crear rápidamente una serie de valores.

número a partir del cual  
inicia el rango (**incluido**)

diferencia entre cada valor  
consecutivo de la secuencia



```
range(valor_inicio, valor_final, paso)
```

número antes del cual el  
rango finaliza (**no incluido**)

```
print(list(range(1,13,3)))  
>> [1,4,7,10]
```

*El único parámetro obligatorio es `valor_final`. Los valores predeterminados para `valor_inicio` y `paso` son 0 y 1 respectivamente.*

# loops while

Si bien los **loops while** son otro tipo de bucles, resultan más parecidos a los **condicionales if** que a los **loops for**. Podemos pensar a los loops while como una estructura condicional que se ejecuta en repetición, hasta que se convierte en falsa.

Estructura **condicional**

la **indentación**  
es obligatoria en Python

```
while condición:  
    expresión  
else:  
    expresión
```

dos puntos (:)

Este código se ejecutará cuando la condición se convierta en **False**

## instrucciones especiales

Si el código llega a una instrucción **break**, se produce la **salida** del bucle.

La instrucción **continue** **interrumpe** la **iteración actual** dentro del bucle, llevando al programa a la parte superior del bucle.

La instrucción **pass** **no altera el programa**: ocupa un lugar donde se espera una declaración, pero no se desea realizar una acción.

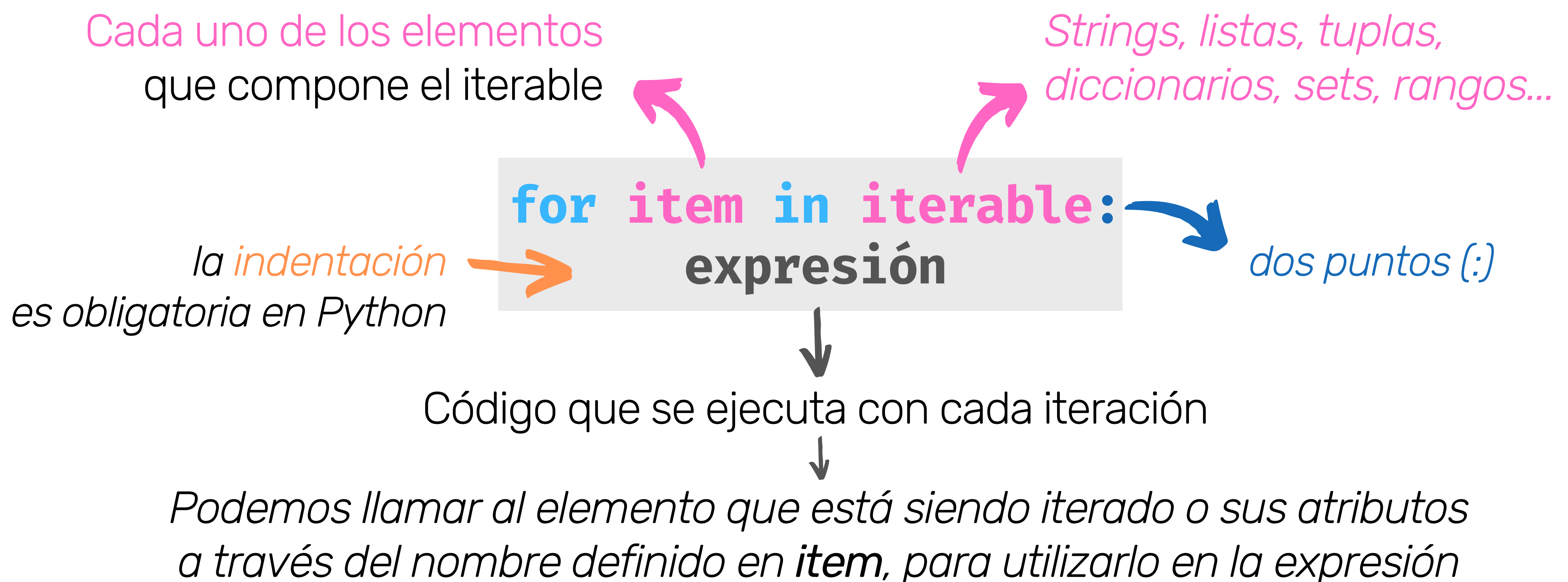


# loops for

A diferencia de otros lenguajes de programación, los **loops for** en Python tienen la capacidad de iterar a lo largo de los elementos de cualquier secuencia (listas, strings, entre otros), en el orden en que dichos elementos aparecen.

## Conceptos

- **Loops/bucles:** son secuencias de instrucciones de código que se ejecutan repetidas veces
- **Iterables:** son objetos en Python que se pueden recorrer (o iterar) a lo largo de sus elementos



# control de flujo

El control de flujo determina el orden en que el código de un programa se va ejecutando. En Python, el flujo está controlado por **estructuras condicionales**, **loops** y **funciones**.

## estructuras condicionales (if)

*Expresión de resultado booleano  
(True/False)*

*Los dos puntos (:) dan paso al código  
que se ejecuta si expresión = True*

*la indentación  
es obligatoria  
en Python*

```
if expresión:
    código a ejecutarse
elif expresión:
    código a ejecutarse
elif expresión:
    código a ejecutarse
...
else:
    código a ejecutarse
```

*else & elif son  
opcionales*

*pueden incluirse  
varias cláusulas elif*

# operadores lógicos

Estos operadores permiten tomar decisiones basadas en múltiples condiciones.

`a = 6 > 5`

`b = 30 == 15*3`

**and** devuelve True si todas las condiciones son verdaderas

```
mi_bool = a and b  
print(mi_bool)  
>> False
```

**or** devuelve True si al menos una condición es verdadera

```
mi_bool = a or b  
print(mi_bool)  
>> True
```

**not** devuelve True si el valor del booleano es False, y False si es True

```
mi_bool = not a  
print(mi_bool)  
>> False
```

# operadores de comparación

Como su nombre lo indica, sirven para comparar dos o más valores. El resultado de esta comparación es un booleano (True/False)

==	igual a
!=	diferente a
>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que

Si la comparación resulta verdadera, devuelve el resultado **True**

Si dicha comparación es falsa, el resultado es **False**

```
mi_bool = 5 >= 6  
print(mi_bool)  
>> False
```

```
mi_bool = 5 != 6  
print(mi_bool)  
>> True
```

```
mi_bool = 10 == 2*5  
print(mi_bool)  
>> True
```

```
mi_bool = 5 < 6  
print(mi_bool)  
>> True
```