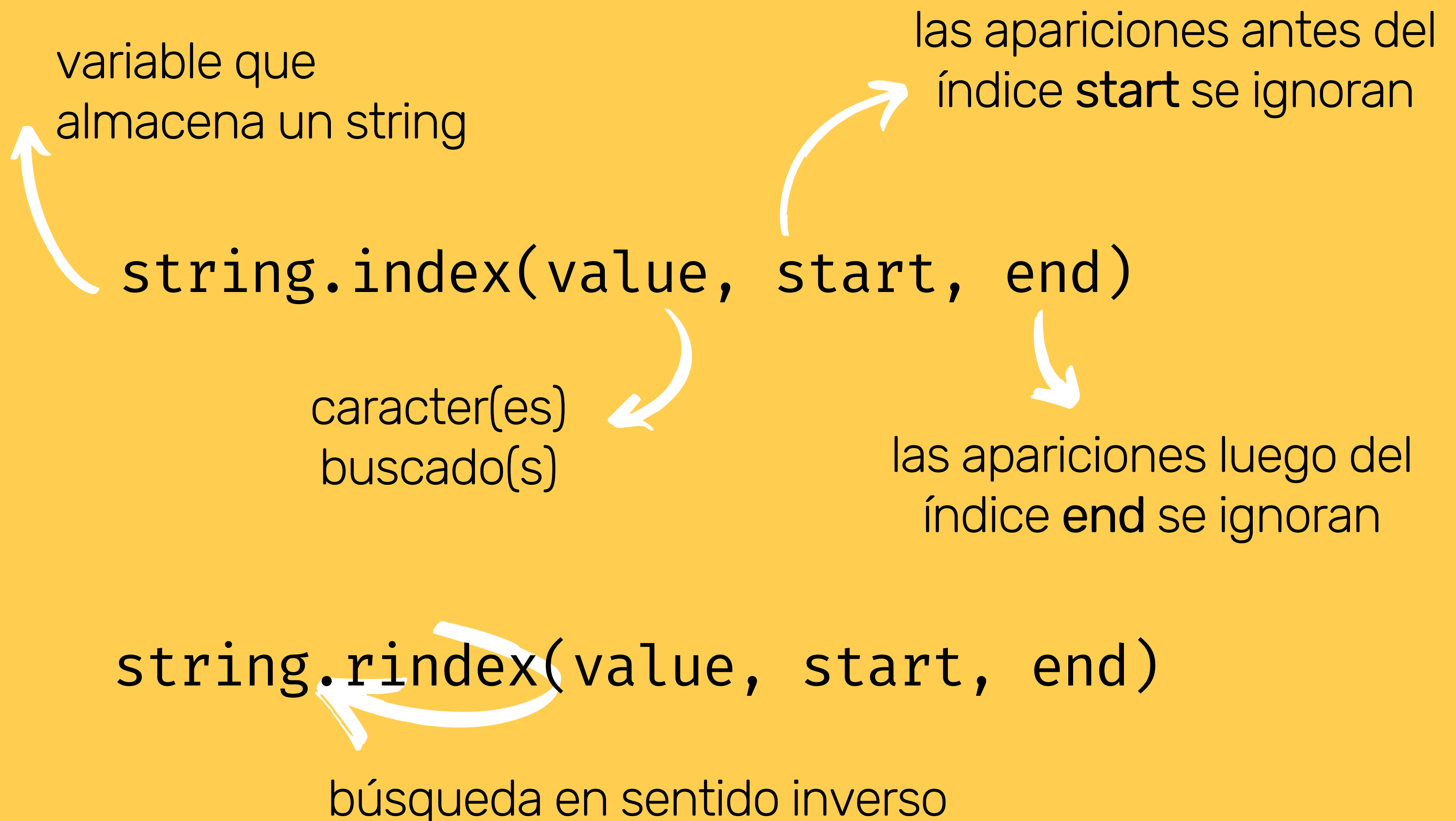


# index()

Utilizamos el método `index()` para explorar strings, ya que permite hallar el índice de aparición de un carácter o cadena de caracteres dentro de un texto dado.



`string[i]` → devuelve el carácter en el índice `i`\*

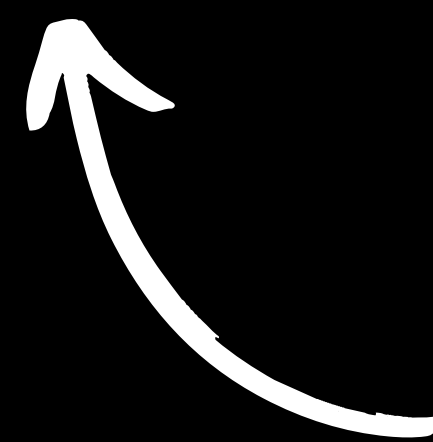
*\*: En Python, el índice en primera posición es el 0*

# input

Función que permite al usuario introducir información por medio del teclado al ejecutarse, otorgándole una instrucción acerca del ingreso solicitado. El código continuará ejecutándose luego de que el usuario realice la acción.

```
input("Dime tu nombre: ")
```

```
>> Dime tu nombre: |
```



Ingreso por teclado

```
print("Tu nombre es " + input("Dime tu  
nombre: "))
```

```
>> Dime tu nombre: Federico
```

```
>> Tu nombre es Federico
```

# integers & floats

Existen dos tipos de datos numéricos básicos en Python: int y float. Como toda variable en Python, su tipo queda definido al asignarle un valor a una variable. La función `type()` nos permite obtener el tipo de valor almacenado en una variable.

## int

Int, o integer, es un número entero, positivo o negativo, sin decimales, de un largo indeterminado.

```
num1 = 7
print(type(num1))
>> <class 'int'>
```

## float

Float, o "número de punto flotante" es un número que puede ser positivo o negativo, que a su vez contiene una o más posiciones decimales.

```
num2 = 7.525587
print(type(num2))
>> <class 'float'>
```

# interacción entre funciones

Las salidas de una determinada función pueden convertirse en entradas de otras funciones. De esa manera, cada función realiza una tarea definida, y el programa se construye a partir de la interacción entre funciones.

```
def funcion_1():  
    ...  
    return a  
  
def funcion_2(a):  
    ...  
    return b  
  
def funcion_3(b):  
    ...  
    return c  
  
def funcion_4(a, c):  
    ...  
    return d
```

# listas

Las listas son secuencias ordenadas de objetos. Estos objetos pueden ser datos de cualquier tipo: strings, integers, floats, booleanos, listas, entre otros. Son tipos de datos mutables.

mutable ✓

ordenado ✓

<sup>admite</sup>  
duplicados ✓

```
lista_1 = ["C", "C++", "Python", "Java"]  
lista_2 = ["PHP", "SQL", "Visual Basic"]
```

**indexado:** podemos acceder a los elementos de una lista a través de sus índices [inicio:fin:paso]

```
print(lista_1[1:3])  
>> ["C++", "Python"]
```

**cantidad de elementos:** a través de la propiedad len()

```
print(len(lista_1))  
>> 4
```

**concatenación:** sumamos los elementos de varias listas con el símbolo +

```
print(lista_1 + lista_2)  
>> ['C', 'C++', 'Python', 'Java', 'PHP', 'SQL', 'Visual Basic']
```



# listas

```
lista_1 = ["C", "C++", "Python", "Java"]  
lista_2 = ["PHP", "SQL", "Visual Basic"]  
lista_3 = ["d", "a", "c", "b", "e"]  
lista_4 = [5, 4, 7, 1, 9]
```

**función append()**: agrega un elemento a una lista *en el lugar*

```
lista_1.append("R")  
print(lista_1)  
>> ["C", "C++", "Python", "Java", "R"]
```

**función pop()**: elimina un elemento de la lista dado el índice, y *devuelve* el valor quitado

```
print(lista_1.pop(4))  
>> "R"
```

**función sort()**: ordena los elementos de la lista *en el lugar*

```
lista_3.sort()  
print(lista_3)  
>> ['a', 'b', 'c', 'd', 'e']
```

**función reverse()**: invierte el orden de los elementos *en el lugar*

```
lista_4.reverse()  
print(lista_4)  
>> [9, 1, 7, 4, 5]
```

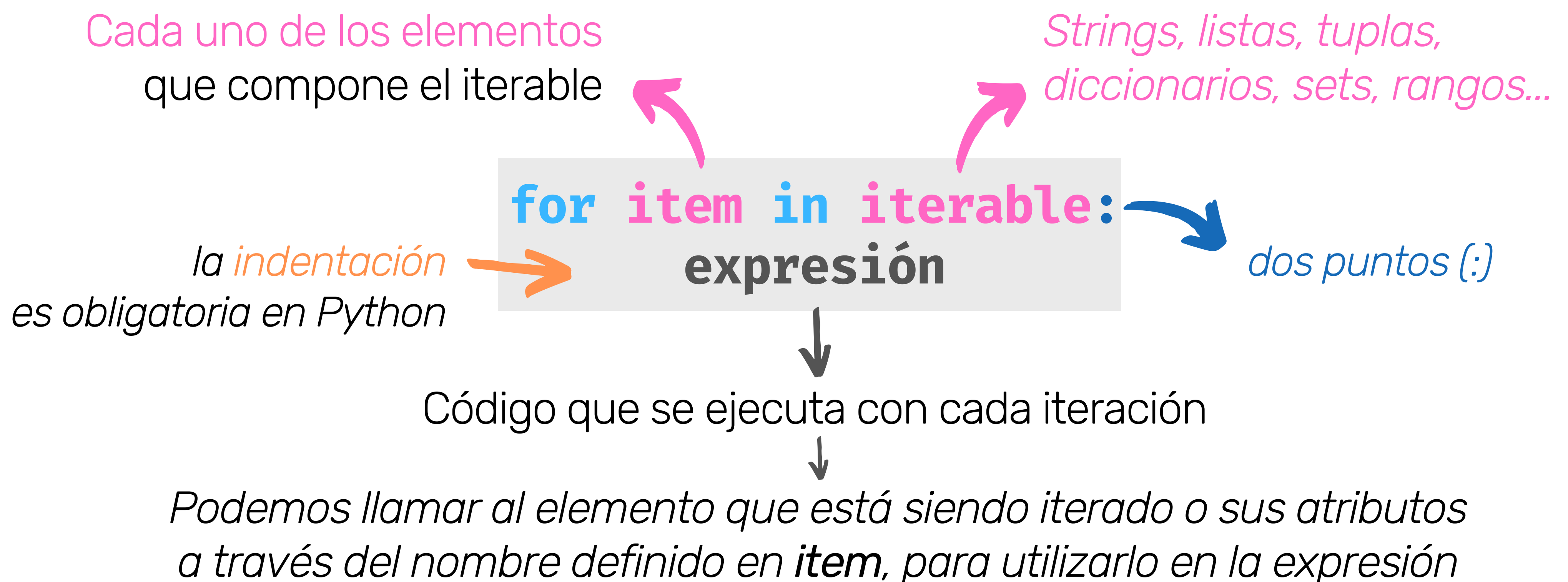
➡ reverse no es lo opuesto a sort

# 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



# 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.



# nombres de variables

Existen convenciones y buenas prácticas asociadas al nombre de las variables creadas en Python. Las mismas tienen la intención de facilitar la interpretabilidad y mantenimiento del código creado.

## reglas

1. **Legible:** nombre de la variable es relevante según su contenido
2. **Unidad:** no existen espacios (puedes incorporar guiones bajos)
3. **Hispanismos:** omitir signos específicos del idioma español, como tildes o la letra ñ
4. **Números:** los nombres de las variables no deben empezar por números (aunque pueden contenerlos al final)
5. **Signos/símbolos:** no se deben incluir : " ' , < > / ? | \ ( ) ! @ # \$ % ^ & \* ~ - +
6. **Palabras clave:** no utilizamos palabras reservadas por Python

# booleanos

Los booleanos son tipos de datos binarios ([True/False](#)), que surgen de operaciones lógicas, o pueden declararse explícitamente.

## operadores lógicos

`==` igual a

`!=` diferente a

`>` mayor que

`<` menor que

`>=` mayor o igual que

`<=` menor o igual que

`and` y ([True](#) si dos declaraciones son [True](#))

`or` o ([True](#) si al menos una declaración es [True](#))

`not` no (invierte el valor del booleano)

# 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]
```

# 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]
```



# 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*



# conversiones

Python realiza conversiones implícitas de tipos de datos automáticamente para operar con valores numéricos. En otros casos, necesitaremos generar una conversión de manera explícita.

```
int(var)  
>> <class 'int'>
```



Convierte el dato en integer

```
float(var)  
>> <class 'float'>
```



Convierte el dato en float

# diccionarios

Los diccionarios son estructuras de datos que almacenan información en pares **clave:valor**. Son especialmente útiles para guardar y recuperar información a partir de los nombres de sus claves (no utilizan índices).

mutable ✓

ordenado ✗ \*

admite duplicados ✗: ✓  
valor  
clave

```
mi_diccionario = {"curso": "Python TOTAL", "clase": "Diccionarios"}
```

agregar nuevos datos, o modificarlos

```
mi_diccionario["recursos"] = ["notas", "ejercicios"]
```

acceso a valores a través del nombre de las claves

```
print(mi_diccionario["recursos"][1])  
>> "ejercicios"
```

métodos para listar los nombres de las **claves**, **valores**, y pares **clave:valor**

keys() ←

↓  
values()

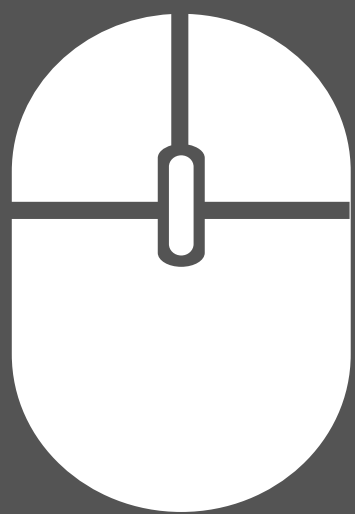
→ items()

*\*: A partir de Python 3.7+, los diccionarios son tipos de datos ordenados, en el sentido que dicho orden se mantiene según su orden de inserción para aumentar la eficiencia en el uso de la memoria.*

# documentación

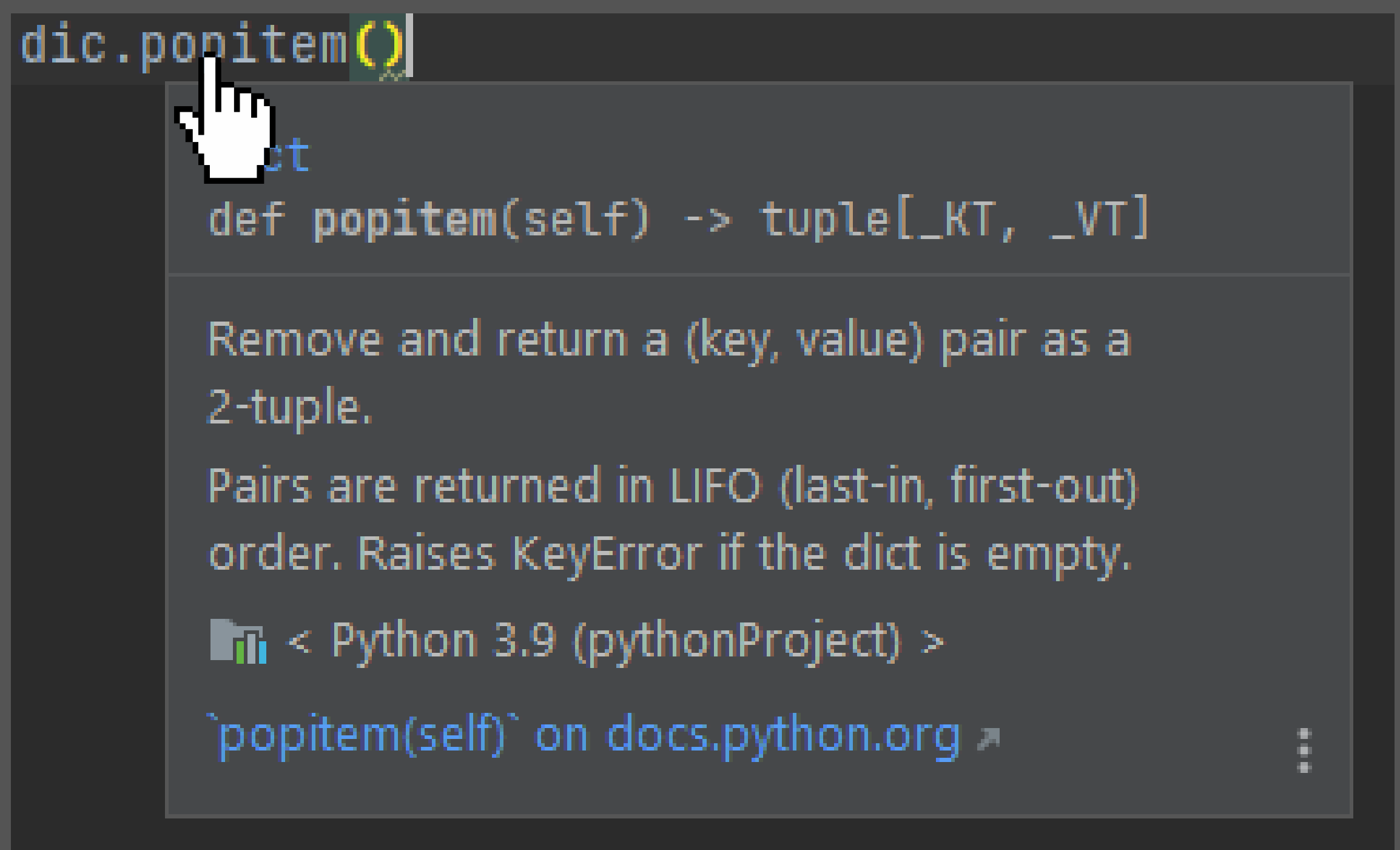
La documentación es nuestra biblioteca de consulta en Python. Al escribir código, es de uso permanentemente para solucionar dudas relacionadas al funcionamiento de métodos y los argumentos que reciben.

Si utilizas **PyCharm**:



Sostén el cursor sobre el nombre del método del que deseas obtener información. Se desplegará una ventana flotante.

Esto funcionará en otros IDEs del mismo modo.



También, debes mantener cerca la **documentación oficial** de Python (o **Biblioteca Estándar de Python**), que contiene toda la información necesaria:

<https://docs.python.org/es/3.9/library/index.html>

*No dejes de buscar en Google tus dudas, para hallar una explicación que se ajuste a ti.*

# 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
```



# funciones

Una función es un bloque de código que solamente se ejecuta cuando es llamada. Puede recibir información (en forma de *parámetros*), y devolver datos una vez procesados como resultado.

una función es definida  
mediante la palabra clave `def`



```
def mi_funcion(argumento):
```

Los argumentos contienen información que la función utiliza o transforma para devolver un resultado

```
mi_funcion(mi_argumento)
```

Para llamar a una función, basta utilizar su nombre, entregando los argumentos que la misma requiere entre paréntesis.



# funciones dinámicas

La integración de diferentes herramientas de control de flujo, nos permite crear funciones dinámicas y flexibles. Si debemos utilizarlas varias veces, lograremos un programa más limpio y sencillo de mantener, evitando repeticiones de código.

- Funciones
- Loops (for/while)
- Estructuras condicionales
- Palabras clave (return, break, continue, pass)

```
def mi_funcion(argumento):  
    for item in ...  
        if a == b ...  
            ...  
        else:  
            return ...  
    return ...
```

# funciones dinámicas

La integración de diferentes herramientas de control de flujo, nos permite crear funciones dinámicas y flexibles. Si debemos utilizarlas varias veces, lograremos un programa más limpio y sencillo de mantener, evitando repeticiones de código.

- Funciones
- Loops (for/while)
- Estructuras condicionales
- Palabras clave (return, break, continue, pass)

```
def mi_funcion(argumento):  
    for item in ...  
        if a == b ...  
            ...  
        else:  
            return ...  
    return ...
```