Palabras reservadas

Palabras utilizadas para especificar sentencias, operadores, tipos de datos...

False	True	None	break	pass	return	raise	await	continue	def	del
if	and	or	not	is	global	else	lambda	nonlocal	for	as
while	assert	in	from	class	import	match	elif	finally	try	yield

El uso de estas expresiones para definir variables u otros usos devolverá un error de sintaxis

Índices

Equivalen a la posición de un elemento en una lista, tupla, cadena ...

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
  0   1   2   3   4   5
 -6  -5  -4  -3  -2  -1
```

Operadores

Operadores relacionales

Operador	Sigfinicado	Ejemplo
>	mayor que	5 > 3 = / True
<	menor que	2 < 4 = / True
==	igual que	2 == 1 / False
>=	mayor o igual	4 >= 4 / True
<=	menor o igual	3 <= 2 / False
!=	distinto que	3 != 5 / True

Operadores matemáticos

```
> 2 + 3      # suma
> 5 - 7      # resta
> 5 * 3      # multiplicacion
> 8 / 3      # division en punto flotante
> 8.0 // 3   # division entera
> 3 ** 2     # exponente
> 11 % 4     # modulo (resto de la division entera)
> 5 - 2 * 3  # operaciones encadenadas
> (5 - 2) * 3  # prioridad con los paréntesis
```

Variables

```
# CONSTANTE : variable declarada en mayúscula y que no va ha ser modificada ( por convenio )
PI = 3.141592

# Variable
nombre = "Pablo"  # String
edad = 18         # Entero
```

Tipos de variables

```
A = 43
print(type(A)) # Tipo de la variable A ( <class 'int'> )

> "Hola"  # String
> 5       # Entero
> 5.25    # Decimal
> True    # Booleano
> 3 + 4j  # Complejo
```

Type Casting

```
# Convertir el tipo de dato de una variable en otro
a = str(5)    # Texto
b = int(5)    # Entero
c = float(5)  # Decimal
```

Entrada y salida de datos

```
# Entrada
nombre = input("Cómo te llamas?: ") # siempre va a almacenar una cadena de texto

# Salida
print(nombre)

print("Me llamo " + nombre)
print(f"Me llamo {nombre}") # f-string (formato moderno)
```

Podemos realizar operaciones, recibir datos de funciones y mostrarlos directamente sin tener que usar una variable

```
def funcion():
    return "Hola"

print(funcion())
>> "Hola"
print( 2 + 4 ) # 6
```

Estructuras de datos

Listas

- *Pueden contener elementos de diferentes tipos y duplicados
- *Sus elementos tienen un orden
- *Pueden ser indexadas y cortadas*
- *Una lista puede contener otras listas como elementos.*

```
lista = [ "Albacete" , "Ciudad Real", "Castellón "]
lista2 = list(("Lunes", "Martes", "Miércoles")) # otra forma de crear una lista
```

```
> len(lista)      # longitud de la lista ( número de elementos)
> lista[2]        # acceder a un elemento ( posición 2 )
> lista[1] = "Cuenca"  # cambia el elemento en la posición 1

> lista.index(valor)   # posición de un elemento
> lista.count(valor)   # cuántas veces aparece un elemento
```

Añadir elementos

```
> lista.insert(2 , "Pamplona") # introduce en la posición dos un nuevo elemento

> lista.append("Lugo")         # añade un elemento al final de la lista

> lista.extend(otra_lista)     # añade elementos de otra lista a la actual. Pueden ser otros el
```

Eliminar elementos

```
> lista.remove("Castellón") // elimina el elemento indicado ( Si hay duplicados, elimina el pri

> lista.pop(1)   # elimina el elemento en esa posición o, por defecto, el último

> del lista[3]   # también elimina el elemento en esa posición

> del lista      # borra la lista ( Sus elementos y ella misma )

> lista.clear()  # Borra los ELEMENTOS de la lista
```

Otras funciones

```
> lista.reverse() / lista [::-1] # invertir el orden

> lista.sort() # ordenar la lista
```

Tuplas

- *No podemos modificar, añadir o retirar elementos una vez creadas*
- *Sus elementos tienen un orden*
- *Admiten varios tipos de datos a la vez y duplicados*

```
tupla = ( "Pedro" , 32 , True , 4.6 )
tupla2 = tuple(("Lunes", 12 , "Montaña")) # otra forma de crear una lista
```

```
> len(tupla)  # longitud de la lista ( número de elementos)
> tupla[1]    # acceder a un elemento ( posición 2 )

> del tupla   # borra la tupla
```

Como las tuplas son **inmutables**. La forma de modificarlas será:

1. Convirtiéndola en una lista -> modificarla -> convertir la lista en una tupla
2. Sumándole otra tupla

```
lista = list(tupla)
> lista.append(5)
tupla = tuple(lista)

tupla1 += tupla2
```

Las tuplas permiten asignar y desasignar sus elementos a múltiples variables de forma simultánea.

```
tupla = ( 1, 2, 3, 4 )
a, b , c , d = tupla    #a= 1, b= 2, c= 3, d= 4
```

Si el **número de variables es menor que el número de valores**, puedes añadir un `*` al nombre de la variable y los valores se asignarán a la variable como una lista

```
tupla = ( 1, 2, 3, 4 )
a , b , *c = tupla  # a = 1, b = 2, c = (3 , 4)
d , *e , f = tupla  # d = 1 e = (2 , 3) f = 4
```

```
tupla = ( 1 , 2 )
tupla *= 2
>> ( 1 , 2 , 1 , 2)

> tupla.index(valor)  # encontrar la posición de un elemento
> tupla.count(valor)  # cuantas veces aparece un elemento
```

Conjuntos

- *No permiten duplicados pero si distintos tipos de datos*
- *Sus elementos no tienen un orden*
- *Una vez creado, no puedes cambiar sus valores, pero puedes añadir o eliminar elementos*

! **True es considerado como 1 y False como 0 , por lo que se les tratará cómo duplicados** ¡

```
conjunto = { 1 , 2 , 3 }

conjunto = set(( 1 , 2 , 3 ))

> len(conjunto) # número de elementos
```

Añadir elementos

```
> conjunto.add(4)             # añadir un valor

> conjunto1.update(conjunto2) # añadir un conjunto ( o lista, tupla ... ) a otro.
# Esto no crea un nuevo set, solo lo actualiza
```

Eliminar elementos

```
> conjunto.remove(1)   # si el elemento no está en el conjunto, devuelve error

> conjunto.discard(2)  # si el elemento no está en el conjunto, NO devuelve error

> conjunto.pop()       # elimina un elemento aleatorio | Permite almacenar el valor borrado

> conjunto.clear()     # borra todos sus elementos

> del conjunto         # borra el conjunto
```

Operaciones con conjuntos

Unión

Devuelve un tercer set con los elementos de ambos

```
> set1.union(set2)                # devuelve un nuevo conjunto
> set1.union(set2 , set3 , set4)  # permite unir múltiples conjuntos

set3 = set1 | set2 # el operador "|" realiza la misma operación
```

También se pueden unir un conjunto y una tupla

Intersección

Mantiene sólo los elementos comunes

```
> set3 = set1.intersection(set2)  # devuelve un nuevo set

set3 = set1 & set2                # sólo permite unir conjuntos

> set1.intersection_update(set2)  # realiza la intersección pero no devuelve un nuevo set, solo
```

Diferencia

Mantiene los elementos del primero que no estén en el segundo.

```
> set3 = set1.difference(set2)  # devuelve un nuevo set

set3 = set1 - set2              # dólo permite operar conjuntos

> set1.difference_update(set2)  # realiza la diferencia pero no devuelve un nuevo set, solo lo
```

Diferencia simétrica

Mantiene los elementos que no estén en ambos

```
> set3 = set1.symmetric_difference(set2)  # devuelve un nuevo set

set3 = set1 ^ set2                        # sólo permite operar conjuntos

> set1.symmetric_difference_update(set2)  # realiza la diferencia pero no devuelve un nuevo set
```

Otros métodos

Metodo	Atajo	Descripción
copy()		Devuelve una copia del conjunto
isdisjoint()		Devuelve si dos conjuntos tienen intersección o no
issubset()	`<=`	Devuelve `True` si todos los elementos del set también pertenecen a otro set
	`<`	Devuelve `True` si todos los elementos del set están presentes en otro, más grande
issuperset()	`>=`	Devuelve `True` si todos los elementos de otro set están presentes en este
	`>`	Devuelve `True` si todos los elementos de otro set, más pequeño, están en este

---

Diccionarios

Los diccionarios se utilizan para almacenar valores de datos en pares clave-valor

- *Desordenados*
- ***Dinámicos***: Se pueden agregar, modificar y eliminar pares clave-valor.
- *Claves Únicas (no permite duplicados)*
- *Valores Accesibles*

```
dict = {
    "clave1" : "valor1"
    "clave2" : 2
}
print(len(thisdict))
```

Acceder a los datos

En los diccionarios, usamos las claves para acceder a su valor asociado

```
print(dict["clave1"])    # aportando una clave existente, nos devuelve su valor
x = dict.get()"clave1")  # mismo resultado

> dict.keys()    # devueve una lista de todas las claves

> dict.values()  # devueve una lista de todas los valores

> dict.items()   # devuelve los pares clave-valor como tuplas dentro de una lista
```

Añadir elementos

```
dict[ "clave3" ] = "valor3"   # crea un nuevo elemento en el diccionario
dict[ "clave2" ] = 4          # cambia el valor

> dict.update({"clave1": "valor nuevo"})  # cambia el valor
```

Eliminar elementos

```
> dict.pop("clave1")   # elimina el elemento con la clave

> del dict["clave2"]   # elimina el elemento con la clave

> del dict             # elimina el diccionario
```

Diccionarios anidados

```
hijo1 = {
  "nombre" : "Emil",
  "año" : 2004
}
hijo2 = {
  "nombre" : "Tobias",
  "año" : 2007
}

familia = {
  "hijo1" : hijo1,
  "hijo2" : hijo2,
}
familia["hijo2"]["nombre"] # acceder a un valor de un diccionario anidado
```

Cadenas de texto

```
texto = "Hola, que tal"
len(texto) # número de caracteres
texto[1] # devuelve el caracter en la posición x

> texto.lower()    # todo en minúsculas
> texto.upper()    # todo en mayúsculas

> texto.strip()    # elimina espacios

> texto.capitalize()  # la primera mayúscula
> texto.title()       # la primera de cada palabra en mayúscula
```

```
> texto.split(",")   # convierte la cadena en una lista de caracteres

> texto.join("-")    # une una lista de caracteres en una cadena separada por el argumento

> texto.replace("H","G")  # remplaza un caracter por otro
```

```
> texto.find("q")    # devuele la posición del primer elemento coincidente
> texto.index("l")   # devuele la posición del primer elemento coincidente

> texto.count("a")   # número de veces que aparece el elemento

> texto.startswith("h")
# verifican si empieza / acaba por ese caracter
> texto.endswith("l")
```

- Más métodos: https://www.w3schools.com/python/python_strings_methods.asp

F-Strings

Permiten formatear cadenas de texto de forma sencilla. Para representar información:

```
nombre = "Juan"
edad = 23

print(f"{nombre} tiene {edad} años")
>> "Juan tiene 23 años"
```

```
num1 = 32
num2 = 4

print(f"El resultado es: {num1 / num2}")
>> "El resultado es 8"
```

```
pi = 3.1415
print(f"{pi:.2}")   # 2 cifras en total
print(f"{pi:.3f}")  # 3 cifras decimales
```

Caracter	Resultado	
`\'`	Comilla simple	
`\\`	Barra invertida	
`\n`	Nueva linea	
`\r`	Retorno de carro	
`\t`	Tabulación	

Condicionales

Ejecutan una instrucción si se cumple una o más condiciones:

Es igual a	**a==b**
No es igual a	a!=b
Menor que	a < b
Menor/igual	a <= b
Mayor que	a > b
Mayor/igual	a >= b

If ... Else

Si se cumple la condición, ejecuta el código siguiente

```
a = 33
b = 200
if b > a:
  print("b es mayor que a")
```

`elif`

`Elif` en python significa: "Si no se cumplió la condición anterior, prueba esta:"

```
a = 33
b = 33
if b > a:
  print("b es mayor que a")
elif a == b:
  print("a y b son iguales")
```

`else`

`Else` captura todo lo que no sea capturado por las condiciones anteriores.aboga

```
a = 200
b = 33
if b > a:
  print("b es bayor que a")
elif a == b:
  print("a y b son iguales")
else:
  print("a es mayor que b")
```

If abreviado

Si solo tienes una instrucción que ejecutar, puedes colocarla en la misma línea que la instrucción if.

```
if a > b: print("a es bayor que b")
```

If ... Else abreviado

```
a = 2
b = 330
print("A") if a > b else print("B")
```

If anidado

```
x = 41

if x > 10:
  print("Mayor que 10 ")
  if x > 20:
    print("También mayor que 20!")
  else:
    print("Pero no mayor que 20.")
```

Match

La instrucción `match` se utiliza para realizar diferentes acciones en función de diferentes condiciones.

```
dia = 4
match dia:
  case 1:
    print("Lunes")
  case 2:
    print("Martes")
  case 3:
    print("Miercoles")
  case 4:
    print("Jueves")
...
```

Utiliza el carácter `_` como último valor se ejecute un bloque de código cuando no haya otras coincidencias:

Combinar valores

Puedes utilizar el carácter de barra vertical `|` como operador «o» si hay más de un valor coincidente en un caso:

```
dia = 4
match dia:
  case 1 | 2 | 3 | 4 | 5:
    print("Semana")
  case 6 | 7:
    print("Fin de semana")
```

Sentencias «if» como guardias

Puedes añadir sentencias `if` en la evaluación del caso como una comprobación de condición adicional:

```
mes = 5
dia = 4
match dia:
  case 1 | 2 | 3 | 4 | 5 if mes == 4:
    print("Semana de abril")
  case 1 | 2 | 3 | 4 | 5 if mes == 5:
    print("Semana de mayo")
  case _:
    print("Nada")
```

Bucles

Bucle While ...

Con `while` podemos ejecutar un conjunto de instrucciones siempre que una condición sea verdadera.

```
i = 0
while i < 6:
  print(i)
  i += 1
```

`break`

Con la instrucción `break` podemos detener el bucle incluso si la condición while es verdadera:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

`continue`

Con la instrucción `continue` podemos detener la iteración actual y continuar con la siguiente:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

`else`

Con la instrucción `else` podemos ejecutar un bloque de código una vez que la condición ya no sea verdadera:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Bucle For ...

Un bucle `for` se utiliza para iterar sobre una secuencia (ya sea una lista, una tupla, un diccionario, un conjunto o una cadena). Funciona más como un método iterador

Con el bucle `for` podemos ejecutar un conjunto de sentencias, una vez por cada elemento de una lista, tupla, conjunto, etc.

```
fruits = ["manzana", "banana", "pera"]
for x in fruits:
  print(x)
```

Even strings are iterable objects, they contain a sequence of characters:

```
for x in "banana":
  print(x)
```

Con la instrucción `continue` podemos detener la iteración actual y continuar con la siguiente
Con la instrucción `break` podemos detener el bucle incluso si la condición while es verdadera

`range`

Para recorrer un conjunto de código un número determinado de veces, podemos utilizar la función `range()`.

La función `range()` devuelve una secuencia de números, que comienza por defecto en 0, se incrementa por defecto en 1 y termina en un número especificado.

```
for x in range(6):
  print(x) # 0 , 1 , 2 , 3 , 4 , 5

#for x in range(inicio, final ( -1 ), pasos ):
```

`else`

El bucle `else` especifica un bloque de código que se ejecutará cuando finalice el bucle:

```
for x in range(6):
  print(x)
else:
  print("Terminado!")
```

Bucles anidados

```
adje = [ 1, 2, 3]
frutas = ["manzana", "banana", "fresa"]

for x in adj:
  for y in fruits:
    print(x, y)
```

Funciones

Una función es un bloque de código que solo se ejecuta cuando se invoca.
Se pueden pasar datos, conocidos como parámetros, a una función. Una función puede devolver datos como resultado.

```
def saludar()
    print("Hola"

saludar() # llamar a la función
```

`*args`

Si no sabes cuántos argumentos se pasarán a tu función, añade un `*` delante del nombre del parámetro en la definición de la función.

De esta forma, la función recibirá una **tupla* de argumentos y podrá acceder a los elementos correspondientes:

```
def funcion(*kids):
  print("Mis hijos son " + kids)

funcion("Jairo", "Pedro", "Pablo")
```

`Argumentos clave`

También se puede enviar argumentos con la sintaxis clave = valor.
De esta forma, el orden de los argumentos no importa.

```
def funcion(iva):
  print(f"Precio con iva: {precio*iva}")

funcion(iva = 0.21 )
```

`**kwargs`

Si no sabes cuántos argumentos clave se pasarán a tu función, añade dos asteriscos: `**` antes del nombre del parámetro en la definición de la función.
De esta forma, la función recibirá un **diccionario** de argumentos y podrá acceder a los elementos correspondientes:

```
def funcion(**producto):
    print(f"El precio de {producto["nombre"]} es {producto["precio"]} euro")

funcion(nombre = "lápiz", precio = 1 )
```

`Valor predeterminado`

El siguiente ejemplo muestra cómo utilizar un valor predeterminado del parámetro.
Si llamamos a la función sin argumentos, se utiliza el valor predeterminado:

```
def funcion(pais = "Noruega"):
  print(f"Soy de {pais}")

funcion("Suecia")
funcion()
```

`return`

Para que la función devuelva un valor, que podemos usar en otros contextos:

```
def funcion(x):
  return 5 * x

print(funcion(3))       # 15
resultado = funcion(4)  # resultado = 20
```

También podemos especificar que en una función solo se acepten argumentos posicionales o clave:

- Con `/`, los argumentos anteriores solo podrán ser posicionales.
- Con `*`, los argumentos posteriores solo podrán ser clave (`kwargs`)

```
def funcion1(x,/)
    print(x)

funcion(x)

def funcion2(*, y)
    print(y)

funcion2(y = 3)
```

Además, podemos combinarlos:

```
def funcion3(a , b , / , * , x , y)
# a y b posicionales
# x e y kwargs
```

Finalmente, podemos crear una **función recursiva** cuando esta se llama a si misma

---

Range

La función integrada `range()` devuelve una secuencia inmutable de números, que se utiliza habitualmente para realizar un número específico de bucles.
Este conjunto de números tiene su propio tipo de datos denominado `range`.

```
# range( inicio , final (-1), pasos )
x = range(8)
```

Para poder mostrarlos, debemos convertirlo primero en una lista

```
print(list(range(6))) # 0 , 1 , 2 , 3 , 4 , 5
```

Sobre el podemos aplicar funciones típicas como su longitud (`len()`), comprobar si un elemento pertenece a el, iterar sobre el...

Try... except

- El bloque `try` te permite comprobar si hay errores en un bloque de código.
- El bloque `except` te permite gestionar el error.
- El bloque `else` te permite ejecutar código cuando no hay ningún error.
- El bloque `finally` te permite ejecutar código, independientemente del resultado de los bloques try y except.

```
try:
  print(x)
except:
  print("Halgo a pasado")
```

Puedes usar `else` para definir un bloque de código que se ejecutará si no se generaron errores:

```
try:
  print("Hola")
except:
  print("Halgo a pasado")
else:
  print("Todo ha ido bien")
```

El bloque `finally`, si se especifica, se ejecutará independientemente de si el bloque try genera un error o no.

```
try:
  print("Hola")
except:
  print("Halgo a pasado")
finally:
  print("Hasta luego")
```

La palabra clave `raise` se utiliza para generar una excepción. Se puede definir qué tipo de error se va a generar y el texto que se mostrará al usuario.

List Comprehension

Las `List comprehension` ofrece una sintaxis más breve cuando se desea crear una nueva lista basada en los valores de una lista existente.

```
# Listas
nums = [1, 2, 3, 4, 5]
cuadrados = [n**2 for n in nums]
#n**2 -> elementos de la nueva lista
#n -> variable sobre la que crear los nuevos elementos

pares = [n for n in nums if n % 2 == 0] # podemos añadir una condición
```

De forma similar podemos hacerlo con tuplas, sin embargo, el objeto resultante es un generador y no una estructura iterable.

```python
lista = [1, 2, 3, 4, 5]

lista2 = [n**2 for n in lista]   # list comprehension
tupla  = (n**2 for n in lista)   # generator expression

print(lista2)  # [1, 4, 9, 16, 25]
print(tupla)   # <generator object <genexpr> at 0x...>
```

Para obtener los valores, debes transformarlo:

```python
print(list(tupla))  # [1, 4, 9, 16, 25]
```

No guarda todos los resultados, **genera uno a uno bajo demanda**. Mucho más eficiente en memoria para listas grandes.
