Índice

1 INTRODUCCIÓN

Python es un lenguaje de programación relativamente reciente (Guido van Rossum 1991) y muy versátil que permite realizar aplicaciones web en el lado del servidor, conexiones a bases de datos y crear o modificar ficheros, y gestionar big data y desarrollar procesos matemáticos complejos.

Con Python se puede trabajar sobre distintos sistemas operativos, Windows, Mac, Linux, Rasberry Pi, etc.... Tiene una sintaxis sencilla similar al inglés que permite a los programadores desarrollar programas con menos líneas de código que otros lenguajes.

Python es un lenguaje interpretado, lo que significa que se pueden ejecutar sus sentencias tan pronto como estén escritas, sin necesidad de pasar por la compilación. Y con este lenguaje es posible programar tanto en modo estructurado como en modo orientado a objetos.

En cuanto a su sintaxis, Python interpreta cada nueva línea como una nueva instrucción, mientras que en otros lenguajes se utiliza un ";" para separar las distintas sentencias.

También en el caso de Python es muy importante la identación, es decir, utilizar espacios en blanco para definir el alcance de una sentencia, como los bucles, funciones o clases. Esta cuestión, en otros lenguajes de programación se realiza utilizando llaves para delimitar el alcance de las distintas sentencias y funciones "{}".

2 IDENTACIÓN EN PYTHON

Se puede definir como el número de espacios en blanco al inicio de una línea de código dentro del lenguaje.

Mientras que en otros lenguajes de programación la identación se utiliza sólo para hacer el código más legible, en el caso de Python se utiliza para indicar un bloque de código.

El número de espacios a utilizar en la identación depende del gusto del programador, lo más normal, en el caso de que el propio entorno de desarrollo no lo aplique, es utilizar 3 o 4, pero como mínimo debe haber 1 para que no se produzca error.

Dentro del mismo bloque de código es necesario que la identación sea la misma para que no se produzcan errores al escribir el código.

3 COMENTARIOS EN PYTHON

En Python es posible utilizar comentarios para documentar el código a medida que se va desarrollando.

Los comentarios comienzan con el símbolo #, y Python interpreta que lo que va a continuación en la misma línea no se trata de una instrucción a ejecutar.

Los comentarios se utilizan habitualmente para explicar el código, para hacerlo más legible y/o para evitar la ejecución de determinadas líneas de código durante el período de pruebas.

En el caso de que sea necesario comentar varias líneas de código es posible iniciar cada una de ellas con el símbolo #, aunque también es posible utilizar triples comillas dobles para comenzar un comentario de varias líneas, (“””), y la misma cantidad de comillas dobles para finalizar el comentario. Las instrucciones o explicaciones que están incluidas entre unas comillas y otras serán ignoradas para su ejecución.

4 VARIABLES EN PYTHON

Una variable se puede definir como un contenedor para almacenar valores de datos.

En Python no existe un comando específico para crear una variable, sino que esta se crea en el momento en el que se le asigna un valor.

Las variables no necesitan ser declaradas de un tipo en particular, y pueden cambiar de tipo cuando se les pasa a asignar un nuevo valor. Sin embargo, si se quiere especificar el tipo de dato que contiene una variable, esto se puede hacer utilizando “casting”.

Ejemplo: x = int(3)

Si lo que se pretende es obtener el tipo de dato que contiene una variable, para ello se utiliza la función type().

En el caso de las variables tipo cadena o string, se pueden declarar utilizando tanto comillas simples como dobles.

Las variables en Python son sensibles a mayúsculas y minúsculas, es decir, una variable escrita en mayúsculas y en minúsculas, no se trata de la misma, si no de dos variables diferentes.

A una variable no se le puede poner cualquier nombre, los nombres de variables deben seguir unas normas como son:

- Deben comenzar por una letra o por guion bajo _
- No pueden empezar por un número.
- Un nombre de variable sólo puede contener caracteres alfanuméricos y guiones bajos (A – z, 0 – 9, _)
- Son sensibles a mayúsculas y minúsculas (edad, Edad y EDAD, son tres variables diferentes).
- No se pueden utilizar como variables ninguna de las palabras reservadas del lenguaje.

Pueden existir nombres de variables formados por más de una palabra, y para ello es posible utilizar distintas técnicas como, que cada palabra excepto la primera comience por una letra mayúscula; que cada palabra comience por una letra mayúscula; o que cada palabra aparezca separada por un guion bajo. De esta manera, las variables formadas por más de una palabra se hacen más legibles.

Python permite asignar valores a distintas variables y sólo en una línea, para ello, es imprescindible que el número de variables y el de valores sean el mismo o de lo contrario nos dará error.

Ejemplo: x, y, z = “Juan”, “Jorge”, “Luis”

Por otro lado, también es posible asignar el mismo valor a distintas variables en una sola línea:

Ejemplo: x = y = z = 5

Con estas dos premisas, es posible tener una colección de valores en una lista, tupla, conjunto, etc... y extraerlos a variables independientes, es lo que se conoce como desempaquetado.

Ejemplo: frutas = [“manzana”, “naranja”, “plátano”]

x, y, z = frutas

Para sacar variables por pantalla se utiliza la función “print”, que permite pintar varias variables separadas por comas.

Dentro de la función print() es posible imprimir más de una cadena de caracteres (strings) si se utiliza el símbolo +. Lo que se consigue con eso es concatenar las distintas variables que están dentro de la función. Si en lugar de tratarse de strings, se trata de variables numéricas, en ese caso el símbolo + realiza la operación matemática de suma.

Si lo que se desea es combinar la impresión de variables string con variables numéricas, en ese caso no es posible utilizar el símbolo +, ya que dará error.

Se llaman variables globales a aquellas que están definidas fuera de cualquier función y por tanto pueden ser usadas tanto dentro como fuera de las funciones.

Si se crea una variable dentro de una función, con el mismo nombre que otra que se ha definido fuera de la función, la variable definida dentro de la función es una variable local y sólo se puede utilizar dentro de la función, fuera de ella no será conocida. La variable global con el mismo nombre seguirá conservando el valor inicial que se le asignó antes de entrar en la función.

Normalmente, cuando se crea una variable dentro de una función, la variable es local y sólo se puede utilizar dentro de la función. Para crear una variable global dentro de una función se debe utilizar la palabra reservada “global”. También es posible modificar el valor de una variable global dentro de una función utilizando la misma palabra reservada “global”.

5 TIPOS DE DATOS EN PYTHON

Las variables pueden almacenar datos de diferentes tipos, y con los distintos tipos se pueden realizar diferentes acciones.

Python tiene los siguientes tipos de datos por defecto y en las siguientes categorías:

Tipo texto: str

Tipos numéricos: int, float, complex

Tipos de secuencia, de orden: list, tuple, range

Tipo mapping: dict

Tipo conjunto: set, frozenset

Tipo boolean: bool

Tipos binario: bytes, bytearray, memoryview

Tipo none: nonetype

Para obtener el tipo de dato de un objeto se utiliza la función type().

Como se ha comentado anteriormente, en Python el tipo de dato se establece cuando se asigna valor a una variable.

Se puede especificar el tipo de dato de una variable utilizando los siguientes constructores.

- Tuplas y listas: Una tupla es **inmutable**. Se usarán cuando se sepa con seguridad que los datos no van a variar durante el ciclo de vida del programa o cuando se quiera garantizar que se mantendrán constantes.
- Una lista es **mutable**. Se puede añadir o eliminar elementos. Las listas crecen y decrecen durante la vida del programa. Se usarán cuando los datos, por su propia naturaleza, sean susceptibles de variar.
- Range: es un tipo de dato que permite crear series de elementos numéricos.
- El tipo dict asocia claves a valores.
- El tipo set se utiliza para definir conjuntos, es decir, colecciones desordenadas de elementos únicos.
- El tipo frozenset es como el tipo set, pero en este caso está congelado, es decir, los elementos no pueden ser modificados.
- El tipo bytes es una secuencia inmutable de bytes que sólo admite caracteres ASCII. Mientras que el tipo bytearray es un tipo mutable de bytes.

5.1 TIPOS NUMÉRICOS.

Son 3, int, float y complex. Una variable de tipo numérico se crea cuando se le asigna un valor.

Tipo int.

Int o integer es un número entero, positivo o negativo, sin decimales y de longitud ilimitada.

Tipo float.

Float es un número positivo o negativo que contiene uno o más decimales. También puede tener expresión con “E” para representar potencias de 10.

Los números complejos se escriben poniendo una “j” como parte imaginaria.

Conversión entre tipos:

Es posible realizar la conversión entre los distintos tipos numéricos utilizando los siguientes métodos, int(), float(), y complex().

***NO ES POSIBLE CONVERTIR UN NÚMERO COMPLEJO EN CUALQUIER OTRO TIPO DE NUMÉRICO.**

Números aleatorios.

Python no tiene una función random() para generar números aleatorios, pero tiene un módulo que se llama random que se puede utilizar para generarlos.

Para importar módulos utilizamos la palabra reservada import.

5.2 TIPO STRING.

Es posible asignar más de una línea a un string utilizando 3 comillas dobles o simples.

5.2.1 Strings y arrays.

Como en otros lenguajes de programación, los strings en Python son arrays de bytes que representan caracteres Unicode. Sin embargo, en Python no existe un tipo de dato carácter. Este tipo de dato es igual a tener un string de longitud 1. Los corchetes permiten acceder a los elementos de un string de forma independiente.

Para obtener la longitud de un string se utiliza la función len().

Para comprobar si una cadena determinada (palabra o carácter) está incluida en un string se utiliza la palabra reservada **in.**

También se puede utilizar en una sentencia "if".

Además, es posible comprobar si una cadena determinada no está incluida en un string, para lo que se utiliza la palabra reservada **not in.**

5.2.2 Trocear Strings.

Es posible devolver parte de una cadena utilizando la sintaxis para trocearlo. Para ellos es necesario indicar la posición de inicio y de fin del trozo de cadena que se desea obtener. Si no se indica el primer índice se coge la cadena desde el principio hasta la posición de fin que se especifica. De la misma manera, si no se muestra el último índice se escribe desde el que se pone de inicio hasta el final de la cadena.

Si se utilizan índices negativos quiere decir que se empieza a contar por el final de la cadena.

5.2.3 Modificar Strings.

Python contiene una serie de métodos que es posible utilizar con Strings.

- Para convertir en mayúsculas se utiliza el método upper().
- Para convertir en minúsculas se utiliza el método lower().
- Es posible eliminar los espacios en blanco al inicio y al final de una cadena con el método strip().
- Utilizando el método replace() es posible modificar caracteres dentro de una cadena.
- El método split() devuelve una lista de subcadenas si encuentra el carácter que se pasa como parámetro.
- Para concatenar strings se utiliza el operador +.
- Como ya se ha visto en el apartado de variables en alguno de los ejemplos, no es posible concatenar strings y variables de tipo entero con el operador +, pero sí es posible concatenarlos utilizando el método format().
- El método format() formatea el argumento que se le pasa como parámetro y lo coloca dentro del string en la posición donde se hayan incluido las llaves.

  A este método se le pueden pasar un número ilimitado de parámetros y cada uno de ellos se coloca en el string en su respectivo lugar donde están las llaves {}.

  También se pueden utilizar índices para colocar cada parámetro en la posición correcta del string.

- En los strings también se pueden incluir caracteres no permitidos utilizando el carácter de escape \.
- Además de las comillas dobles, es posible "escapar" los siguientes caracteres:
  - Comilla simple \'
  - Barra de escape \\
  - Nueva línea \n
  - Retorno de carro \r
  - Tabulador \t
  - Retroceso \b
  - Form feed \f (carácter para que la impresora empiece a imprimir en una nueva hoja)
  - Valor en octal \ooo
  - Valor en hexa \xhh

Existen un montón de métodos que se pueden utilizar con strings y todos ellos devuelven un nuevo string con el resultado del método que se ha aplicado, no modifican el string original.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value

rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

5.3 TIPO BOOLEAN

El tipo boolean sólo puede tomar dos valores, “True” o “False”. Este tipo de valores es útil en programación cuando se evalúan determinadas expresiones, por ejemplo, estos valores serán el resultado de realizar comparaciones.

En Python también dan como resultado estos valores las sentencias if.

La función bool() permite evaluar cualquier valor y devolver “True” o “False”. En este caso, la mayor parte de los valores devuelven “True” excepto algunas excepciones:

- Todos los strings son “True” excepto las cadenas vacías.
- Todos los números son “True” excepto el 0.
- Todas las listas, tuplas, conjuntos y diccionarios son “True” excepto si están vacíos.

Por el contrario, se evalúan a “False” lo comentado en los puntos anteriores y el valor “None”.

Las funciones que se definen también pueden devolver un valor de tipo boolean.

En Python también existen métodos ya construidos que devuelven un valor booleano, como por ejemplo el método isinstance() que se utiliza para determinar si un objeto es de un determinado tipo de dato.

5.4 OPERADORES

Python divide los operadores en los siguientes grupos:

- Operadores aritméticos
- Operadores de asignación
- Operadores de comparación
- Operadores lógicos
- Operadores de identidad
- Operadores de afiliación
- Operadores bit a bit

Los operadores aritméticos se utilizan con valores numéricos para realizar operaciones matemáticas.

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Los operadores de asignación se utilizan para asignar valor a las variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Los operadores de comparación se utilizan para comparar dos valores:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Los operadores lógicos se utilizan para combinar sentencias condicionales:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Los operadores de identidad se utilizan para comparar objetos, no con el fin de saber si son iguales, sino para comprobar si son el mismo objeto, es decir, que se trata de la misma posición de memoria.

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Los operadores de afiliación se utilizan para comprobar si una secuencia aparece en un objeto:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Los operadores bit a bit se utilizan para comparar números en binario:

Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y
~	NOT	Inverts all the bits	~x
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

5.4.1 PRECEDENCIA DE LOS OPERADORES

La precedencia de los operadores describe el orden en el que deben evaluarse las operaciones. Los paréntesis son los que tienen la mayor prioridad, lo que implica que las operaciones que están entre paréntesis deben evaluarse primero.

La multiplicación tiene mayor prioridad que la suma, por lo que se evalúa primero.

En general, el orden de precedencia de los operadores se muestra en la siguiente tabla que está ordenada de mayor a menor precedencia.

Operator	Description
`()`	Parentheses
`**`	Exponentiation
`+x  -x  ~x`	Unary plus, unary minus, and bitwise NOT
`*  /  //  %`	Multiplication, division, floor division, and modulus
`+  -`	Addition and subtraction
`<<  >>`	Bitwise left and right shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==  !=  >  >=  <  <=  is  is not  in  not in`	Comparisons, identity, and membership operators
`not`	Logical NOT
`and`	AND
`or`	OR

En el caso de que coincidan dos operadores de igual precedencia la expresión será evaluada de izquierda a derecha.

5.5 ARRAYS EN PYTHON

El lenguaje Python no tiene soporte para arrays como tal, en su lugar se utilizan las listas. Por lo tanto, vamos a ver cómo se utilizan las listas como si fuesen arrays. Para trabajar con arrays en Python se tendría que importar una librería.

Los arrays se utilizan para múltiples valores en una única variable.

Un array es una variable especial que puede albergar más de un valor a la vez.

Si se tiene una lista de elementos, como por ejemplo de marcas de coches, se podría almacenar cada marca en una variable diferente. Sin embargo, esta solución complicaría tareas como recorrer todas las marcas de coche en un bucle, o tener 300 variables diferentes con valores distintos de marcas de coche. Para solucionar estas situaciones se utilizan los arrays.

Un array puede almacenar muchos valores bajo un mismo nombre, y es posible acceder a cada uno de esos valores mediante un índice.

5.5.1 Acceso a los elementos de un array

Se puede hacer referencia a cualquiera de los elementos de un array a través de un índice que comienza en 0 hasta el número de elementos del array menos uno.

5.5.2 Longitud de un array

Para conocer la longitud de un array se utiliza el método len(), es decir, para obtener el número de elementos que componen el array.

Es posible utilizar el bucle for para recorrer cada uno de los elementos del array.

Para añadir nuevos elementos al array se utiliza el método append().

Sin embargo, para eliminar elementos de un array podemos utilizar diferentes métodos. Con el método pop() sin especificar nada más, eliminamos el último elemento del array. Si a este método le añadimos un índice, se elimina el elemento que está en esa posición del array.

También se puede utilizar el método remove() para eliminar un elemento de un array, aunque en este caso se tiene que especificar el valor del elemento que se quiere suprimir. Si hay elementos repetidos en el array, este método sólo borra la primera ocurrencia del elemento.

Python tiene un conjunto de métodos ya construidos que se pueden utilizar con listas/arrays.

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

5.6 LISTAS EN PYTHON

Como se ha comentado antes con los arrays, las listas en Python se utilizan para almacenar distintos elementos en una sola variable. Son uno de los 4 tipos de datos en Python que se utilizan para almacenar colecciones de datos, los otros tres tipos son Tuple, Set y Dictionary y todos tienen diferentes cualidades y usos.

Las listas se crean poniendo sus valores entre corchetes.

Los elementos de una lista se pueden ordenar, intercambiar y es posible que en una lista haya valores repetidos.

Los elementos de la lista están indexados, el primer elemento está en la posición [0], el segundo en la posición [1], etc...

Cuando decimos que los elementos de una lista se pueden ordenar, quiere decir que los elementos tienen un orden definido que no suele cambiar, sin embargo, sí puede modificarse.

Si se añaden nuevos elementos a la lista normalmente se añadirán al final de esta.

Las listas son modificables, es decir, se pueden añadir, borrar y cambiar elementos de la lista después de que esta haya sido creada.

Para conocer el número de elementos que tiene una lista se utiliza el método len().

Las listas pueden ser de cualquier tipo de dato.

También pueden contener diferentes tipos de datos a la vez.

Desde la perspectiva de Python, las listas se definen como objetos de tipo lista. De forma que, también es posible utilizar el constructor de listas list() cuando se crea una nueva lista.

5.6.1 Índices negativos

Si se utilizan índices negativos, significa que se está accediendo desde el final de la lista. El –1 se refiere al último elemento, el –2 al penúltimo elemento, etc...

Es posible especificar un rango de índices indicando donde comienza y dónde termina dicho rango. Cuando se especifica un rango, el resultado es una nueva lista con los elementos indicados en el rango.

Si no se indica el elemento inicial del rango, se entiende que la nueva lista comienza en el primer elemento hasta el marcado como último elemento del rango, pero sin incluir este.

Si no se indica el último elemento del rango, se entiende que la nueva lista comienza en el elemento indicado hasta el último elemento de la lista inicial.

5.6.2 Rango de índices negativos

Si se ponen índices negativos al indicar un rango, significa que se empiezan a seleccionar los elementos por el final de la lista.

Para determinar si un elemento en concreto está incluido en una lista se utiliza la palabra reservada in.

Para modificar el valor de un elemento de la lista, se utiliza el índice de dicho elemento para acceder a él directamente y cambiar ese valor.

En el caso de modificar el valor de varios elementos de una lista en un rango específico, se define una lista con los nuevos valores haciendo referencia a los índices de la lista existente en donde queremos modificar esos nuevos valores.

Milista = [“manzana”, “platano”, “fresa”, “naranja”, “kiwi”, “mango”]

Milista[1 : 3] = [“mandarina”, “limón”]

Al imprimir la lista se puede ver que la mandarina sustituye a la manzana y el limón al plátano en la lista original.

Si se insertan más elementos de los que se reemplazan, los nuevos elementos se insertan en el lugar en el que se especifican y el resto se mueven hacia adelante como corresponde.

Milista = [“manzana”, “plátano”, “fresa”]

Milista [1:2] = [“mandarina”, “limón”]

Si se insertan menos elementos de los que se reemplazan, los nuevos elementos se introducen en donde se indica, y el resto de los que están ya en la lista se mueven, de nuevo, hacia donde corresponde.

Milista = [“manzana”, “plátano”, “fresa”]

Milista[1 : 3] = [“limón”]

La longitud de la lista original cambia, cuando el número de elementos que se insertan no coincide con el número de elementos que se reemplazan.

Para insertar elementos nuevos en la lista, sin reemplazar ninguno de los existentes, se utiliza el método insert().

Con el método insert() se añade un nuevo elemento en el lugar del índice señalado.

Para añadir un nuevo elemento a la lista por el final, se utiliza el método append().

Al añadir elementos a una lista con cualquiera de los dos métodos, el tamaño de la lista cambia.

También es posible añadir los elementos de una lista a otra, los elementos de la que se añaden se incluyen al final de la lista que los recibe.

Por ejemplo, para añadir a una lista de frutas, las frutas de otra lista de frutas tropicales:

Milista = [“manzana”, “plátano”, “fresa”]

Tropical = [“mango”, “papaya”]

Milista.extend(Tropical)

Al igual que es posible añadirlos, es posible eliminar un elemento de una lista especificando de qué elemento se trata. Para ello se utiliza el método remove().

Milista.remove(“plátano”)

También es posible borrar un elemento de la lista indicando el índice de dicho elemento. En este caso se utiliza el método pop().

Milista.pop(1)

Si al método pop() no se le indica ningún índice, éste borra el último elemento de la lista.

Con la palabra reservada del también se puede eliminar un elemento de una lista indicándolo mediante su índice.

del Milista[0]

Con del también es posible borrar la lista completa.

del Milista

El método clear() vacía una lista. No es lo mismo borrar una lista que vaciarla, si se vacía la lista sigue existiendo aunque no tenga elementos, mientras que si se borra la lista dejar de existir.

5.6.3 Bucles con listas

Es posible recorrer los elementos de una lista utilizando un bucle for.

Milista = [“manzana”, “plátano”, “fresa”]

for x in Milista:

print(x)

También es posible recorrer los elementos de una lista a través de sus índices. Para ello se utilizan las funciones range() y len().

for i in range(len(Milista)):

Print(Milista[i]

También se puede recorrer los elementos de una lista utilizando el bucle while. Para ello se utiliza la función len para determinar la longitud de la lista, se comienza desde cero y se utilizan los índices para recorrerla. En este caso, hay que recordar que se debe incrementar en uno la variable que recorre el bucle en cada iteración.

Milista = [“manzana”, “plátano”, “fresa”]

i = 0

while i < len(Milista):

print(Milista[i]

i = i + 1

5.6.4 Ordenar listas

El objeto lista tiene un método sort() que permite ordenar una lista alfanumérica de forma ascendente por defecto.

Milista = [“naranja”, “mango”, “kiwi”, “plátano”]

Milista.sort()

Para ordenar una lista de forma descendente se utiliza el argumento reverse = True

Milista = [“naranja”, “mango”, “kiwi”, “plátano”]

Milista.sort(reverse = True)

Es posible personalizar la función de ordenar utilizando la palabra reservada key = function. La función devolverá un número que se utilizará para ordenar la lista, el número más pequeño primero:

```
def mifuncion(n):

return(n-50)

Milista = [100, 50, 65, 82, 23]

Milista.sort(key = mifuncion)
```

Por defecto el método sort() distingue entre mayúsculas y minúsculas, de forma que las mayúsculas van antes que las minúsculas.

```
Milista = [“plátano”, “Naranja”, “Kiwi”, “fresa”]

Milista.sort()
```

La ordenación que distingue entre mayúsculas y minúsculas, puede devolver en este caso resultados inesperados. Por suerte, es posible que utilicemos funciones ya existentes como funciones llave para realizar ordenaciones de listas, de forma que si queremos una función de ordenación que no sea sensible a mayúsculas ni minúsculas, podemos utilizar la función str.lower como key function.

```
Milista = [“plátano”, “Naranja”, “Kiwi”, “fresa”]

Milista.sort(key = str.lower)
```

¿Qué ocurre si se desea ordenar una lista de forma inversa, independientemente del alfabeto utilizado? El método reverse() invierte el orden actual de los elementos de la lista.

```
Milista = [“plátano”, “Naranja”, “Kiwi”, “fresa”]

Milista.reverse()
```

5.6.5 Copiar listas

No es posible copiar una lista simplemente con la sentencia lista2 = lista1, porque en este caso lista2 será una referencia a lista1 y cualquier cambio realizado sobre lista1 se realizará también de forma automática en lista2. Pero existen formas de hacer copias de listas, una de ellas es utilizar el método copy().

```
Milista = [“manzana”, “plátano”, “fresa”]

Milista2 = Milista.copy()
```

Otra forma de crear una lista es utilizar el método list().

```
Milista = [“manzana”, “plátano”, “fresa”]

Milista2 = list(Milista)
```

Unir listas

Existen varias formas de unir o concatenar dos o más listas en Python. Una de las formas es utilizar el operador +.

Lista1 = [“a”, “b”, “c”]

Lista2 = [1, 2, 3]

Lista3 = Lista1 + Lista2

Otra forma, es añadir los elementos de la lista2 al final de la lista1:

Lista1 = [“a”, “b”, “c”]

Lista2 = [1, 2, 3]

for x in Lista2:

Lista1.append(x)

También se puede utilizar el método extend(), que añade elementos de una lista en la otra:

Lista1 = [“a”, “b”, “c”]

Lista2 = [1, 2, 3]

Lista1.extend(Lista2)

A continuación, se muestran algunos de los métodos sobre listas que se pueden utilizar en Python:

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

6 FUNCIONES EN PYTHON

Una función es un bloque de código que sólo se ejecuta cuando se le llama, a este bloque de código se le pueden pasar datos, que se llaman parámetros, y puede devolver otros datos como resultado.

Para crear una función en Python es necesario utilizar la palabra reservada def.

```
Def mifuncion():
    print("hola desde mi función")
```

Para realizar la llamada a una función se utiliza su nombre más los paréntesis.

```
Def mifuncion():
    print("hola desde mi función")
mifuncion()
```

Como se ha comentado antes, a una función se le puede pasar información mediante argumentos. Los argumentos se pasan después del nombre de la función dentro de los paréntesis. A una función se le pueden pasar tantos argumentos como se desee y deberán ir separados por comas.

En el siguiente ejemplo se le va a pasar a una función un único argumento que será el nombre, y esta devolverá el nombre completo al añadirle el apellido.

```
def mifuncion(nombre):
print(nombre + " López")
mifunción("José")
mifunción("Luis")
mifunción("Ana")
```

El término parámetro o argumento se puede utilizar indistintamente para indicar la información que se le pasa a la función, aunque desde el punto de vista de la función, se puede definir el parámetro como la variable que se incluye dentro de los paréntesis de la función cuando se define, mientras que el argumento es el valor que se envía a la función cuando se invoca.

Por defecto, cuando se llama a una función se le deben de pasar el número correcto de argumentos, es decir, que si la función espera dos argumentos, se debe llamar a la función enviándole los dos argumentos.

```
def mifuncion(nombre, apellido):

print(nombre + " " + apellido)

mifuncion("José", "López")
```

Si se intenta llamar a la función con 1 o 3 argumentos, se obtendrá un error.

Si se desconoce el número de argumentos que se le va a pasar a una función, se debe añadir un * antes del nombre del parámetro en la definición de la función. De esta forma, la función recibirá una tupla de argumentos, y podrá acceder a cada uno de ellos.

```
def mifuncion(*ninhos):

print("El niño más pequeño es " + ninhos[2])

mifuncion("Lucía","Hugo","Brais")
```

Esta forma de llamar al número desconocido de argumentos de una función, también se puede ver como *args en la documentación de Python.

También es posible pasar argumentos a una función con la sintaxis nombre = valor, en este caso, el orden de los argumentos no importa.

```
def mifuncion(ninho3, ninho2, ninho1)

print("El niño más pequeño es ", ninho3)

mifuncion(ninho1 = "Lucía", ninho2 = "Hugo", ninho3 = "Brais")
```

Esta forma de nombrar a los argumentos de una función se suele conocer en la documentación de Python como kwargs.

Si se desconoce cuántos argumentos se le van a pasar a una función con esta sintaxis de nombre = valor, se deben añadir dos asteriscos ** antes del nombre del parámetro en la definición de la función. De esta manera, la función recibirá un diccionario de argumentos, y se puede acceder a cada uno de ellos.

```
def mifuncion(**ninho):

print("Su apellido es " + ninho["apellido"]

mifuncion(nombre = "José", apellido = "López")
```

Esta forma de pasar los argumentos se conoce en la literatura de Python como **kwargs.

También es posible llamar a una función sin pasarle ningún argumento, y que se utilice el parámetro por defecto. El siguiente ejemplo muestra un caso de uso de parámetro por defecto:

```
def mifuncion(pais = “Noruega”):
print(“Soy de “ + pais)
mifuncion(“Suecia”)
mifuncion(“India”)
mifuncion()
mifuncion(“Brasil”)
```

Una función puede recibir argumentos de cualquier tipo (string, número, lista, diccionario, etc...), y dentro de la misma se tratarán como variables de dicho tipo.

```
def mifuncion(comida):
        for x in comida:
print(x)
fruta = [“manzana”, “plátano”, “fresa”]
mifuncion(fruta)
```

Para que una función pueda devolver un valor se debe utilizar la sentencia return.

```
def mifuncion(x):
return 5 * x
print(mifuncion(3))
print(mifuncion(5))
print(mifuncion(9))
```

La definición de una función no puede estar vacía, pero por si alguna razón se tiene que definir una función sin contenido es posible utilizar la sentencia pass para que no devuelva ningún error.

```
def mifuncion()
pass
```

6.1 RECURSIVIDAD

Python también acepta funciones recursivas, es decir, funciones que se pueden llamar a sí mismas dentro de ellas.

La recursividad es un concepto matemático y de programación. Significa que una función puede llamarse a sí misma lo que tiene como ventaja que puede recorrer los datos con un bucle para llegar a un resultado.

Cuando se programa y se utiliza recursividad es necesario tener cuidado ya que es muy fácil caer en situaciones en las que se llame continuamente a la función y nunca se termine, o que por el número de llamadas que se hagan se exceda el consumo de los recursos de la máquina como la memoria o el procesador. Sin embargo, cuando está bien definida, la recursividad puede ser una forma muy eficiente y matemáticamente elegante de dar solución a un problema en programación.

En el siguiente ejemplo, la función tri_recursividad() se define como recursiva. En ella se utiliza la variable k como parámetro que se decrementa en –1 en cada ejecución. La recursividad termina cuando la condición es cero.

```
def tri_recursividad(k):

if (k > 0):

resultado = k + tri_recursividad(k – 1)

print(resultado)

else:

resultado = 0

return resultado

Print(“\n\nResultado del ejemplo de recursividad”)

Tri_recursividad(6)
```

7 SENTENCIA IF...ELSE

Python soporta las condiciones matemáticas lógicas, que habitualmente se van a utilizar en bucles y sentencias “if”.

Estas condiciones son:

Igual: a == b

Distinto: a != b

Menor que: a < b

Menor o igual que: a <= b

Mayor que: a > b

Mayor o igual que: a >= b

La sentencia “if” se escribe utilizando la palabra reservada “if”.

La palabra reservada “elif” es la manera de decir en Python que, si la condición previa que se ha evaluado no es verdadera, entonces que pruebe con la siguiente condición que se añade a continuación.

La palabra reservada “else” captura cualquier opción que no haya encajado en las condiciones evaluadas antes.

También es posible tener una sentencia “else” sin existir una sentencia “elif” previa.

Forma corta de la sentencia if.

Si dentro de la sentencia if se va a ejecutar una única instrucción se puede escribir de una forma corta en una sola línea.

Si hay una sentencia if ... else con una única instrucción en cada una de las partes, se puede escribir de forma abreviada en una sola línea.

También es posible escribir varias sentencias if en una sola línea.

And

El operador lógico and se utiliza para combinar sentencias condicionales.

Or

El operador lógico or también se utiliza para combinar sentencias condicionales.

Not

El operador lógico not se utiliza para invertir el resultado de una sentencia condicional.

If anidados

Es posible tener sentencias if dentro de otras sentencias if, es lo que se llama sentencias if anidadas.

Sentencia pass

Las sentencias if no pueden estar vacías, pero si por algún motivo se da el caso de tener una sentencia if sin contenido, es necesario poner la sentencia pass para evitar que se produzca un error.

8 BUCLES EN PYTHON

Python dispone de dos comandos principales para realizar bucles.

8.1 BUCLE WHILE

Con el bucle while es posible ejecutar una serie de instrucciones mientras se cumple una condición. Es importante que se revise y modifique el estado de la condición para que en algún momento deje de cumplirse, o de lo contrario el bucle while será infinito.

La sentencia break

Con la sentencia break es posible parar el bucle aunque la condición del bucle se cumpla y pudiese continuar.

La sentencia continue

Con la sentencia continue es posible parar la iteración actual del bucle y continuar con la siguiente.

La sentencia else

Con la sentencia else es posible ejecutar un bloque de código una vez más, cuando la condición del bucle ha dejado de cumplirse.

8.2 BUCLE FOR

Un bucle for se utiliza para iterar sobre los elementos de una lista, una tupla, un diccionario, un conjunto o un string.

Con este bucle es posible ejecutar un conjunto de sentencias, una vez por cada uno de los elementos de la lista, tupla, conjunto, etc.

En Python este bucle no necesita una variable de indexación que se establezca de antemano como en otros lenguajes.

Las cadenas (strings) también son objetos iterables, ya que están formados por caracteres.

La sentencia break

Con la sentencia break es posible parar el bucle for antes de que se haya llegado al final, es decir, haya recorrido todos los elementos de la lista, cadena, conjunto….

La sentencia continue

Con esta sentencia es posible parar la iteración actual del bucle y continuar con la siguiente, sin interrumpir el bucle totalmente.

La función range()

Para recorrer un conjunto de sentencias un número concreto de veces, se puede utilizar la función range().

La función range() devuelve una secuencia de números que por defecto empiezan en 0 y se incrementan de 1 en 1 también por defecto, y termina en el número especificado. Es decir, si se pone range(6), esto devuelve una secuencia de números de 0 a 5.

Como se acaba de indicar, la función range() empieza en 0 por defecto, pero puede empezar por cualquier otro número añadiendo un nuevo parámetro, range(2,6), lo que significa que la sencuencia comenzará en el valor 2 y llegará hasta el 5.

También, como se ha indicado, esta función incrementa la secuencia de 1 en 1 por defecto, sin embargo, es posible especificar el valor del incremento añadiendo un tercer parámetro range(2, 30, 3), en este ejemplo, la secuencia empieza en dos, llega a 29 y el incremento va de 3 en 3.

La sentencia else

La palabra reservada else en un bucle for especifica un bloque de código que se ejecutará una vez cuando haya finalizado el bucle.

Si el bucle for se para por una sentencia BREAK la sentencia else NO SE EJECUTA.

Bucles anidados

Un bucle anidado se define como un bucle dentro de otro. En estos casos, el bucle interior se ejecuta por cada iteración del bucle exterior.

Sentencia pass

Los bucles for no pueden estar vacíos, pero si por alguna razón se tiene un bucle for sin contenido, se deberá poner la sentencia pass para evitar errores.