2.2.- Contenido 2: Reconocer los tipos de datos y sentencias básicas del lenguaje para la construcción de programas

Objetivo de la jornada

1. Reconoce los tipos de datos usados en el entorno Python y su uso para la construcción de un programa.
2. Reconoce los operadores matemáticos, lógicos y de comparación para la construcción de expresiones.
3. Reconoce las sentencias básicas del lenguaje como condicionales y bucles para la construcción de programas.

2.2.1.- Variables, Tipos de Datos y Operadores

2.2.1.1. Variables

Una variable de Python es una ubicación de memoria reservada para almacenar valores. En otras palabras, una variable en un programa de Python proporciona datos al computador para su procesamiento.
Cada valor de una variable en Python tiene un tipo de datos. Los diferentes tipos de datos en Python son Números, Lista, Tupla, Cadenas de Caracteres (Strings), Diccionario, etc.

NOMBRAR Y USAR VARIABLES

Cuando se usan variables en Python, se debe cumplir con algunas reglas y pautas. Romper algunas de estas reglas provocará errores; otras reglas solo ayudan a escribir código que sea más fácil de leer y comprender. Es útil tener presente las siguientes reglas para variables en Python:

- Los nombres de las variables solo pueden contener letras, números y guiones bajos. Pueden comenzar con una letra o un guión bajo, pero no con un número. Por ejemplo, podemos llamar a una variable **`mensaje_1`** pero no **`1_mensaje`**.
- No se permiten espacios en los nombres de variables, pero se pueden usar guiones bajos para separar palabras en nombres de variables. Por ejemplo, **`mesaje_de_saludo`** funciona, pero **`mensaje de saludo`** provocará errores.
- Se vita el uso de palabras clave de Python y nombres de funciones como nombres de variables; es decir, no se deben utilizar palabras que Python haya reservado para un propósito particular en un programa, como por ejemplo la palabra **`print`**.
- Los nombres de las variables deben ser breves pero descriptivos. Por ejemplo, el nombre es mejor que n, student_name es mejor que s_n y name_length es mejor que length_of_persons_name.

- Tenga cuidado al usar la letra l minúscula y la letra O mayúscula porque podrían confundirse con los números 1 y 0.

Puede llevar algo de práctica aprender a crear buenos nombres de variables, especialmente a medida que sus programas se vuelven más interesantes y complicados. Como escribes más programas y comienzas a leer el código de otras personas, mejorará en la creación de nombres significativos.

Las variables de Python que está utilizando en este momento deben estar en minúsculas. No obtendrás errores si usa letras mayúsculas, pero es una buena idea evitar usarlas por ahora.

ALCANCE DE VARIABLES

En primer lugar, podemos decir que las variables declaradas fuera de una función tienen alcance global. Por otro lado, aquellas definidas dentro de una función tienen alcance local a la misma. En general se dice que las primeras son variables globales y las segundas variables locales. Las variables globales son válidas durante la ejecución del programa, mientras que las locales pueden usarse únicamente dentro de la función en la que son declaradas.
https://blog.carreralinux.com.ar/2017/06/alcance-de-las-variables-en-python/

Las variables locales no pueden accederse fuera de la función. Sin embargo, dentro de una función sí se puede acceder a variables declaradas en el alcance (o scope) global. El ejemplo que compartimos a continuación y que es ilustrado en la Fig. 1 servirá para aclarar estos conceptos.

```
>>> persona = "Alberto"
>>> lugar = "Alemania"
>>> def saludo(nombre):
...     comienzo_saludo = "Hola"
...     print(comienzo_saludo, nombre, "de", lugar )
...
>>> # Usamos la variable persona al llamar la función:
>>> saludo(persona)
Hola Alberto
>>> # No podemos utilizar la variable comienzo_saludo fuera de la
función
>>> print(comienzo_saludo, persona)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'comienzo_saludo' is not defined
```

En este ejemplo, la variable **persona** está en el scope global del programa. Por otro lado, **comienzo_saludo** es local dentro de la función **saludo**. Por ese motivo, podemos utilizar **persona** cuando llamamos a la función. Como podemos observar, la función **saludo** se ejecutó correctamente. Sin embargo, no podemos utilizar **comienzo_saludo** fuera de la función. Esa es la razón por la que el último print arroja un error. Aunque no es una buena práctica, es posible definir una variable global

dentro de una función de tal forma que pueda ser accedida desde cualquier lugar de nuestro programa. Para esto, en el último ejemplo deberíamos definir:

```
>>> def saludo(nombre):
...     global comienzo_saludo
...     comienzo_saludo = "Hola"
...     print(comienzo_saludo, nombre, "de", lugar )
...
```

2.2.1.2. TIPOS DE DATOS

Esta sección cubre números, cadenas, listas, diccionarios, tuplas, archivos, conjuntos y otros tipos integrados básicos. En general, todos los tipos de datos compuestos cubiertos aquí (por ejemplo, listas, diccionarios y tuplas) pueden anidarse unas dentro de otras arbitrariamente y tan profundamente como sea necesario. Los conjuntos también pueden participar en el anidamiento, pero pueden contener solo objetos inmutables.

NÚMEROS

Los números son valores inmutables (inmutables), que soportan operaciones numéricas. Esta sección cubre los tipos de números básicos (Enteros, punto flotante), así como tipos más avanzados (complejos, decimales y fracciones).

Los números se escriben en una variedad de formas literales numéricas. Veremos aquí cómo asignaríamos estos números a una variable en Python en algunos casos particulares:

Enteros

```
>>> numero_entero_1 = 1234
>>> numero_entero_2 = -24
>>> numero_entero_3 = +42
>>> numero_entero_4 = 0
```

Enteros creados desde otros objetos o desde cadenas de caracteres (Strings) con posible conversión de base:

```
>>> numero_entero_5 = int(9.1)
>>> numero_entero_6 = int('-9')
>>> numero_entero_7 = int('1111', 2)
>>> numero_entero_8 = int('0b1111', 0)
```

Nota sobre números booleanos: Los booleanos no son propiamente números; sin embargo, estos solo pueden tomar dos valores diferentes: True (verdadero) o False (falso). Dada esta circunstancia, parece lógico utilizar un tipo entero que necesite menos espacio en memoria que el original, ya que, solo necesitamos dos números: 0 y 1. En realidad, aunque Python cuenta con el tipo integrado bool, este no es más que una versión personalizada del tipo int.

Punto Flotante (En CPython equivalen a Dobles de C)

```
>>> numero_flotante_1 = 1.23
```

```
>>> numero_flotante_2 = 1.
>>> numero_flotante_3 = .1
>>> numero_flotante_4 = 3.14e-10
>>> numero_flotante_5 = 4E210
>>> numero_flotante_6 = 4.0e+210
```

Flotantes creados desde otros objetos:

```
>>> numero_flotante_7 = float(9)
>>> numero_flotante_8 = float('1e2')
>>> numero_flotante_9 = float('-.1')
```

Los valores para números reales que podemos utilizar en Python tienen un amplio rango, gracias a que el lenguaje emplea para su representación un bit para el signo (positivo o negativo), 11 para el exponente y 52 para la mantisa. Esto también implica que se utiliza la precisión doble. Recordemos que en algunos lenguajes de programación se emplean dos tipos de datos para los reales, que varían en función de la representación de su precisión. Es el caso de C, que cuenta con el tipo float y double. En Python no existe esta distinción y podemos considerar que los números reales representados equivalen al tipo double de C.

Octal, Hexadecimal y Binario para enteros

```
>>> numero_octal = 0o177
>>> numero_hexadecimal = 0x9ff
>>> numero_binario = 0b1111
```

Nota: hex(N), oct(N) y bin(N) crean cadenas de caracteres desde enteros.

Complejos

```
>>> numero_complejo_1 = 4+4j
>>> numero_complejo_2 = 3.0 + 4.0j
>>> numero_complejo_3 = 3J
```

Complejos creados desde otros objetos:

```
>>> numero_flotante_10 = complex(3, 4.0)
```

Tipos basados en módulos: Decimal y fracción

```
>>> import decimal
>>> numero_decimal_4 = decimal.Decimal('1.33')
>>> import fractions
>>> numero_fraccion = fractions.Fraction(4, 3)
```

Otros tipos numéricos

Python también incluye un tipo numérico llamado **set** y otros tipos numéricos tales como vectores optimizados y matrices que están disponibles como extensiones de código abierto de terceros (por ejemplo, el paquete **NumPy** que posee una gran cantidad de herramientas numéricas http://www.numpy.org). El mundo opensource también provee otras capacidades numéricas a Python, tales como herramientas de visualización, estadísticas, matemática de punto flotante de precisión extendida, y más.

Operadores Numéricos

Para trabajar con números, no solo necesitamos representarlos a través de diferentes tipos, sino también es importante realizar operaciones con ellos. Python cuenta con diversos operadores para aplicar diferentes operaciones numéricas. Dentro del grupo de las aritméticas, contamos con las básicas suma, división entera y real, multiplicación y resta. En cuanto a las operaciones de bajo nivel y entre bits, existen tanto las operaciones NOT y NOR, como XOR y AND. También contamos con operadores para comprobar la igualdad y desigualdad y para realizar operaciones lógicas como AND y OR. Como en otros lenguajes de programación, en Python también existe la precedencia de operadores, lo que deberemos tener en cuenta a la hora de escribir expresiones que utilicen varios de ellos. Sin olvidar que los paréntesis pueden ser usados para marcar la preferencia entre unas operaciones y otras dentro de la misma expresión.

Operación	Descripción
a + b	Suma
a - b	Resta
a * b	Multiplicación
a % b	Resto
a / b	División real
a // b	División entera
a ** b	Potencia
a | b	OR (bit)
a ^ b	XOR (bit)
a & b	AND (bit)
a == b	Igualdad
a != b	Desigualdad
a or b	OR (lógica)
a and b	AND (lógica)
not a	Negación (lógica)

Funciones matemáticas sobre Números

A parte de las operaciones numéricas básicas, anteriormente mencionadas, Python nos permite aplicar otras muchas funciones matemáticas. Entre ellas, tenemos algunas como el valor absoluto, la raíz cuadrada, el cálculo del valor máximo y mínimo de una lista o el redondo para números reales. Incluso es posible trabajar con operaciones trigonométricas como el seno, coseno y tangente. La mayoría de estas operaciones se encuentran disponibles a través de un módulo que es parte de la librería básica llamado **math**. Por ejemplo, el valor absoluto del número -32.67 puede ser calculado de la siguiente forma:

```
>>> abs(-32.34)
32.34
```

Para algunas operaciones necesitaremos importar el mencionado módulo math. Por ejemplo el siguiente código calcula la raíz cuadrada del número 25:

```
>>> import math
>>> math.sqrt(25)
5.0
```

Otras operaciones que podemos realizar con números es cambio de base. Por ejemplo, para pasar de decimal a binario o de octal a hexadecimal. Para ello, Python cuenta con las funciones **int(), hex()**, **oct()** y **bin()**. La siguiente sentencia muestra cómo obtener en hexadecimal el valor del entero 245:

```
>>> hex(245)
'0xf5'
```

Si lo que necesitamos es el valor octal, bastará con ejecutar la siguiente sentencia:

```
>>> oct(245)
'0o365'
```

Las funciones de cambio de base admiten como argumentos cualquier representación numérica admitida por Python. Esto quiere decir, que la siguiente expresión también sería válida:

```
>>> bin(0xf5)
'0b11110101'
```

CADENAS DE CARACTERES (STRINGS)

Las cadenas de caracteres (Strings) son otro de los tipos de datos más utilizados en programación. El intérprete de Python integra este tipo de datos, además de una extensa serie de funciones para interactuar con diferentes cadenas de texto.

En algunos lenguajes de programación, como en C, las cadenas de texto no son un tipo integrado como tal en el lenguaje. Esto implica un poco de trabajo extra a la hora de realizar operaciones como la concatenación. Sin embargo, esto no ocurre en Python, lo que hace mucho más sencillo definir y operar con este tipo de dato.

Básicamente, una cadena de caracteres o string es un conjunto inmutable y ordenado de caracteres. Para su representación y definición se pueden utilizar tanto comillas dobles ("), como simples (‘). Por ejemplo, en Python, la siguiente sentencia crearía una nueva variable de tipo string:

```
>>> string = “string en python”
```

Si necesitamos declarar un string que contenga más de una línea, podemos hacerlo utilizando comillas triples en lugar de dobles o simples:

```
>>> string_multilinea = “””string multilinea en python. De esta
... forma se pueden definir textos con varias
... lineas en un mismo string”””
```

Python tiene tres tipos de strings con interfaces similares:

Str (Unicode)
Una secuencia inmutable de caracteres, utilizada para todo texto, tanto ASCII como Unicode. Por defecto las cadenas de caracteres en Python corresponden a Unicode. Para las cadenas de texto declaradas por defecto, internamente, Python emplea el tipo denominado str. Podemos comprobarlo sencillamente declarando una cadena de texto y preguntando a la función type():

```
>>> string_str = "Ñuñoa de Chile"
>>> type(string_str)
<class 'str'>
>>> print(string_str)
Ñuñoa de Chile
```

byte y bytearray
El tipo **byte** solo admite caracteres en codificación ASCII y, al igual que los de tipo Unicode, son inmutables. Por otro lado, el tipo **bytearray** es una versión mutable del tipo **byte**.

Para declarar un **string** de tipo **byte**, basta con anteponer la letra b antes de las comillas:

```
>>> string_tipo_byte = b”string de tipo byte en python”
>>> type(string_tipo_byte)
<class ‘bytes’>
```

Un **bytearray** se declara utilizando la función integrada del intérprete. Debe indicarse el tipo de codificación que deseamos emplear. Por ejemplo para crear un string de este tipo con codificación de caracteres **latin1** debemos hacer:

```
>>> string_bytearray=bytearray("Ñuñoa de Chile", "latin1")
>>> print(string_bytearray)
bytearray(b'\xd1u\xf1oa de Chile')

>>> string_bytearray_2=bytearray("Ñuñoa de Chile", "utf16")
>>> print(string_bytearray_2)
bytearray(b'\xff\xfe\xd1\x00u\x00\xf1\x00o\x00a\x00 \x00d\x00e\x00
\x00C\x00h\x00i\x00l\x00e\x00')
```

Realizar conversión entre los distintos tipos de strings es posible gracias a dos tipos de funciones llamadas **encode()** y **decode()**. La primera de ellas se utiliza para transformar un tipo **str** en un tipo **byte**. La función **decode()** realiza el paso inverso, es decir, convierte un string **byte** a otro de tipo **str**.

```
>>> string_str = "string de tipo str"
>>> string.str.encode()
b'string de tipo str'
```

```
>>> string_byte = b"string de tipo byte"
>>> string_byte.decode()
'string de tipo byte'
```

La función **encode()** admite como parámetro un tipo de codificación específico. Si este tipo es indicado, el intérprete utilizará el número de bytes necesarios para su representación en memoria, en función de cada codificación. Esto pues para su representación interna, cada tipo de codificación de caracteres requiere de un determinado número de bytes.

Funciones y Métodos sobre Strings

Python posee una variada gama de funciones y métodos para trabajar con strings. Las primeras pueden ser invocadas directamente y reciben como argumento una string. Por otro lado, pueden invocarse diferentes métodos con los que cuenta este tipo de dato.

Una de las funciones más comunes que podemos utilizar sobre strings es el cálculo del número de caracteres que contiene. El siguiente ejemplo nos muestra cómo hacerlo:

```
>>> string_ejemplo = "Ejemplo de cálculo de longitud de string"
>>> len(string_ejemplo)
40
```

Otro ejemplo de función que puede ser invocada es **print()**, que imprime en pantalla el argumento que le entreguemos.

```
>>> nombre = "Alan Madison Turing"
>>> print(nombre)
Alan Madison Turing
```

Respecto a los métodos con los que cuentan los objetos de tipo string, Python incorpora varios de ellos para poder llevar a cabo funcionalidades básicas relacionadas con cadenas de caracteres. Entre ellas, contamos con métodos para buscar un substring dentro de otro, para reemplazar substrings, para borrar espacios en blanco, para pasar de mayúsculas a minúsculas, y viceversa.

La función **find()** devuelve el índice correspondiente al primer carácter de la cadena original que coincide con el buscado:

```
>>> nombre = "Alan Madison Turing"
>>> nombre.find("M")
5
```

Si el carácter buscado no existe en la cadena, find() devolverá -1. Para reemplazar una serie de caracteres por otros, contamos con el método **replace()**. En el siguiente ejemplo, sustituiremos la subcadena "América" por "Marte":

```
>>> texto_base = "Los americanos vivimos en América"
>>> texto_base.replace("América", "Marte")
'Los americanos vivimos en Marte'
```

Obsérvese que **replace()** no altera el valor de la variable sobre el que se ejecuta. Así pues, en nuestro ejemplo, el valor de la variable texto_base seguirá siendo "Los americanos vivimos en América".

Los métodos **strip()**, **lstrip()** y r**strip()** ayudan a eliminar espacios en blanco iniciales y finales, solo los que aparecen al inicio y solo los que se encuentran al final, respectivamente:

```
>>> texto_base = "      Los     americanos vivimos en América    "
>>> texto_base.strip()
'Los     americanos vivimos en América'
>>> texto_base.lstrip()
'Los     americanos vivimos en América    '
>>> texto_base.rstrip()
'      Los     americanos vivimos en América'
```

El método upper() convierte todos los caracteres de una cadena de texto a mayúsculas, mientras que lower() lo hace a minúsculas. Por ejemplo:

```
>>> texto_base = "Somos Americanos"
>>> texto_base.upper()
'SOMOS AMERICANOS'
>>> texto_base.lower()
'somos americanos'
```

Relacionados con **upper()** y **lower()** encontramos otro método llamado **capitalize()**, el cual solo convierte el primer carácter de un string a mayúsculas:

```
>>> texto_base="nombre"
>>> texto_base.capitalize()
'Nombre'
```

En ocasiones puede ser muy útil dividir un string basándonos en un caracter que aparece repetidamente en ella. Esta funcionalidad es la que nos ofrece **split().** Supongamos que tenemos un cadena con varios valores separadas por **","** y que necesitamos una lista donde cada valor se corresponda con los que aparecen delimitados por el mencionado carácter. El siguiente ejemplo nos muestra cómo hacerlo:

```
>>> nombres = "Carlos,Julio,Roberto,Patricio"
>>> nombres.split(",")
['Carlos', 'Julio', 'Roberto', 'Patricio']
```

join() devuelve una cadena de caracteres donde los caracteres del string original aparecen separados por el caracter que llama al método:

```
>>> texto_base="texto_de_prueba"
```

```
>>> ",".join(texto_base)
't,e,x,t,o,_,d,e,_,p,r,u,e,b,a'
```

Operadores sobre Strings

El operador **+** nos permite concatenar dos strings, el resultado puede ser almacenado en una nueva variable:

```
>>> texto_concatenado = "Hola" + " Mundo!"
>>> print(texto_concatenado)
Hola Mundo!
```

También es posible concatenar una variable de tipo string con otro string o concatenar directamente dos variables. Sin embargo, también podemos prescindir del operador + para concatenar strings. A veces, la expresión es más fácil de leer si empleamos el método format(). Este método admite emplear, dentro del string, los caracteres {}, entre los que irá, número o el nombre de una variable. Como argumentos del método pueden pasarse variables que serán sustituidas, en tiempo de ejecución, por los marcadores {}, indicados en la cadena de texto. Por ejemplo, veamos cómo las siguientes expresiones son equivalentes y devuelven el mismo resultado:

```
>>> txt1 = "Albert"
>>> txt2 = "Einstein"
>>> "Nombre: " + txt1 + ", Apellido: " + txt2
'Nombre: Albert, Apellido: Einstein'
>>> "Nombre: {0}, Apellido: {1}".format(txt1, txt2)
'Nombre: Albert, Apellido: Einstein'
>>> "Nombre: {txt1}, Apellido: {txt2}".format(txt1=txt1,txt2=txt2)
'Nombre: Albert, Apellido: Einstein'
```

La concatenación entre strings y números también es posible, siendo para ello necesario el uso de funciones como **int()** y **str()**. El siguiente ejemplo es un caso sencillo de cómo utilizar la función **str()**:

```
>>> num = 3
>>> "Número: " + str(num)
```

Interesante resulta el uso del operador * aplicado a strings, ya que nos permite repetir un string **n** veces. Supongamos que deseamos repetir la cadena "Hola Mundo" cuatro veces. Para ello, bastará con ejecutar la siguiente sentencia:

```
>>> print("Hola Mundo "*4)
Hola Mundo Hola Mundo Hola Mundo Hola Mundo
```

Gracias al operador **in** podemos averiguar si un determinado caracter se encuentra o no en un string. Al aplicar el operador, como resultado, obtendremos **True** o **False**, en función de si el valor se encuentra o no en la cadena. Comprobémoslo en el siguiente ejemplo:

```
>>> texto_de_prueba = "Esto es un texto para probar"
```

```
>>> "para" in cad
True
```

Un string es inmutable en Python, pero podemos acceder, a través de índices, a cada carácter que forma parte de la cadena:

```
>>> texto_de_prueba = "textos"
>>> print(texto_de_prueba[2])
x
```

Los comentados índices también nos pueden ayudar a obtener subcadenas de texto basadas en la original. Utilizando la variable **texto_de_prueba** del último ejemplo podemos imprimir solo los tres primeros caracteres:

```
>>> print(texto_de_prueba[:3])
tex
```

En el ejemplo anterior el operador **:** nos ha ayudado a nuestro propósito. Dado que delante del operador no hemos puesto ningún número, estamos indicando que vamos a utilizar el primer carácter del string. Detrás del operador añadimos el número del índice de la cadena de texto que será utilizado como último valor. Así pues, obtendremos los tres primeros caracteres. Los índices negativos también funcionan, simplemente indican que se empieza contar desde el último carácter. La siguiente sentencia devolverá el valor **o**:

```
>>> texto_de_prueba[-2]
o
```

A continuación, veamos un ejemplo donde utilizamos un número junto con mencionado operador:

```
>>> texto_de_prueba[3:]
'tos'
```

TUPLAS

En Python una tupla es una estructura de datos que representa una colección de objetos, pudiendo estos ser de distintos tipos. Internamente, para representar una tupla, Python utiliza un array de objetos que almacena referencias hacia otros objetos.

Para declarar una tupla se utilizan paréntesis, entre los cuales deben separarse por comas los elementos que van a formar parte de ella. En el siguiente ejemplo, crearemos una tupla con tres valores, cada uno de un tipo diferente:

```
>>> t = (1, 'a', 3.5)
```

Los elementos de una tupla son accesibles a través del índice que ocupan en la misma, exactamente igual que en un array:

```
>>> t[1]
'a'
```

Debemos tener en cuenta que las tuplas son un tipo de dato inmutable, esto significa que no es posible asignar directamente un valor a través del índice.
A diferencia de otros lenguajes de programación, en Python es posible declarar una tupla añadiendo una coma al final del último elemento:

```
>>> t = (1, 3, 'c', )
```

Dado que una tupla puede almacenar distintos tipos de objetos, es posible anidar diferentes tuplas; veamos un sencillo ejemplo de ello:

```
>>> t = (1, ('a', 3), 5.6)
```

Una de las peculiaridades de las tuplas es que es un objeto iterable; es decir, con un sencillo bucle for (Que estudiaremos más adelante) podemos recorrer fácilmente todos sus elementos:

```
>>> for ele in t:
...     print(ele)
...
1
('a', 3)
5.6
```

Concatenar dos tuplas es sencillo, se puede hacer directamente a través del operador **+**. Otros de los operadores que se pueden utilizar es * , que sirve para crear una nueva tupla donde los elementos de la original se repiten **n** veces.
Observemos el siguiente ejemplo y el resultado obtenido:

```
>>> (’r', 2) * 3
('r', 2, 'r', 2, 'r', 2)
```

Los principales métodos que incluyen las tupas son **index()** y **count()**. El primero de ellos recibe como parámetro un valor y devuelve el índice de la posición que ocupa en la tupla. Veamos el siguiente ejemplo:

```
>>> t = (1, 3, 7)
>>> t.index(3)
1
```

El método **count()** sirve para obtener el número de ocurrencias de un elemento en una tupla:

```
>>> t = (1, 3, 1, 5, 1,)
>>> t.count(1)
3
```

Sobre las tuplas también podemos usar la función integrada **len()**, que nos devolverá el número de elementos de la misma. Obviamente, deberemos pasar la variable tupla como argumento de la mencionada función.

LISTAS

Básicamente, una lista es una colección ordenada de objetos, similar al array dinámico empleado en otros lenguajes de programación. Puede contener distintos tipos de objetos, es mutable y Python nos ofrece una serie de funciones y métodos integrados para realizar diferentes tipos de operaciones.

Para definir una lista se utilizan corchetes **[ ]** entre los cuales pueden aparecer diferentes valores separados por comas. Esto significa que ambas declaraciones son válidas:

```
>>> lista = []
>>> li = [2, 'a' , 4]
```

Al igual que las tuplas, las listas son también iterables, así pues, podemos recorrer sus elementos empleando un bucle:

```
>>> for ele in li:
...     print(ele)
...
2
'a'
4
```

A diferencia de las tuplas, los elementos de las listas pueden ser reemplazados accediendo directamente a través del índice que ocupan en la lista. De este modo, para cambiar el segundo elemento de nuestra lista **li**, bastaría como ejecutar la siguiente sentencia:

```
>>> li[1] = 'b'
```

Obviamente, los valores de las listas pueden ser accedidos utilizando el valor del índice que ocupan en la misma:

```
>>> li[2]
4
```

Podemos comprobar si un determinado valor existe en una lista a través del operado in, que devuelve **True** en caso afirmativo y **False** en caso contrario:

```
>>> 'a' in li
True
```

Existen dos funciones integradas que relacionan las listas con las tuplas: **list()** y **tuple()**. La primera toma como argumento una tupla y devuelve una lista. En cambio, **tuple()** devuelve una tupla al recibir como argumento una lista. Por ejemplo, la siguiente sentencia nos devolverá una tupla:

```
>>> tuple(li)
(2, 'a', 4)
```

Operaciones como la suma (+) y la multiplicación (*) también pueden ser aplicadas sobre listas. Su funcionamiento es exactamente igual que en las tuplas.

Insertar y borrar en listas

Para añadir un nuevo elemento a una lista contamos con el método **append()**. Como parámetro hemos de pasar el valor que deseamos añadir y este será insertado automáticamente al final de la lista. Volviendo a nuestra lista ejemplo de tres elementos, uno nuevo quedaría insertado a través de la siguiente sentencia:

```
>>> li.append('nuevo')
```

Nótese que, para añadir un nuevo elemento, no es posible utilizar un índice superior al número de elementos que contenga la lista. La siguiente sentencia lanza un error:

```
>>> li[4] = 23
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Sin embargo, el método **insert()** sirve para añadir un nuevo elemento especificando el índice. Si pasamos como índice un valor superior al número de elementos de la lista, el valor en cuestión será insertado al final de la misma, sin tener en cuenta el índice pasado como argumento. De este modo, las siguientes sentencias producirán el mismo resultado, siendo **'c'** el nuevo elemento que será insertado en la lista **li**:

```
>>> li.insert(3,'c')
>>> li.insert(12,'c')
```

Por el contrario, podemos insertar un elemento en una posición determinada cuyo índice sea menor al número de valores de la lista. Por ejemplo, para insertar un nuevo elemento en la primera posición de nuestra lista li, bastaría con ejecutar la siguiente sentencia:

```
>>> li.insert(0,'d')
>>> li
>>> ['d',2,'a',4]
```

Si lo que necesitamos es borrar un elemento de una lista, podemos hacerlo gracias a la función **del()**, que recibe como argumento la lista junto al índice que referencia al

elemento que deseamos eliminar. La siguiente sentencia ejemplo borra el valor 2 de nuestra lista li:

```
>>> del(li[1])
```

Como consecuencia de la sentencia anterior, la lista queda reducida en un elemento. Para comprobarlo contamos con la función len(), que nos devuelve el número de elementos de la lista:

```
>>> len(li)
2
```

Obsérvese que la anterior función también puede recibir como argumento una tupla o un string. En general, **len()** funciona sobre tipos de objetos iterables. También es posible borrar un elemento de una lista a través de su valor. Para ello contamos con el método **remove()**:

```
>>> li.remove('d')
```

Si un elemento aparece repetido en la lista, el método **remove()** solo borrará la primera ocurrencia que encuentre en la misma. Otro método para eliminar elementos es **pop()**. A diferencia de **remove()**, **op()** devuelve el elemento borrado y recibe como argumento el índice del elemento que será eliminado. Si no se pasa ningún valor como índice, será el último elemento de la lista el eliminado. Este método puede ser útil cuando necesitamos ambas operaciones (borrar y obtener el valor) en una única Sentencia.

Orden de listas

Los elementos de una lista pueden ser ordenados a través del método **sort()** o utilizando la función **sorted()**. Como argumento se puede utilizar **reverse** con el valor **True** o **False**. Por defecto, se utiliza el segundo valor, el cual indica que la lista será ordenada de mayor a menor. Si por el contrario el valor es **True**, la lista será ordenada inversamente. Veamos un ejemplo para ordenar una lista de enteros:

```
>>> lista = [3, 1, 9, 8, 7]
>>> sorted(lista)
[1, 3, 7, 8, 9]
>>> sorted(lista, reverse=True)
[9, 8, 7, 3, 1]
>>> lista
[3, 1, 9, 8, 7]
```

Como el lector habrá podido observar, la lista original ha quedado inalterada. Sin embargo, si en lugar de utilizar la función **sorted()**, empleamos el método **sort()**, la lista quedará automáticamente modificada. Ejecutemos las siguientes setencias para comprobarlo:

```
>>> lista.sort()
>>> lista
[1, 3, 7, 8, 9]
```

Tanto para aplicar sort() como sorted() debemos tener en cuenta que la lista que va a ser ordenada contiene elementos que son del mismo tipo. En caso contrario, el intérprete de Python lanzará un error. No obstante, es posible realizar ordenaciones de listas con elementos de distinto tipo si es el programador el encargado de establecer el criterio de ordenación. Para ello, contamos con el parámetro key que puede ser pasado como argumento. El valor del mismo puede ser una función que fijará cómo ordenar los elementos.

Además, el mencionado parámetro también puede ser utilizado para cambiar la forma de ordenar que emplee el intérprete por defecto, aunque los elementos sean del mismo tipo. Supongamos que definimos la siguiente lista:

```
>>> lis = ['aA', 'Ab', 'Cc', 'ca']
```

Ahora ordenaremos con la función **sorted()** sin ningún parámetro adicional y observaremos que el criterio de ordenación que utiliza el intérprete, por defecto, es ordenar primero las letras mayúsculas:

```
>>> sorted(lis)
['Ab', 'Cc' , 'aA', 'ca']
```

Sin embargo, al pasar como argumento un determinado criterio de ordenación, el resultado varía:

```
>>> sorted(lis, key=str.lower)
['aA', 'Ab', 'ca', 'Cc']
```

Otro método que contienen las listas relacionado con la ordenación de valores es **reverse()**, que automáticamente ordena una lista en orden inverso al que se encuentran sus elementos originales. Tomando el valor de la última lista de nuestro ejemplo, llamaremos al método para ver qué ocurre:

```
>>> lista.reverse()
>>> lista
[9, 8, 7, 3, 1]
```

Los métodos y funciones de ordenación no solo funcionan con números, sino también con caracteres y con strings:

```
>>> lis = ['be', 'ab', 'cc', 'aa', 'cb']
>>> lis.sort()
>>> lis
['aa', 'ab','be', 'cb', 'cc']
```

Comprensión de Listas

La comprensión de listas es una construcción sintáctica de Python que nos permite declarar una lista a través de la creación de otra. Esta construcción está basada en el principio matemático de la teoría de comprensión de conjuntos. Básicamente, esta teoría afirma que un conjunto se define por comprensión cuando sus elementos son nombrados a través de sus características. Por ejemplo, definimos el conjunto **S** como aquel que está formado por todos los meses del año: **S = {meses del año}** Veamos un ejemplo práctico para utilizar la mencionada construcción sintáctica en Python:

```
>>> lista = [ele for ele in (1, 2, 3)]
```

Como resultado de la anterior sentencia, obtendremos una lista con tres elementos diferentes:

```
>>> print(lista)
[1, 2, 3]
```

Gracias a la comprensión de listas podemos definir y crear listas ahorrando líneas de código y escribiendo el mismo de forma más elegante. Sin la comprensión de listas, deberíamos ejecutar las siguientes sentencias para lograr el mismo resultado:

```
>>> lista = []
>>> for ele in (1, 2, 3):
...     lista.append(ele)
...
```

Matrices

Anidando listas podemos construir matrices de elementos. Estas estructuras de datos son muy útiles para operaciones matemáticas. Debemos tener en cuenta que complejos problemas matemáticos son resueltos empleando matrices.

Además, también son prácticas para almacenar ciertos datos, aunque no se traten estrictamente de representar matrices en el sentido matemático.

Por ejemplo, una matriz matemática de dos dimensiones puede definirse de la siguiente forma:

```
>>> matriz = [[1, 2, 3],[4, 5, 6]]
```

Para acceder al segundo elemento de la primera matriz, bastaría con ejecutar la siguiente sentencia:

```
>>> matriz = [0][1]
```

Asimismo, podemos cambiar un elemento directamente:

```
>>> matriz[0][1] = 33
>>> m
```

```
[[1, 33, 3],[4, 5, 6]]
```

DICCIONARIOS

Un diccionario es una estructura de datos que almacena una serie de valores utilizando otros como referencia para su acceso y almacenamiento. Cada elemento de un diccionario es un par clave-valor donde el primero debe ser único y será usado para acceder al valor que contiene. A diferencia de las tuplas y las listas, los diccionarios no cuentan con un orden específico, siendo el intérprete de Python el encargado de decidir el orden de almacenamiento. Sin embargo, un diccionario es iterable, mutable y representa una colección de objetos que pueden ser de diferentes tipos.

Gracias a su flexibilidad y rapidez de acceso, los diccionarios son una de las estructuras de datos más utilizadas en Python. Internamente son representadas como una tabla hash, lo que garantiza la rapidez de acceso a cada elemento, además de permitir aumentar dinámicamente el número de ellos. Otros muchos lenguajes de programación hacen uso de esta estructura de datos, con la diferencia de que es necesario implementar la misma, así como las operaciones de acceso, modificación, borrado y manejo de memoria. Python ofrece la gran ventaja de incluir los diccionarios como estructuras de datos integradas, lo que facilita en gran medida su utilización.

Para declarar un diccionario en Python se utilizan las llaves **{}** entre las que se encuentran los pares clave-valor separados por comas. La clave de cada elemento aparece separada del correspondiente valor por el carácter **:** . El siguiente ejemplo muestra la declaración de un diccionario con tres valores:

```
>>> diccionario = {'a': 1, 'b': 2, 'c': 3}
```

Alternativamente, podemos hacer uso de la función **dict()** que también nos permite crear un diccionario. De esta forma, la siguiente sentencia es equivalente a la anterior:

```
>>> diccionario = dict(a=1, b=2, c=3)
```

Acceder, Insertar y Borrar en Listas

Como hemos visto previamente, para acceder a los elementos de las listas y las tuplas, hemos utilizado el índice en función de la posición que ocupa cada elemento. Sin embargo, en los diccionarios necesitamos utilizar la clave para acceder al valor de cada elemento. Volviendo a nuestro ejemplo, para obtener el valor indexado por la clave **'c'** bastará con ejecutar la siguiente sentencia:

```
>>> diccionario 'c']
3
```

Para modificar el valor de un diccionario, basta con acceder a través de su clave:

```
>>> diccionario['b'] = 28
```

Añadir un nuevo elemento es tan sencillo como modificar uno ya existente, ya que si la clave no existe, automáticamente Python la añadirá con su correspondiente valor. Así pues, la siguiente sentencia insertará un nuevo valor en nuestro diccionario ejemplo:

```
>>> diccionario['d'] = 4
```

Tres son los métodos principales que nos permiten iterar sobre un diccionario: **items()**, **values()** y **keys()**. El primero nos da acceso tanto a claves como a valores, el segundo se encarga de devolvernos los valores, y el tercero y último es el que nos devuelve las claves del diccionario. Veamos estos métodos en acción sobre el diccionario original que declaramos previamente:

```
>>> for k, v in diccionario.items():
...     print("clave={0}, valor={1}".format(k, v))
...
clave=a, valor=1
clave=b, valor=2
clave=c, valor=3
>>> for k in diccionario.keys():
...     print("clave={0}".format(k))
...
clave=a
clave=b
clave=c
>>> for v in diccionario.values():
...     print("valor={0}".format(v))
...
valor=1
valor=2
valor=3
```

Por defecto, si iteramos sobre un diccionario con un bucle for, obtendremos las claves del mismo sin necesidad de llamar explícitamente al método **keys()**:

```
>>> for k in diccionario:
...     print(k)
a
b
C
```

A través del método **keys()** y de la función integrada **list()** podemos obtener una lista con todas las claves de un diccionario:

```
>>> list(diccionario.keys())
```

Análogamente es posible usar **values()** junto con la función **list()** para obtener un lista con los valores del diccionario. Por otro lado, la siguiente sentencia nos

devolverá una lista de tuplas, donde cada una de ellas contiene dos elementos, la clave y el valor de cada elemento del diccionario:

```
>>> list(diccionario.items())
[ ('a', 1), (’b', 2), ('c', 3)]
```

La función integrada **del()** es la que nos ayudará a eliminar un valor de un diccionario. Para ello, necesitaremos pasar la clave que contiene el valor que deseamos eliminar. Por ejemplo, para eliminar el valor que contiene la clave **'c'** de nuestro diccionario, basta con ejecutar:

```
>>> del(diccionario['b'])
```

El método **pop()** también puede ser utilizado para borrar eliminar elementos de un diccionario. Su funcionamiento es análogo al explicado en el caso de las listas.
Otra función integrada, en este caso **len()**, también funciona sobre los diccionarios, devolviéndonos el número total de elementos contenidos.
El operador in en un diccionario sirve para comprobar si una clave existe. En caso afirmativo devolverá el valor **True** y **False** en otro caso:

```
>>> 'x' in diccionario
False
```

Comprensión en Diccionarios
De forma similar a las listas, los diccionarios pueden también ser creados por comprensión. El siguiente ejemplo muestra cómo crear un diccionario utilizando la iteración sobre una lista:

```
>>> {k: k+1 for k in (1, 2, 3)}
{1: 2, 3: 4, 4: 5}
```

La comprensión de diccionarios puede ser muy útil para inicializar un diccionario a un determinado valor, tomando como claves los diferentes elementos de una lista. Veamos cómo hacerlo a través del siguiente ejemplo que crea un diccionario inicializándolo con el valor 1 para cada clave:

```
>>> {clave: 1 for clave in ['x', 'y', 'z']}
{'x': 1, ’y': 1, 'z': 1}
```

Orden de Diccionarios
A diferencia de las listas, los diccionarios no tienen el método **sort()**, pero sí que es posible utilizar la función integrada **sorted()** para obtener una lista ordenada de las claves contenidas. Volviendo a nuestro diccionario ejemplo inicial, ejecutaremos la siguiente sentencia:

```
>>> sorted(diccionario)
['a', 'b', 'c']
```

También podemos utilizar el parámetro reverse con el mismo resultado que en las listas:

```
>>> sorted(diccionario, reverse=True)
['c', ' b' , 'a']
```

CONVERSION DE TIPOS DE DATOS

El proceso de convertir el valor de un tipo de datos en otro tipo se denomina Conversión de tipos. Python tiene dos tipos de conversión de tipos.

- Conversión de tipo implícita
- Conversión de tipo explícito

https://www.c-sharpcorner.com/article/type-conversion-in-python/

Conversión de tipo implícita

Python convierte automáticamente un tipo de datos en otro tipo de datos. Este proceso no necesita la participación del usuario. A continuación se dan algunos ejemplos de esto.

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo
print('type of num_int is : ', type(num_int))
print('type of num_flo is : ', type(num_flo))
print('value of num_new is : ', num_new)
print('type of num_new is : ', type(num_new))
```

En el ejemplo anterior, según el tipo de valor de la variable, num_int es un tipo de datos entero, num_flo es un tipo de datos flotantes y, finalmente, num_new es un tipo de datos flotantes. Aquí, la conversión de tipo implícita tiene lugar porque la adición de entero y flotante da como resultado un valor flotante (el tipo de datos más pequeño se convierte automáticamente en el tipo de datos más grande).

Conversión de tipo explícita

Python define funciones de conversión de tipos como **int()**, **float()**, **str()** para convertir directamente un tipo de datos en otro. Este tipo de conversión también se llama typecasting.

- **int(a, base)**
  Esta función convierte cualquier tipo de datos en un número entero. "Base" especifica la base en la que se convierte un string si el tipo de datos es string. La Base predeterminada es 10.
- **float()**
  Esta función se utiliza para convertir cualquier tipo de datos a un número de coma flotante.
- **ord()**

Esta función se utiliza para convertir un caracter en entero (valor ascii del caracter).

- **hex()**
  Esta función se utiliza para convertir un número entero en un string hexadecimal.
- **oct()**
  Esta función convierte un número entero en un string octal.
- **tuple()**
  Esta función se utiliza para convertir un tipo de datos en una tupla.
- **set()**
  Esta función convierte una secuencia, colección u objeto iterable, en un conjunto.
- **list()**
  Esta función se utiliza para convertir cualquier tipo de datos en un tipo lista.
- **dict()**
  Esta función se utiliza para convertir una tupla de orden (clave, valor) en un diccionario.
- **str()**
  Esta función se utiliza para convertir un valor (entero o flotante) en un string.

OBJETOS Y ATRIBUTOS

Python es un lenguaje de programación orientado a objetos, y en Python todo es un objeto. En lenguajes de programación orientados a objetos como Python, un objeto es una entidad que contiene datos junto con metadatos y/o funcionalidades asociados. En Python todo es un objeto, lo que significa que cada entidad tiene algunos metadatos (llamados atributos) y una funcionalidad asociada (llamados métodos). Se accede a estos atributos y métodos mediante la sintaxis de punto (.)

Podemos ver los atributos y métodos de un objeto con la función **dir()**. Posteriormente podemos manipular un atributo a través de la sintaxis de punto ya mencionada. Cómo todo en Python es un objeto, podemos importar un módulo e intentar manipular uno de sus atributos. Un módulo muy popular de la librería estándar de Python es **datatime**, para manejo de fechas. Podemos cargarlo, revisar todos sus atributos y métodos y luego modificar su atributo MAXYEAR:

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__spec__',
'date', 'datetime', 'datetime_CAPI', 'sys', 'time', 'timedelta',
'timezone', 'tzinfo']
>>> datetime.MAXYEAR
9999
>>> datetime.MAXYEAR=8888
>>> datetime.MAXYEAR
8888
```

2.2.2.- Sentencias Básicas de Control

Al igual que otros lenguajes de programación, Python incorpora una serie de sentencias de control. Entre ellas, encontramos algunas tan básicas y comunes a otros lenguajes como **if/else**, **while**, **for,** y otras específicas como **pass** y **with.** A continuación, echaremos un vistazo a cada una de estas sentencias.

IF, ELSE, ELIF

La sentencia **if/else** funciona evaluando la condición indicada, si el resultado es **True** se ejecutará la siguiente sentencia o sentencias, en caso negativo se ejecutarán las sentencias que aparecen a continuación del else. Recordemos que Python utiliza la indentación para establecer sentencias que pertenecen al mismo bloque. Además, en el carácter dos puntos "**:**" indica el comienzo de bloque. A continuación, vemos un ejemplo:

```
x = 4
y = 0
if x == 4:
    y = 5
else:
    y = 2
```

Obviamente, también es posible utilizar solo la sentencia **if** para comprobar si se cumple una determinada condición y actuar en consecuencia. Además, podemos anidar diferentes niveles de comprobación a través de **elif** :

```
if X = = 4 :
    y = 1
elif x = = 5
    y = 2
elif x = = 6
    y = 3
else:
    y = 5
```

Como el lector habrá podido observar y a diferencia de otros lenguajes de programación, los paréntesis para indicar las condiciones han sido omitidos. Para Python son opcionales y habitualmente no suelen ser utilizados. Por otro lado, a pesar de que Python emplea la indentación, también es posible escribir una única sentencia a continuación del final de la condición. Así pues, la siguiente línea de código es válida:

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

FOR Y WHILE

Para iterar contamos con dos sentencias que nos ayudarán a crear bucles, nos referimos a **for** y a **while**.

for

La primera de ellas aplica una serie de sentencias sobre cada uno de los elementos que contiene el objeto sobre el que aplicamos la sentencia for. Python incorpora una función llamada **range()** que podemos utilizar para iterar sobre una serie de valores. Por ejemplo, echemos un vistazo al siguiente ejemplo:

```
>>> for x in range(1, 4):
...     print(x)
...
1
2
3
```

Asimismo, es muy común iterar a través de **for** sobre los elementos de una tupla o de una lista:

```
>>> lista = ["uno", "dos", "tres"]
>>> cad = ""
>>> for ele in lista:
...     cad += ele
...
>>> cad
"unodostres"
```

Opcionalmente, **for** admite la sentencia **else**. Si esta aparece, todas las sentencias posteriores serán ejecutadas si no se encuentra otra sentencia que provoque la salida del bucle. Por ejemplo, en la ejecución de un bucle for que no contiene ningún break, siempre serán ejecutadas las sentencias que pertenecen al else al finalizar el bucle. A continuación, veamos un ejemplo para ilustrar este caso:

```
>>> for item in (1, 2, 3):
...     print(item)
...
    else:
...     print ("fin")
...
1
2
3
fin
```

while

Otra sentencia utilizada para iterar es **while**, la cual ejecuta una serie de sentencias siempre y cuando se cumpla una determinada condición o condiciones.

Para salir del bucle podemos utilizar diferentes técnicas. La más sencilla es cambiar la condición o condiciones iniciales para así dejar que se cumplan y detener la

iteración. Otra técnica es llamar directamente a **break** que provocará la salida inmediata del bucle. Esta última sentencia también funciona con for. A continuación, veamos un ejemplo de cómo utilizar **while**:

```
>>> x = 0
>>> y = 3
>>> while x < y:
...     print(x)
...     x += 1
0
1
2
```

Al igual que for, **while** también admite opcionalmente else. Observemos el siguiente código y el resultado de su ejecución:

```
>>> x = 0
>>> y = 3
>>> while x < y:
...     print(x)
...     x+ = 1
...     if x == 2:
...         break
...     else:
...         print("x es igual a 2")
0
1
```

Si en el ejemplo anterior eliminamos la sentencia **break**, comprobaremos cómo la última sentencia **print** es ejecutada.
Además de **break**, otra sentencia asociada a **for** y **while** es **continue**, la cual se emplea para provocar un salto inmediato a la siguiente iteración del bucle. Esto puede ser útil, por ejemplo, cuando no deseamos ejecutar una determinada sentencia para una iteración concreta. Supongamos que estamos iterando sobre una secuencia y solo queremos imprimir los números pares:

```
>>> for i in range(1, 10):
...     if i % 2 != 0:
...         continue
...     print(i)
2
4
6
8
```

pass y with

Python incorpora una sentencia especial para indicar que no se debe realizar ninguna acción. Se trata de **pass** y especialmente útil cuando deseamos indicar que

no se haga nada en una sentencia que requiere otra. Por ejemplo, en un sencillo **while**:

```
>>> while True:
...     pass
```

Muchos desarrolladores emplean **pass** cuando escriben esqueletos de código que posteriormente rellenarán. Dado que inicialmente no sabemos qué código contendrá una determinada sentencia, es útil emplear **pass** para mantener el resto del programa funcional. Posteriormente, el esqueleto de código será rellenado con código funcional y la sentencia **pass** será reemplazada por otras que realicen una función específica.

La sentencia **with** se utiliza con objetos que soportan el protocolo de manejador de contexto y garantiza que una o varias sentencias serán ejecutadas automáticamente. Esto nos ahorra varias líneas de código, a la vez que nos garantiza que ciertas operaciones serán realizadas sin que lo indiquemos explícitamente. Uno de los ejemplos más claros es la lectura de las líneas de un archivo de texto. Al terminar esta operación siempre es recomendable cerrar el archivo. Gracias a with esto ocurrirá automáticamente, sin necesidad de llamar al método **close()**. Las siguientes líneas de código ilustran el proceso:

```
>>> with open(r'info.txt') as myfile:
...     for line in myfile:
...         print(line)
```