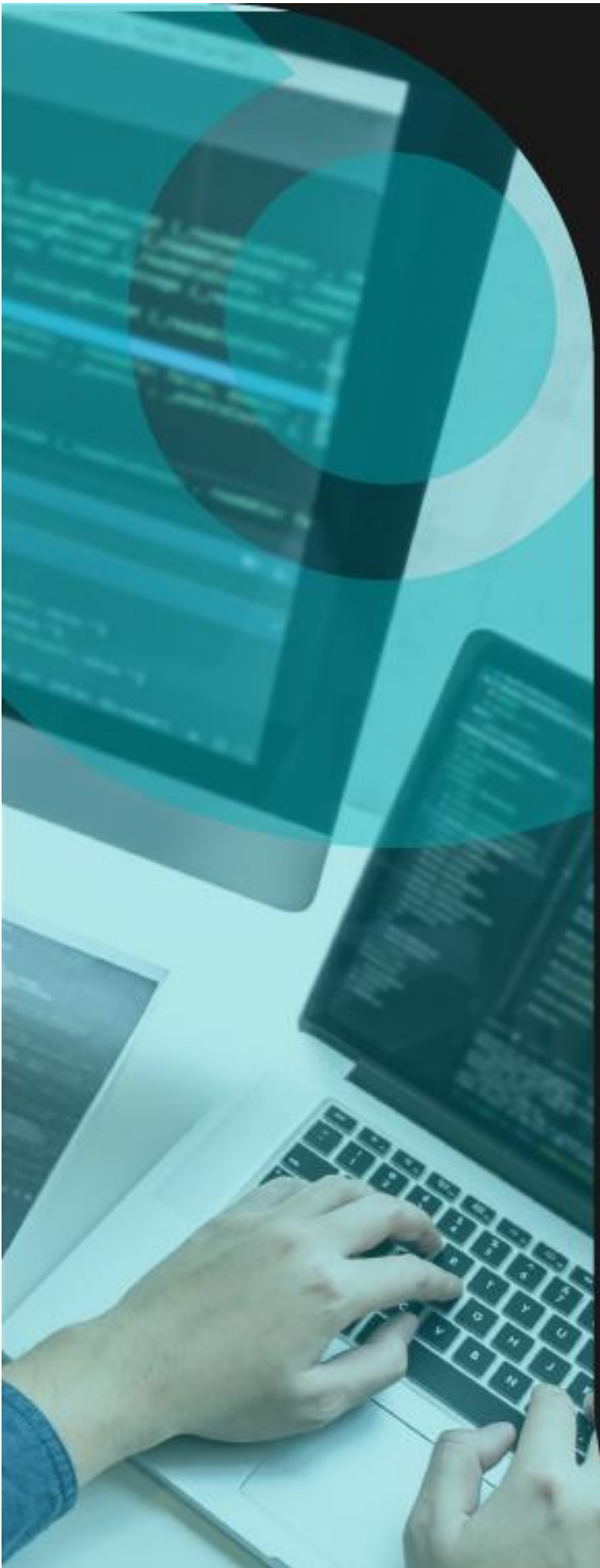

AWAKELAB

BOOTCAMP

Estructura de Datos

Objetivo de la jornada

1. Identifica las características de las distintas estructuras de dato para la resolución de problemas.
2. Utiliza operaciones de creación y acceso a los elementos de una estructura de datos acorde al lenguaje Python para resolver un problema.
3. Utiliza operaciones para la agregación, modificación y eliminación de elementos de una estructura de datos acorde al lenguaje Python para resolver un problema.

3.5.- Utilizar estructuras de datos apropiadas para la elaboración de un algoritmo que resuelve un problema acorde al lenguaje Python

3.5.1.- Estructuras de datos en Python.

Las computadoras almacenan y procesan datos con una velocidad y precisión extraordinarias. Por lo tanto, es muy importante que los datos se almacenen de manera eficiente y se pueda acceder a ellos rápidamente. Además, el procesamiento de datos debe ocurrir en el menor tiempo posible y sin perder precisión. Las **estructuras de datos** son la forma en cómo se organizan los datos y se mantienen en la memoria cuando un programa los procesa.

En el apartado 3.2.1. Hicimos una revisión general de los tipos de datos en Python y además tuvimos una primera aproximación a estructuras de datos como tuplas, listas, diccionarios. Una estructura de datos es una colección de elementos de datos (como números o caracteres, o incluso otras) que está estructurado de alguna manera, como por ejemplo enumerando los elementos de ésta. La estructura de datos más básica en Python es la secuencia. A cada elemento de una secuencia se le asigna un número, llamado su posición o índice. El primer índice es cero, el segundo índice es uno y así sucesivamente. Algunos lenguajes de programación enumeran sus elementos de secuencia comenzando con el número uno, pero la convención de indexación desde cero tiene una interpretación natural que permite que índices negativos signifiquen una indexación

envolvente de la secuencia desde el final de ésta. Puede encontrarse extraño este tipo de indexación, sin embargo con la práctica se adopta rápidamente.

3.5.1.1. SECUENCIAS

Python tiene varios tipos de secuencias integradas. Nos enfocamos inicialmente en los dos más comunes: **listas** y **tuplas**. Los **strings** también son secuencias, están compuestos por caracteres individuales los utilizaremos de manera simple en descripciones de listas y tuplas, y los cubriremos en detalle más adelante.

La principal diferencia entre listas y tuplas es que, tal como hemos visto anteriormente, una lista puede cambiar y una tupla no. Esto significa que una lista puede ser útil si necesitamos agregar elementos a medida que nuestro programa se ejecuta, mientras que una tupla puede ser útil si no necesitamos que contenga datos que cambian durante la ejecución. Las razones de esto último suelen ser bastante técnicas. Éstas tienen que ver con la eficiencia y el cómo funcionan las cosas internamente en Python. Es por eso que muchas funciones nativas de Python entregan tuplas como resultados. Para nuestros desarrollos, lo más probable es que podamos utilizar listas en lugar de tuplas en casi todos los casos.

Las secuencias son útiles cuando deseamos trabajar con colecciones de valores. Por ejemplo, podemos tener una secuencia que representa a una persona en una base de datos, siendo el primer elemento su nombre y el segundo su edad. Si escribimos estos elementos como una lista (Separados por comas y encerrados entre corchetes), se verá de la siguiente forma:

```
>>> alberto = ['Alberto Einstein', 46]
```

Pero las secuencias también pueden contener otras secuencias, por lo que podría hacer una lista de esas personas, a la que llamaremos base de datos, **db**.

```
>>> albert = ['Albert Einstein', 46]
>>> donald = ['Donald Knut', 30]
```

```
>>> db = [['Alberto Einstein', 46], ['Donald Knut', 30]]
>>> db
[['Alberto Einstein', 46], ['Donald Knut', 30]]
```

Contenedores

Python posee un concepto básico, un tipo de estructura de datos llamada **contenedor**, que es básicamente cualquier objeto que puede contener otros objetos. Los dos tipos principales de contenedores son **secuencias** (como **listas** y **tuplas**) y **mapeos** (como **diccionarios**). Mientras que los elementos de una secuencia están numerados, cada elemento en un mapeo tiene un nombre (también llamado clave). Aprenderá más sobre mapeos en el Capítulo 4. Para un ejemplo de un tipo de contenedor que no es una secuencia ni un mapeo, vea la discusión de conjuntos en el Capítulo 10.

Operando Secuencias

Hay ciertas acciones que pueden realizarse con todos los tipos de secuencia. Estas operaciones incluyen indexación, segmentación, agregar, multiplicar y verificar la membresía. Además, Python tiene funciones integradas para encontrar la longitud de una secuencia o sus elementos mayores y menores.

Una operación importante de las secuencias, que no cubriremos aquí es la iteración, que hemos utilizado cuando estudiamos los loops **for**.

Indexación De Secuencias

Todos los elementos de una secuencia están numerados, desde cero en adelante. Puede accederse a ellos individualmente con un número que es el índice, así:

```
>>> saludo = 'Hola'
>>> saludo[0]
'H'
```

Esto se llama indexación. Utilizamos un índice para buscar un elemento. Todas las secuencias se pueden indexar de esta manera. Cuando se usa un índice negativo, Python cuenta desde la derecha, es decir, desde el último elemento. El último elemento está en la posición –1.

```
>>> saludo[-1]
'a'
```

Los literales de string (y otros literales de secuencia, para el caso) se pueden indexar directamente, sin usar una variable para referirse a ellos. El efecto es exactamente el mismo.

```
>>> 'Hola'[1]
'o'
```

Si una llamada a una función devuelve una secuencia, puede indexarse directamente. Por ejemplo, si simplemente estamos interesados en el cuarto dígito de un año ingresado por el usuario, podemos hacer algo como:

```
>>> cuarto = input('Año: ')[3]
Año: 2005
>>> cuarto
'5'
```

El siguiente ejemplo corresponde a un programa que pide al usuario un año, un mes (Un número del 1 al 12) y un día (Entre 1 a 31) y luego imprime la fecha con el nombre correcto del mes.

secuencias_indexacion.py

```
meses = [
   'Enero',
   'Febrero',
   'Marzo',
   'Abril',
   'Mayo',
   'Junio',
   'Julio',
   'Agosto',
   'Septiembre',
   'Octubre',
   'Noviembre',
   'Deciembre'
]
anio   = input('Año: ')
mes    = input('Mes  (1-12): ')
```

```
dia     = input('Día (1-31): ')
mes_numero = int(mes)
dia_numero = int(dia)
# Recuerde restar 1 del mes y el día para obtener un índice correcto
mes_nombre = meses[mes_numero-1]
print(str(dia_numero) + ' de ' + mes_nombre + ' de ' + agno)
```

Un ejemplo de uso de este programa sería:

```
$ python secuencias_indexacion.py
Año: 2020
Mes  (1-12): 4
Día (1-31): 4
4 de Abril de 2020
```

donde la última línea es la salida entregada por el programa.

Sub-Secuencias (Slicing)

Del mismo modo que usamos la indexación para acceder a elementos individuales, podemos extraer subsecuencias para acceder a rangos de elementos. Esto lo realizamos usando dos índices, separados por dos puntos.

```
>>> elemento = '<a href="http://www.python.org"Sitio de Python</a>'
>>> elemento[9:30]
'http://www.python.org'
>>> elemento[32:-4]
'Sitio de Python'
```

Como podemos ver, es muy útil extraer partes de una secuencia. Para esto la numeración es muy importante. El primer índice es el número del primer elemento que desea incluir. Sin embargo, el último índice es el número de la posición del elemento siguiente al que queremos extraer en último lugar. Como ejemplo consideremos lo siguiente:

```
>>> numeros = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numeros[3:6]
[3, 4, 5]
```

```
>>> numeros[0:1]
[0]
```

En resumen, proporciona dos índices como límites para su segmento, análogo a lo que sería un intervalo matemático cerrado por la izquierda y abierto por la derecha **[límite_inferior, límite_superior[**.

En caso que no incluyamos el límite inferior o superior en nuestro rango de índices de la Subsecuencia requerida, no se limitará la secuencia en ese sentido. Por ejemplo,

```
>>> numeros[5:]
[5, 6, 7, 8, 9, 10]
>>> numeros[:5]
[0, 1, 2, 3, 4]
>>> numeros[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Es posible extraer subsecuencias que no obtengan números adyacentes de la lista original sino espaciados con un paso entregado como parámetro. Hasta los ejemplos anteriores hemos omitido este parámetro y se ha asumido por defecto el valor 1. Podemos saltar elementos de manera regular si especificamos un valor mayor a 1 para éste parámetro. Por ejemplo si queremos obtener los números de la última lista con un **paso=2** podemos escribir alguna de estas opciones:

```
>>> numeros[0:11:2]
[0, 2, 4, 6, 8, 10]
>>> numeros[::2]
[0, 2, 4, 6, 8, 10]
```

Podemos utilizar un paso negativo para obtener saltos en casos de obtención de subsecuencias invertidas:

```
>>> numeros[11:0:-2]
[10, 8, 6, 4, 2]
>>> numeros[::-2]
[10, 8, 6, 4, 2, 0]
```

Concatenando Secuencias

Las secuencias se pueden concatenar con el operador de suma (+).

```
>>> [1, 2, 3] + [4, 5,  6]
[1, 2, 3, 4, 5, 6]
>>> 'Hola,' + 'mundo!'
'Hola, mundo!'
>>> [1, 2, 3] + 'mundo!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

Como se puede ver en el mensaje de error, no es permitido concatenar una lista y un string, aunque ambas son secuencias. En general, no es posible concatenar secuencias de diferentes tipos.

Multiplicando Secuencias

Multiplicar una secuencia por un número **x** crea una nueva secuencia donde la secuencia original se repite **x** veces:

```
>>> 'python' * 5
'pythonpythonpythonpythonpython'
>>> [42] * 10
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

A continuación revisamos un programa que imprime (en pantalla) una "caja" formada por caracteres, que está centrada en pantalla y adaptado al tamaño de una frase proporcionada por el usuario. El código puede parecer complicado pero es básicamente cálculo de espacio en pantalla: averiguar cuántos espacios, guiones, etc., necesita para colocar los elementos en pantalla correctamente.

```
>>>frase = input("Ingrese una frase: ")
>>>ancho_pantalla = 80
>>>ancho_texto = len(frase)
>>>ancho_caja = ancho_texto + 6
>>>margen_izquierdo  = (ancho_pantalla - ancho_caja) // 2
>>>print()
>>>print(' ' * margen_izquierdo + '+'  + '-' * (ancho_caja -2)  +  '+')
>>>print(' ' * margen_izquierdo + '|  ' + ' ' * ancho_texto    + '  |')
>>>print(' ' * margen_izquierdo + '|  ' +     frase     + '  |')
>>>print(' ' * margen_izquierdo + '|  ' + ' ' * ancho_texto    + '  |')
>>>print(' ' * margen_izquierdo + '+'  + '-' * (ancho_caja -2)  +  '+')
>>>print()
```

Lo que nos entregaría lo siguiente en un caso de ejecución:

```
$ python secuencias_vacias.py
Ingrese una frase: Estamos probando código

          +--------------------------+
          |                          |
          |  Estamos probando codigo  |
          |                          |
          +--------------------------+
```

Membresía en Secuencias

Para comprobar si un valor es parte de una secuencia, se utiliza el operador **in**. Este operador es un poco diferente de los discutidos hasta ahora (como multiplicación o suma). Comprueba si algo es cierto y devuelve un valor de acuerdo a eso, tal como lo vimos en las expresiones booleanas del apartado de estructuras condicionales **if**. A continuación se muestran algunos ejemplos de uso de este operador:

Los dos primeros ejemplos utilizan la prueba de pertenencia para comprobar si se encuentran **'w'** y **'x'**, respectivamente en el string de permisos. Esto podría ser un script en una máquina Linux que verifica permisos de un archivo:

```
>>> permisos = 'rw'
>>> 'w' in permisos
True
>>> 'x' in permisos
False
```

El siguiente ejemplo comprueba si un nombre de usuario proporcionado (**'Marco'**) se encuentra en una lista de usuarios. Esto podría ser útil si su programa aplica alguna política de seguridad, donde probablemente se utilice también para verificar contraseñas:

```
>>> usuarios = ['Marco', 'José', 'Luis']
>>> input('Ingrese su nombre: ') in usuarios
Ingrese su nombre: Marco
True
```

El último ejemplo verifica si el asunto de la cadena contiene el string '$$$'. Esto podría usarse, por ejemplo, como parte de un filtro de spam:

```
>>> asunto = '$$$ Hágase rico ahora !!! $$$'
>>> '$$$' in asunto
True
```

Podríamos realizar un ejemplo un poco más sofisticado en el que verifiquemos la existencia de un par de credenciales dentro de una secuencia (En este caso una Lista). Aprovechamos de utilizar lo aprendido sobre loops anteriormente para insistir hasta que el usuario ingrese las credenciales correctas:

```
db = [
   ['alberto', '1234'],
   ['marcela', '4242'],
   ['julia', '7524'],
   ['analía', '9843']
]

while True:
   usuario = input('Usuario: ')
   password = input('Contraseña: ')
   if [usuario, password] in db:
      print('Acceso permitido!')
      break
```

```
else:
    print('Contraseña incorrecta!')
```

Longitud, Mínimo y Máximo de una Secuencia

Las funciones **len()**, **min()** y **max()** pueden son de gran utilidad y permiten lo siguiente:

a. La función **len()** devuelve el número de elementos contiene una secuencia.
b. La función **min()** devuelve el elemento de menor valor de una secuencia numérica.
c. La función **max()** devuelve el elemento de mayor valor de una secuencia numérica.

A continuación se muestran algunos ejemplos de estas funciones:

```
>>> numeros = [100, 34, 678]
>>> len(numeros)
3
>>> max(numeros)
678
>>> min(numeros)
34
>>> max(2,3)
3
>>> min(9,3,2,5)
2
```

3.5.1.2. LISTAS

En los ejemplos anteriores, hemos utilizado listas como parte de la revisión del concepto de secuencias. En esta sección cubriremos características más específicas de las listas en comparación a tuplas y strings, que tienen que ver principalmente con su característica de **mutabilidad**.

LA FUNCIÓN LIST()

Dado que tuplas y strings no se pueden modificar de la misma manera que las listas, a veces puede resultar útil crear una lista a partir de un string. Si intentamos de modificar un elemento de un string o de una tupla obtendremos errores como se muestra a continuación:

```
>>> saludo = "'Hola cómo estás?'"
>>> saludo[1]='k'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

>>> numeros = (1,2,3,4,5,6)
>>> numeros[3]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Para evitar esto podemos convertir a listas estas estructuras de datos con la función **list()**. Con esto podremos modificar los ítems que intentamos previamente:

```
>>> saludo_lista = list(saludo)
>>> saludo_lista
["'", 'H', 'o', 'l', 'a', ' ', 'c', 'ó', 'm', 'o', ' ', 'e', 's', 't', 'á', 's', '?', "'"]

>>> numeros_lista = list(numeros)
>>> numeros_lista
[1, 2, 3, 4, 5, 6]
```

OPERACIONES BÁSICAS CON LISTAS

Podemos realizar todas las operaciones de secuencia estándar en listas, como indexar, segmentar, concatenar, y multiplicar. Lo interesante en las listas es que se pueden modificar. En esta sección, veremos algunas formas en las que podemos modificar una lista: asignaciones de elementos, eliminación de elementos, asignaciones de sublistas y métodos de lista. Debemos tener en cuenta que no necesariamente todos los métodos de las lista cambian sus elementos.

Modificando Listas: Asignación de Elementos

Cambiar una lista es fácil. Simplemente se usa la asignación con indexación de paréntesis cuadrados para indicar la posición del elemento que deseamos modificar. Por ejemplo, en los últimos ejemplos en que intentamos modificar un string y una tupla, luego de haberlos convertido a listas podemos intentar las modificaciones nuevamente y observar el resultado:

```
>>> saludo_lista[1]='k'
>>> saludo_lista
['"', 'k', 'o', 'l', 'a', ' ', 'c', 'ó', 'm', 'o', ' ', 'e', 's', 't', 'á', 's', '?', '"']

>>> numeros_lista[3] = 1
>>> numeros_lista
[1, 2, 3, 1, 5, 6]
```

Modificando Listas: Eliminación de Elementos

Eliminar elementos de una lista también es fácil. Simplemente podemos usar la instrucción **del**. Por ejemplo si deseamos borrar el último elemento de numeros_lista hacemos:

```
>>> del numeros_lista[-1]
>>> numeros_lista
[1, 2, 3, 4, 5]
```

Modificando Listas: Asignando Sublistas

Cortar listas es una utilidad muy poderosa, y se hace aún más poderosa considerando que podemos asignar valores a las sublistas que logramos con ella.

```
>>> nombre = list('Pelo')
>>> nombre
['P', 'e', 'l', 'o']
>>> nombre[2:] = list('ra')
>>> nombre
['P', 'e', 'r', 'a']
```

Cuando usamos asignaciones de sublistas, también podemos reemplazar el segmento seleccionado con una secuencia cuya longitud sea diferente a la de la original.

```
>>> nombre = list('Perl')
>>> nombre[1:] = list('Python')
>>> nombre
['P', 'y', 't', 'h', 'o', 'n']
```

Las asignaciones de sublistas se pueden usar incluso para insertar elementos sin reemplazar ninguno de los originales.

```
>>> nnmeros = [1, 5]
>>> nnmeros[1:1] = [2, 3, 4]
>>> nnmeros
[1, 2, 3, 4, 5]
```

Aquí, básicamente reemplazamos un segmento vacío, insertando una sublista. Podemos hacer lo contrario para eliminar una sublista.

```
>>> numeros
[1, 2, 3, 4, 5]
>>> numeros[1:4] = []
>>> numeros
[1, 5]
```

Se sugiere al lector experimentar con distintas listas y formas de indexación para reemplazo o inserción de elementos, pues son muchas las posibilidades y con la práctica se adquiere habilidad sobre este aspecto.

MÉTODOS DE LISTAS

Un método es una función que está estrechamente acoplada a algún objeto, ya sea una lista, un número, una cadena o lo que sea. En en general, un método se invoca así:

```
object.method(argumentos)
```

Una llamada a un método se parece a una llamada a una función, excepto que el objeto se coloca antes del nombre del método, separándolos con un punto (Esto lo revisaremos más en detalle en un apartado más avanzado relacionado con Programación Orientada a Objetos). Las listas tienen varios métodos que le permiten examinar o modificar su contenido.

Método	Descripción	Ejemplo con objeto: lista= [2,'b','a']

append()	Agrega un elemento al final de la lista	>>> lista.append('María')
		>>> lista
		[2, 'b', 'a', 'María']

clear()	Elimina todos los elementos de la lista	>>> lista.clear()
		>>> lista
		[]

copy()	Crea una copia de la lista	>>> copia_lista = lista.copy()
		>>> copia_lista
		[2, 'b', 'a']

count()	Entrega el índice del elemento en la lista. especificado como argumento.	>>> lista.count('b') 1

extend()	Agrega elementos de una lista u otra secuencia (pasado como argumento) a la lista del método.	>>> lista.extend(('pedro','lily'))
		>>> lista
		[2, 'b', 'a', 'pedro', 'lily']
		
		>>> lista.extend(['pedro','lily'])
		>>> lista
		[2, 'b', 'a', 'pedro', 'lily']

index()	Entrega el índice del primer elemento con el valor que se especifique como argumento.	>>> lista.index('a') 2

insert()	Agrega un elemento en una posición determinada.	>>> lista.insert(2, 'c')
		>>> lista
		[2, 'b', 'c', 'a']

pop()	Elimina el elemento especificado en la posición indicada en el argumento.	>>> lista.pop(2)
		'a'
		>>> lista
		[2, 'b']

remove()	Elimina el primer elemento con el valor especificado en el argumento.	>>> lista.remove('b')
		>>> lista
		[2, 'a']

reverse()	Invierte el orden de la lista.	>>> lista.reverse()
		>>> lista
		['a', 'b', 2]

sort()	Ordena la lista	>>> lista_texto=lista[1:3]
		>>> lista_texto
		['b', 'a']
		>>> lista_texto.sort()
		>>> lista_texto
		['a', 'b']

3.5.1.3. TUPLAS

Las tuplas son secuencias, como las listas. La única diferencia es que las tuplas no se pueden modificar, lo que también es así para los strings. La sintaxis de una tupla es simple: elementos separados por comas y delimitados por paréntesis redondos. Si separamos algunos valores con comas, automáticamente tenemos una tupla.

```
>>> 1, 2, 3
(1, 2, 3)
```

También pueden crearse incluyendo los paréntesis:

```
>>> (1, 2, 3)
(1, 2, 3)
```

Una tupla vacía se escribe como dos paréntesis que no contienen nada.

```
>>> ()
()
```

Para crear una tupla que contenga un solo valor debemos incluir una coma, aunque solo haya un valor.

```
>>> 42,
(42,)
>>> (42,)
(42,)
```

Una coma es crucial y puede cambiar el significado de toda una expresión:

```
>>> 3 * (40 + 2)
126
>>> 3 * (40 + 2,)
(42, 42, 42)
```

La función de **tuple()** funciona prácticamente de la misma manera que la **list()**: toma una secuencia como argumento y lo convierte en una tupla. Si el argumento ya es una tupla, se devuelve sin cambios.

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple('abc')
('a B C')
>>> tuple((1, 2, 3))
(1, 2, 3)
```

Las tuplas no tienen mayor complejidad y no hay mucho más que hacer con ellas, excepto crearlas y acceder a sus elementos, y se hace igual que con otras secuencias.

```
>>> x=1,2,3
>>> x[1]
2

>>> x[0:2]
(1, 2)
```

Las subtuplas de una tupla también son tuplas, al igual que los sublistas de una lista son en sí mismas listas.

Hay dos razones importantes por las que se necesitan conocer las tuplas.

- Pueden utilizarse como claves en mapeos (y miembros de conjuntos); las listas no se pueden usar de esta forma.
- Son devueltas por algunas funciones y métodos nativos de Python, lo que significa que debemos lidiar con ellas. Siempre que no intentemos cambiarlos, "lidiar" con ellos la mayoría de las veces significa tratarlos como listas (A menos que necesitemos métodos como **index()** y **count()**, que las tuplas no tienen).

En general, el uso de listas probablemente serán suficientes para las necesidades que podamos tener en relación a secuencias.

3.5.1.4. CONJUNTOS (SETs)

Definir y operar con conjuntos matemáticos también es posible en Python. La función para crear un conjunto se llama **set()** y acepta como argumentos una serie de valores pasados entre comas, como si se tratara de una cadena de texto. Un conjunto es una colección que no está ordenada ni indexada. En Python, los conjuntos se escriben con llaves.

```
>>> frutas = {"manzana", "banana", "frambuesa"}
>>> frutas
{'frambuesa', 'banana', 'manzana'}
```

Los conjuntos no tienen orden, por lo que no podemos estar seguros de en qué orden aparecerán los elementos.

Existen otras formas de crear conjuntos, como la siguiente línea de código que define un conjunto de tres números diferentes:

```
>>> conjunto = set('846')
>>> conjunto
{'8', '6', '4'}
```

Podemos crear un conjunto con elementos numéricos de la siguiente forma:

```
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Accediendo Elementos de un Conjunto

No podemos acceder a los elementos de un conjunto haciendo referencia a un índice, ya que los conjuntos no están ordenados y los elementos no tienen índice. Sin embargo, podemos recorrer los elementos del conjunto utilizando un loop **for**, o preguntando si un valor especificado está presente en un conjunto, utilizando la palabra clave **in**.

```
>>> frutas = {"manzana", "banana", "frambuesa"}
>>> for fruta in frutas:
...     print(fruta)
...
frambuesa
banana
manzana

>>> 'banana' in frutas
True
```

Longitud, Mínimo y Máximo de un Conjunto

De la misma forma que en el caso de secuencias, en los conjuntos podemos obtener su longitud, elemento de valor mínimo y elemento de valor máximo, con **len()**, **min()** y **max()**, respectivamente.

```
>>> conjunto = {2,6,3,7,6,6,6,78,3}
>>> conjunto
{2, 3, 6, 7, 78}
>>> len(conjunto)
5
>>> min(conjunto)
2
>>> max(conjunto)
78
```

MÉTODOS DE CONJUNTOS

Método	Descripción	Ejemplo en base a conjunto: conjunto = {'b','c',2,4,'t'}

add()	Agrega un elemento al conjunto.	>>> conjunto.add('perro')
		>>> conjunto
		{'c', 2, 4, 't', 'b', 'perro'}

clear()	Elimina todos los elementos del conjunto.	>>> conjunto.clear()
		>>> conjunto
		set()

copy()	Devuelve una copia del conjunto.	>>> copia_conjunto = conjunto.copy()
		>>> copia_conjunto
		{'c', 2, 4, 't', 'b'}

difference()	Devuelve un conjunto que es la diferencia entre el objeto conjunto y un conjunto pasado como argumento.	>>> conjunto.difference({'b',2,'t'}) {'c', 4}

difference_update()	Elimina del objeto conjunto el item que también existe en el conjunto pasado como argumento.	>>> conjunto.difference_update({'b','c','e'})
		>>> conjunto
		{2, 4, 't'}

discard()	Elimina el elemento especificado.	>>> conjunto.discard(2)
		>>> conjunto
		{'c', 4, 't', 'b'}

intersection()	Devuelve un conjunto que es la intersección de éste conjunto con otro pasado como argumento.	>>> conjunto.intersection({'b','c','e'}) {'c', 'b'}

intersection_update()	Elimina los elementos del set que no están presentes en otros set especificado como argumento.	>>> conjunto.intersection_update({'b','c','e'})
		>>> conjunto
		{'c', 'b'}

isdisjoint()	Devuelve una evaluación de si dos conjuntos tienen intersección (False) o no (True).	>>> conjunto.isdisjoint({1,3,6}) True
issubset()	Devuelve una evaluación de si otro conjunto especificado como argumento contiene a este conjunto.	>>> conjunto.issubset({'b', 'c', 2, 4, 't', 5,6,7}) True
issuperset()	Devuelve una evaluación de si este conjunto contiene a otro conjunto especificado como argumento.	>>> conjunto.issuperset({2,4}) True

pop()	Elimina aleatoriamente un elemento de este conjunto.	>>> conjunto.pop()
		'c'
		>>> conjunto
		{2, 4, 't', 'b'}

remove()	Elimina el elemento especificado como argumento.	>>> conjunto.remove('t')
		>>> conjunto
		{'c', 2, 4, 'b'}

symmetric_difference()	Devuelve un conjunto que contiene todos los elementos de ambos conjuntos, pero no los elementos que están presentes en ambos conjuntos.	>>> conjunto.symmetric_difference({'luisa','mesa','auto',2}) {'c', 4, 't', 'b', 'mesa', 'auto', 'luisa'}
symmetric_difference_update()	Actualiza el conjunto original eliminando elementos que están presentes en ambos conjuntos e insertando los demás elementos.	>>> conjunto.symmetric_difference_update({'luisa','mesa','auto',2}) >>> conjunto {'c', 4, 't', 'b', 'mesa', 'auto', 'luisa'}
union()	Devuelve un conjunto que es la unión de este conjunto con otro	>>> conjunto.union({'luisa','mesa','auto',2}) {'c', 2, 4, 't', 'b', 'mesa', 'auto', 'luisa'}

	pasado como argumento.	

update()	Actualiza el conjunto con la unión de éste con otro conjunto pasado como argumento.	>>> conjunto.update({'luisa','mesa','auto',2})
		>>> conjunto
		{'c', 2, 4, 't', 'b', 'mesa', 'auto', 'luisa'}

3.5.1.5. DICCIONARIOS

Hemos visto que las listas son útiles cuando se desea agrupar valores en una estructura y hacer referencia a cada valor por medio de un índice. En esta sección, revisaremos una estructura de datos en la que se puede hacer referencia a cada valor por su nombre. Este tipo de estructura se llama mapeo. La única estructura de datos de tipo mapeo incorporado nativamente en Python es el **diccionario**. Los valores en un diccionario no tienen ningún orden en particular, pero cada uno se almacenan bajo una estructura de clave-valor, que puede ser un número, un string, o incluso una tupla.

USOS DE UN DICCIONARIO

El nombre **diccionario** nos entrega una intuición sobre el propósito de esta estructura. Un libro típico está hecho para leer de principio a fin. Si lo deseamos, podemos abrirlo rápidamente en cualquier número de página determinado. Esto es análogo a una lista en Python. Por otro lado, los diccionarios, tanto los reales como su equivalente en Python, se construyen para que pueda buscar una palabra específica (clave) fácilmente para encontrar su definición (valor).

Un diccionario es más apropiado que una lista en muchas situaciones. A continuación se muestran algunos ejemplos de usos de diccionarios de Python:

- Representar el estado de un tablero de juego, donde cada clave es una tupla de coordenadas.
- Almacenamiento de fecha-hora de modificación de archivos, usando como claves los nombres de los archivos.
- Un teléfono digital / libreta de direcciones.

Supongamos que tenemos una lista de personas.

```
>>> nombres = ['Alicia', 'Bety', 'Cecilia', 'Dina', 'Earnesto']
```

Podríamos desear crear una pequeña base de datos donde pudiéramos almacenar los números de teléfono de estas personas. ¿Cómo haríamos esto?

Una forma sería hacer otra lista. Supongamos que estamos almacenando solo sus números cortos de cuatro dígitos. Entonces obtendremos algo como esto:

```
>>> numeros = ['2341', '9102', '3158', '0142', '5551']
```

Una vez creadas estas listas, y con los conocimientos que poseemos hasta el momento, podemos buscar el número de teléfono de Cecilia de la siguiente manera:

```
>>> numeros[nombres.index ('Cecilia')]
'3158'
```

Funciona, pero es poco práctico. Lo que realmente nos gustaría hacer es algo como lo siguiente:

```
>>> agenda_telefonica['Cecilia']
'3158'
```

Justamente, si definimos agenda_telefonica como un diccionario, podemos hacerlo de esta forma.

Creando y Utilizando Diccionarios
Para crear un diccionario utilizamos la siguiente sintaxis:

```
agenda_telefonica = {'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258'}
```

Los diccionarios constan de pares (Llamados ítems) de claves y sus valores correspondientes. En este ejemplo, los nombres son las claves y los números de teléfono son los valores. Cada clave está separada de su valor por dos puntos (:), los ítems están separados por comas y todo está

entre llaves. Un diccionario vacío (sin ningún ítem) está escrito con solo dos llaves: **{}**.

Las claves son únicas dentro de un diccionario (Y en cualquier otro tipo de estructura de mapeo). Los valores no necesariamente son únicos dentro de un diccionario.

La Función dict()

Podemos utilizar la función **dict()** para construir diccionarios a partir de otros mapeos (Por ejemplo, otros diccionarios) o de secuencias de pares (clave, valor).

```
>>> itemes = [('nombre', 'Gaby'), ('edad', 42)]
>>> d = dict(itemes)
>>> d
{'edad': 42, 'nombre': 'Gaby'}
>>> d ['nombre']
'Gaby'
```

También se puede utilizar con argumentos de palabras clave, de la siguiente manera:

```
>>> d = dict(nombre = 'Gumby', edad = 42)
>>> d
{'edad': 42, 'nombre': 'Gumby'}
```

Si se usa sin argumentos **dict()** devuelve un nuevo diccionario vacío, al igual que otras funciones similares como **list()**, **tuple()** y **str()**.

Operaciones Básicas de Diccionarios

El comportamiento básico de un diccionario refleja en muchos sentidos el de una secuencia.

- **len(d)** devuelve el número de elementos (pares clave-valor) en **d**.
- **d[k]** devuelve el valor asociado con la clave **k**.
- **d[k] = v** asocia el valor **v** con la clave **k**.
- **del d[k]** borra el elemento con la clave **k**.
- **k** in **d** comprueba si hay un elemento en **d** que tiene la clave **k**.

Aunque los diccionarios y las listas comparten varias características comunes, existen algunas distinciones importantes:

Tipos de clave: las claves del diccionario no tienen que ser números enteros (aunque pueden serlo). Pueden ser de cualquier tipo inmutable, como números de punto flotante (reales), strings o tuplas.

Incorporación automática: Podemos asignar un valor a una clave, incluso si esa clave no está en el diccionario para empezar; en ese caso, se creará un nuevo elemento. En el caso de listas, no podemos asignar un valor a un índice fuera del rango de la lista (Sin usar append u otro método similar). Lo indicado lo podemos verificar en el siguiente ejemplo:

```
>>> x = []
>>> x[42] = 'Algún contenido'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range

>>> x = {}
>>> x[42] = 'Algún contenido'
>>> x
{42: 'Algún contenido'}
```

Pertenencia: La expresión **k** i**n d** (Donde **d** es un diccionario) busca una clave, no un valor. En el caso de listas, la expresión **v** en **l** (Donde **l** es una lista) busca un valor, no un índice.

Ejemplo práctico: Agenda Telefónica.
A continuación mostramos la construcción de un programa que implementa, con los conocimientos que acumulamos hasta el momento, una agenda telefónica. Se incluyen comentarios a lo largo del código, que explican lo que se implementa en cada sección:

```
# Base de datos simple
# Un diccionario con nombres de personas como claves.
# Cada persona es representada como otro diccionario
# con las claves 'tel' y 'dir' refiriendose a numero
```

```
# telefonico y direccion, respectivamente
personas = {
   'Alicia': {
      'tel': '2341',
      'dir': 'calle ancha 23'
   },
   'Bety': {
      'tel': '9102',
      'dir': 'calle verde 42'
   },
   'Cecilia': {
      'tel': '3158',
      'dir': 'avenida nueva 90'
   }
}

# Etiquetas descriptivas para el numero telefonico y la direccion.
# Seran usadas cuando se imprima la salida del programa.
etiquetas = {
   'tel': 'El Número telefónico',
   'dir': 'La Dirección'
}

nombre = input('Nombre: ')

# ¿Buscamos un numero telefonico o una direccion?
requerimiento = input('¿Número Telefónico (t) o Dirección (d)? ')

# Uso de la clave correcta:
if requerimiento == 't': clave = 'tel'
if requerimiento == 'd': clave = 'dir'

# Solo intentar imprimir informacion si el nombre es una
# clave valida en el diccionario

if nombre in personas:
   print("{} de {} es {}."
         .format(
               etiquetas[clave],
               nombre,
```

```
                personas[nombre][clave])
                )
else:
    print("La información no existe en la agenda.")
```

A continuación algunos ejemplos de ejecución de este programa:

```
$ python agenda_telefonica.py
Nombre: Alicia
¿Número Telefónico (t) o Dirección (d)? t
El Número telefónico de Alicia es 2341.

$ python agenda_telefonica.py
Nombre: Cecilia
¿Número Telefónico (t) o Dirección (d)? d
La Dirección de Cecilia es avenida nueva 90.

$ python agenda_telefonica.py
Nombre: Rosa
¿Número Telefónico (t) o Dirección (d)? d
La información no existe en la agenda.
```

Formato de Strings en Diccionarios

Un diccionario puede contener todo tipo de información, y nuestro string de formato solo seleccionará lo que necesite. Tendremos que especificar que estamos entregando un mapeo, utilizando **format_map()**.

```
>>> agenda_telefonica = {'Alicia': '2341',
... 'Bety': '9102',
... 'Cecilia': '3258'
... }
>>> print("El número de teléfono de Cecilia es {Cecilia}"
... .format_map(agenda_telefonica))
'El número de teléfono de Cecilia es 3258'
```

Al usar diccionarios como éste, puede tener cualquier número de especificadores de conversión, siempre que todas las claves se encuentren en el diccionario. Este tipo de string de formato puede ser

muy útil en sistemas de plantillas, por ejemplo en el siguiente caso de plantilla HTML:

```
plantilla = '''
<! DOCTYPE html>
<html lang="es">
 <head><title>{titulo}</title></head>
 <body>
  <h1>{titulo}</h1>
  <p>{texto}</p>
 </body>
</html>
'''
datos = {
   'titulo': 'Mi página de inicio',
   'texto': '¡Bienvenido a mi página de inicio!'
   }
print(plantilla.format_map(datos))
```

Con lo que obtenemos el siguiente resultado:

```
<! DOCTYPE html>
<html lang="es">
 <head><title>Mi página de inicio</title></head>
 <body>
  <h1>Mi página de inicio</h1>
  <p>¡Bienvenido a mi página de inicio!</p>
 </body>
</html>
```

Este ejemplo es muy importante porque es la primera conexión que hacemos entre los conocimientos actuales de Python con apartados anteriores que han cubierto los tópicos de HTML.
Lo realizado por el último ejemplo es muy similar a lo que haremos con el motor de plantillas del Framework de desarrollo Web basado en Python, llamado Django, que estudiaremos en módulos posteriores.

MÉTODOS DE DICCIONARIOS

Método	Description	Ejemplo en base a: agenda_telefonica = {'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258'}

clear()	Elimina todos los elementos del diccionario.	>>> agenda_telefonica.clear()
		>>> agenda_telefonica
		{}

copy()	Devuelve una copia del diccionario.	>>> copia_agenda = agenda_telefonica.copy()
		>>> copia_agenda
		{'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258'}

fromkeys()	Retorna un diccionario con las claves y valor especificados.	>>> {}.fromkeys(['name', 'age'])
		{'age': None, 'name': None}
		
		>>> {}.fromkeys(['name', 'age'],1)
		{'name': 1, 'age': 1}

get()	Retorna el valor de la clave especificada.	>>> print(agenda_telefonica
		... .get('Alicia'))
		2341
		>>> print(agenda_telefonica
		... .get('Luisa'))
		None
		Nótese la diferencia con:
		>>> agenda_telefonica['Luisa']
		Traceback (most recent call last):
		File "<stdin>", line 1, in <module>
		KeyError: 'Luisa'

items()	Retorna una lista que contiene una tupla para cada par clave, valor.	>>> agenda_telefonica.items() dict_items([('Alicia', '2341'), ('Bety', '9102'), ('Cecilia', '3258')])
keys()	Retorna una lista que contiene las claves del diccionario.	>>> agenda_telefonica.keys() dict_keys(['Alicia', 'Bety', 'Cecilia'])

pop()	Elimina el elemento con la clave especificada.	>>> agenda_telefonica.pop('Betty')
		'9102'
		>>> agenda_telefonica
		{'Alicia': '2341', 'Cecilia': '3258'}

popitem()	Elimina el último par clave, valor insertado.	>>> agenda_telefonica['Rosa']= '2456'

		>>> agenda_telefonica
		{'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258', 'Rosa': '2456'}
		>>> agenda_telefonica.popitem()
		('Rosa', '2456')
		>>> agenda_telefonica
		{'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258'}

setdefault()	Retorna el valor de la clave especificada. Si la clave no existe, inserta una clave con el valor especificado.	>>> agenda_telefonica.setdefault('Alicia')
		'2341'
		>>> agenda_telefonica.setdefault('Alicia','2222')
		'2341'
		>>> agenda_telefonica
		{'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258'}
		>>> agenda_telefonica.setdefault('Luis','2222')
		'2222'
		>>> agenda_telefonica
		{'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258', 'Luis': '2222'}

update()	Actualiza el diccionario con los pares clave, valor especificados.	>>> agenda_telefonica ... .update({'Luis':'2345', 'Samy':'3453'}) >>> agenda_telefonica {'Alicia': '2341', 'Bety': '9102', 'Cecilia': '3258', 'Luis': '2345', 'Samy': '3453'}
values()	Retorna una lista de todos los valores presentes en el diccionario.	>>> agenda_telefonica.values() dict_values(['2341', '9102', '3258'])

Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[3] Eric Matthes, Python Crash Course, 2016.

[4] Magnus Lie Hetland, Beginning Python: From Novice to Professional, 2017.

[5] Python - Data Structure.
https://www.tutorialspoint.com/python_data_structure/index.htm

[6] Python Tutorial.
https://www.w3schools.com/python/