GOBIERNO DE ESPAÑA
MINISTERIO DE HACIENDA Y FUNCIÓN PÚBLICA
INAP
INSTITUTO NACIONAL DE ADMINISTRACIÓN PÚBLICA

Introducción a la programación en Python

Tema 4. Tipos estructurados

Autor: Francisco Mariño Ruiz

Contenido

1 Tuplas

1.1 Definición

Una tupla es un tipo de dato estructurado que nos permite agrupar variables de distinto tipo. Una tupla se definirá con todos sus elementos y será inmutable, es decir, los elementos que estén dentro de ella no podrán variar durante el ciclo de vida de la variable.

Por ejemplo, podemos crear una tupla en la que se almacene una fecha:

```
>>> fecha = (8, "mayo", 2018)
```

También podemos crear una tupla que almacene los datos de un objeto:

```
>>> clase = (4571, “Variables complejas”, "Programación en Python")
```

E incluso podemos almacenar tuplas dentro de otras tuplas:

```
>>> clase = (4571, “Variables complejas”, "Programación en Python", (8, "mayo", 2018))
```

Tal y como hemos dicho, la tupla es invariable desde su creación. Por lo tanto, si intentamos añadir un nuevo valor, borrar algún valor existente o sustituir uno de los valores que contiene nos devolverá un error.

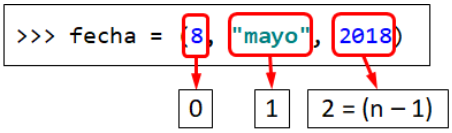
```
>>> fecha[0] = 30


Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

1.2 Métodos de las tuplas

Las tuplas aceptan los métodos genéricos de Python como son, por ejemplo:

- *Len()*: Devuelve la longitud de una variable.
- *Str()*: Transforma una variable en una cadena de texto.
- *Type()*: Devuelve el tipo al que pertenece una variable.

Las tuplas, al igual que las cadenas permite el acceso por posición de elemento, para ello estableceremos entre corchetes el número de la posición a la que queremos acceder. Es importante reseñar que las tuplas, como toda variable que tenga "n" elementos, empieza a contar sus elementos por la posición "0" y acaba en la posición "n-1", de tal modo que si intentamos acceder al elemento de la posición "n" nos devolverá un error ya que este elemento no existirá. Para conocer el tamaño de una tupla podemos usar el método genérico "*len(objeto)*". Por ejemplo:



En el siguiente ejemplo retomamos la tupla "*clase*" creada en el punto anterior:

```
>>> clase = (4571, "Variables complejas", "Programación en Python", (8, "mayo", 2018))
```

Si tratamos de acceder a la posición 1:

```
>>> clase[1]
'Variables complejas'
```

Si tratamos de acceder a la posición [3]:

```
>>> clase[3]
 (8, 'mayo', 2018)
```

A pesar de que pueda dar lugar a equívocos, esta tupla tiene cuatro elementos:

- 4571
- "Variables complejas"
- "Programación en Python"
- (8, "mayo", 2018)

Aunque el último elemento sea una tupla, éste no extiende la longitud de la tupla en la que se aloja, si no que cuenta como un solo elemento. Si quisiésemos acceder a cualquiera de las posiciones de la fecha utilizaríamos un doble índice:

```
>>> clase[3][0]
 8
```

Respecto a los métodos propios, las tuplas sólo tienen dos:

Método	Descripción
Count(valor)	Devuelve el número de apariciones del valor introducido.
Index(valor)	Devuelve la posición en la tupla del valor introducido. En caso de que se pida un valor que no exista devolverá un error.

A continuación, podemos ver unos ejemplos de uso de dichos métodos. Para todos ellos partiremos de la tupla fecha que hemos definido en el punto anterior:

```
>>> fecha = (8, "mayo", 2018)
```

En el primer ejemplo vamos a aplicar el método "*index()*" con el valor "*8*" para que nos devuelva la posición en la que dicho valor se encuentra en la tupla, en este caso nos devolverá que se encuentra en la posición cero.

```
>>> fecha.index(8)
0
```

En el segundo ejemplo vamos a aplicar el método "*count*" para que nos devuelva el número de apariciones del valor "*8*". Nos devuelve 1.

```
>>> fecha.count(8)
1
```

En el tercer ejemplo vamos a ver cuándo buscamos un valor que no existe en la tupla. En caso de que utilicemos el método "*index*", si el valor que buscamos no existe nos devolverá un error.

```
>>> fecha.index(2)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
```

En caso de que utilicemos el método "*count*" con un valor que no existe, obtendremos como resultado el número de ocurrencias real, es decir cero.

```
>>> fecha.count(2)
0
```

1.3 Definiciones especiales de tuplas

Se pueden crear tuplas vacías, aunque a priori no tengan mucha utilidad, pueden ser el resultado vacío de una función (veremos más adelante que es una función y que devuelve). Para ello basta con definirla con los paréntesis vacíos.

```
>>> tupla = ()
```

También podemos crear tuplas de un solo elemento. Aunque a priori pueda parecer que no son de mucha utilidad, debemos recordar la principal propiedad de las tuplas, la invariabilidad de sus elementos. Una tupla nos permitiría mantener un valor inalterable durante todo el ciclo de vida de un programa sin que otros elementos puedan alterarlo.

A la hora de definir una tupla unitaria no basta con poner un dato entre paréntesis, pues en ese caso lo reconocerá como un solo dato del tipo que sea y no una tupla. Para definir la tupla unitaria se requiere el elemento seguido de una coma.

```
# No es una tupla:
>>> tupla_unitaria = (2018)
>>> len(tupla_unitaria)
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()


# Si es una tupla:
>>> tupla_unitaria = (2018, )
>>> len(tupla_unitaria)
1
```

1.4 Empaquetado y desempaquetado

Otras operaciones que podemos hacer con las tuplas, son las operaciones de empaquetado y desempaquetado de tuplas. Si tenemos tres variables simples podemos empaquetarlas como una tupla con una asignación directa de valores separados por comas de manera muy cómoda:

```
>>> a = 8
>>> b = "mayo"
>>> c = 2018

>>> d = a,b,c

>>> d
(8, 'mayo', 2018)

>>> len(d)
3
```

Y del mismo modo, podemos desempaquetar las tuplas asignando sus valores a variables:

```
>>> f, g, h = d

>>> f
8

>>> g
'mayo'

>>> h
2018
```

Este tipo de operaciones son muy usadas para devolver múltiples variables en la salida de funciones, de tal modo que las labores de entrada/salida entre el código principal y la función se limite a un mapeo de variables.

Es importante recordar que a la hora de hacer el desempaquetado de variables hay que ser muy cuidadoso, ya que si las variables de asignación no son distintas se perderán los datos y si el número de variables no es exactamente el número de elementos de la tupla, el proceso fallará y obtendremos un error.

```
>>> p, p, p = d
>>> p
2018

>>> f, g = d
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: too many values to unpack (expected 2)

>>> f, g, h, i = d
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: not enough values to unpack (expected 4, got 3)
```

2 Listas

2.1 Definición

Las listas son el segundo tipo estructurado que vamos a estudiar, poseen una estructura muy semejante a la de tuplas. A diferencia de las tuplas, las listas se pueden usar para almacenar datos compuestos cuya cantidad y valor puede variar a lo largo del ciclo de vida de la propia lista, por lo que estas estructuras ya no se consideran inmutables. Otra característica importante es que posee una gran cantidad de métodos propios que las hacen muy útiles a la hora de diseñar algoritmos complejos.

Para declarar una lista se usará la notación de valores separados por comas y encerrados entre corchetes. Es importante recordar que las listas, al igual que las tuplas, permiten almacenar datos del mismo tipo o combinación de distintos tipos de datos. Esto es una característica de Python, ya que, en la mayoría de los lenguajes, las listas son de un mismo tipo de dato.

Otra característica importante de las listas y que a su vez es una clave del lenguaje Python es el hecho de que las listas anidadas sean la manera de almacenar datos matriciales, ya que, al contrario de la mayoría de lenguajes de programación, Python no soporta las entidades matriciales de manera nativa, lo que obliga a usar listas anidadas para simular el comportamiento matricial tal y como lo conocemos o acudir a librerías científicas que si lo permiten.

```
>>> alumnos = [“Luis”, "Juan", "Alberto"]
```

Al igual que con las tuplas, podemos consultar su longitud con la función *len()* o podemos acceder a posiciones concretas utilizando los índices de posición, con valores que van desde cero a la propia longitud de la lista -1.

```
>>> alumnos = [“Luis”, "Juan", "Alberto"]
>>> alumnos[0]
“Luis”


>>> len(alumnos)
3


>>> alumnos[3]
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> alumnos[2]
"Alberto"
```

Apoyándonos en los índices de posición podemos hacer lo que se denomina “slice” de listas, es decir, extraer sublistas de las listas. Para ello utilizaremos la notación de rangos, al igual que se usa en las cadenas de texto. Veamos unos ejemplos con la lista alumnos:

```
>>> alumnos = [“Luis”, "Juan", "Alberto"]
```

Si queremos acceder a un subconjunto desde el primer elemento hasta el segundo, tendremos que decir el índice del primer elemento al que queremos acceder y el índice del primer elemento que no queremos que se nos devuelva:

```
>>> alumnos[0:2]
["Luis”, “Juan"]
```

Podemos ver que el primer que nos devuelve es el que está en la posición cero (“*Luis*”) y el primer elemento que no nos devuelve es el que está en la posición 2 (“*Alberto*”).

El “*slice*” también nos permite que cuando deseemos obtener todos los elementos desde el principio no poner índice de inicio:

```
>>> alumnos[:2]
["Luis", "Juan"]
```

O bien, si queremos obtener desde uno cualquiera hasta el final, no poner el último índice:

```
>>> alumnos[1:]
["Juan", "Alberto"]
```

En caso de que estemos trabajando con listas variables, también nos facilitan el acceso a los subconjuntos utilizando índices negativos, que en lugar de contar de izquierda a derecha, cuenta de derecha a izquierda:

```
>>> alumnos[:-1]
["Luis", "Juan"]
```

2.2 Métodos de inserción y modificación

Como comentamos anteriormente, las listas son mutables y para hacerlo posible, Python incorpora una serie de funciones que nos permite cambiar, agregar o quitar valores de una lista entre otras cosas.

Para cambiar un valor, bastará con seleccionar el índice de la lista al que se quiere acceder y el nuevo valor que se le quiere dar.

```
>>> alumnos = [“Luis”, "Juan", "Alberto"]
>>> alumnos [1] = "Pedro"
>>> alumnos
["Luis", "Pedro", “Alberto”]
```

Para agregar un valor, podemos hacerlo de dos maneras, o bien agregándolo al final de la lista, para lo que utilizaremos el método *append()*.

```
>>> alumnos.append("Juan")
>>> alumnos
["Luis", "Pedro", “Alberto”, “Juan”]
```

O bien podemos insertar un valor en una posición determinada y desplazar el resto de elementos de la lista hacia la derecha mediante el comando *insert()*.

```
>>> alumnos.insert(2, "Ángel")
>>> alumnos
["Luis", "Pedro", “Ángel”, “Alberto”, “Juan”]
```

También podemos agregar a la vez varios elementos procedentes de otra lista mediante el comando *extend()* que agrega los elementos de la lista que se pasa como parámetro a la lista sobre la que se aplica el método.

```
>>> alumnos.extend(["Jesús", “Antonio”])
>>> alumnos
["Luis", "Pedro", “Ángel”, “Alberto”, “Juan”, “Jesús”, “Antonio”]
```

2.3 Métodos de eliminación

Para eliminar un elemento de la lista, podemos utilizar el comando *remove()* con el valor que queremos eliminar como parámetro, lo que nos permite eliminar un valor determinado sin conocer su posición.

```
>>> alumnos.remove("Ángel")
>>> alumnos
["Luis", "Pedro", “Alberto”, “Juan”, “Jesús”, “Antonio”]
```

Si le damos como parámetro un valor que no está en la lista, nos devolverá un error.

```
>>> alumnos.remove("Ángel")
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

En caso de que tengamos más de una entrada en la lista con el mismo valor, solo borrará la primera ocurrencia de la misma que encuentre en la lista.

```
>>> alumnos.append(“Antonio”)
["Luis", "Pedro", “Alberto”, “Juan”, “Jesús”, “Antonio”, “Antonio”]

>>> alumnos.remove("Antonio")
["Luis", "Pedro", “Alberto”, “Juan”, “Jesús”, “Antonio”]
```

Otro comando para eliminar un elemento de una lista, es el comando *pop()* que nos permite borrar un elemento conociendo su posición en la lista (pasaremos dicha posición como parámetro).

```
>>> alumnos.pop(1)
>>> alumnos
["Luis", “Alberto”, “Juan”, “Jesús”, “Antonio”]
```

En caso de no pasar ningún valor, borrará el último elemento de la lista.

```
>>> alumnos.pop()
>>> alumnos
["Luis", “Alberto”, “Juan”, “Jesús”]
```

Y en caso de pasar un índice mayor al número de elementos de la lista nos devolverá un error.

```
>>> alumnos.pop(5)
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

2.4 Métodos de búsqueda

Otros métodos importantes en lo que se refiere a listas de elementos, son los métodos de búsqueda. Tenemos la opción de obtener un booleano, que nos especifica si un valor está o no está en una lista mediante el operador de comparación *in*, lo cual será muy útil para evaluar condicionales.

```
>>> alumnos = ["Luis", "Pedro", “Alberto”, “Juan”]
>>> "Luis" in alumnos
True
>>> "Javier" in alumnos
False
```

Por otro lado, podemos saber la posición en la que está un determinado elemento mediante el comando *index()*.

```
>>> alumnos.index(“Alberto”)
2
```

Siguiendo la tendencia de otros métodos, si preguntamos por un elemento que no está en la lista, obtendremos un error.

```
>>> alumnos.index(“Javier”)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: 'Javier' is not in list
```

Cabe señalar, que en el caso del índice, si el valor que buscamos está repetido, el índice que se nos devolvería es sólo el de la primera aparición.

```
>>> alumnos.append("Luis")
>>> print(alumnos)
["Luis", "Pedro", “Alberto”, “Juan”, “Luis”]
>>> alumnos.index(“Luis”)
0
```

En caso de que quisiésemos otra aparición que no fuese la primera, podemos añadir parámetros opcionales, que son la posición de inicio de búsqueda y la posición de fin de búsqueda. Si realmente sólo queremos omitir el principio, podríamos ponerlo sin el segundo índice.

```
>>> alumnos.index(“Luis”, 1, 5)
4
```

También podemos utilizar el método *“count()”* para contar las apariciones de un elemento dentro de la lista.

```
>>> alumnos.count("Luis")
2
```

En ocasiones, querremos recorrer todos los elementos de una lista para operar con ellos o sobre ellos, utilizaremos los bucles *for* con la siguiente sintaxis:

```
>>> for alumno in alumnos:
...     print(alumno)
...
Luis
Pedro
Alberto
Juan
```

O Bien como se explicó en módulos anteriores, especificando un rango (cuando el inicio es cero, dicho valor es opcional):

```
>>> for i in range (0, len(alumnos)):
...     print(alumnos[i])
...
Luis
Pedro
Alberto
Juan
```

2.5 Métodos de ordenación

Podemos ordenar los elementos de una lista de distintas maneras, en primer lugar, podemos usar el comando *reverse()* que invierte el orden actual de los elementos.

```
>>> alumnos.reverse()
>>> print(alumnos)
[“Juan”, “Alberto”, "Pedro", "Luis"]
```

Podemos ordenar las listas con el comando *sort()*, que por defecto nos ordena los elementos alfabéticamente (en caso de elementos textuales) o de menor a mayor (en caso de elementos numéricos).

```
>>> alumnos.sort()
>>> print(alumnos)
[“Alberto”, “Juan”, "Luis", "Pedro"]
```

O bien podemos usar el mismo comando con el valor *reverse=True* que nos permite ordenar en orden contrario.

```
>>> alumnos.sort(reverse=True)
>>> print(alumnos)

["Pedro", "Luis", “Juan”, “Alberto”]
```

2.6 Otras consideraciones

Cabe destacar que cuando asignamos una lista a una variable, lo que realmente estamos asignando a la variable es la dirección de memoria donde se almacenan las variables. Esto es importante porque en caso de que queramos hacer una "copia" de una lista para mantener los datos mientras operamos sobre la copia, todos los cambios que hagamos en cualquiera de las dos listas repercutirán en ambas listas. En el siguiente ejemplo puedes verlo con más claridad:

```
>>> alumnos = ["Luis", "Pedro", "Alberto", "Juan"]
>>> ex_alumnos = alumnos
print(ex_alumnos)
["Luis", "Pedro", "Alberto", "Juan"]
>>> alumnos.pop()
print(alumnos)
["Luis", "Pedro", "Alberto"]
print(ex_alumnos)
["Luis", "Pedro", "Alberto"]
```

Para evitar esto, crearíamos una nueva lista y utilizando un bucle for rellenaríamos la lista elemento a elemento:

```
>>> alumnos = ["Luis", "Pedro", "Alberto", "Juan"]
>>> ex_alumnos = []
>>> for alumno in alumnos:
>>>     ex_alumnos.append(alumno)
print(ex_alumnos)
["Luis", "Pedro", "Alberto", "Juan"]
>>> alumnos.pop()
print(alumnos)
["Luis", "Pedro", "Alberto"]
print(ex_alumnos)
["Luis", "Pedro", "Alberto", "Juan"]
```

2.7 Ejercicios propuestos

A continuación, se propone una serie de ejercicios para aplicar los conceptos estudiados sobre listas. Las soluciones a dichos ejercicios, se encuentran al final del documento, pero se recomienda al alumno que trate de desarrollarlos.

2.7.1 Ejercicio 1

Diseña un programa que pida la longitud de una lista nueva y a continuación permita introducir por teclado ese número de elementos. El programa imprimirá la lista antes de finalizar.

```
¿Que longitud tendrá la lista? 3

Introduce un elemento: Alberto

Introduce un elemento: Luis

Introduce un elemento: Mercedes

['Alberto', 'Luis', 'Mercedes']


Process finished with exit code 0
```

2.7.2 Ejercicio 2

Tomando como referencia el programa del ejercicio 1. Diseña un programa que pida longitud y elementos de una lista y a continuación permita contar las apariciones de un elemento.

```
¿Que longitud tendrá la lista? 4

Introduce un elemento: Luis

Introduce un elemento: Mercedes

Introduce un elemento: Ana

Introduce un elemento: Luis

¿Cual es el elemento a buscar? Luis

2


Process finished with exit code 0
```

2.7.3 Ejercicio 3

Diseña un programa que lea una lista de números de la longitud que deseemos, la imprima por pantalla, posteriormente eliminará de la lista todos los números impares y mostrará la lista resultante por pantalla.

```
¿Que longitud tendrá la lista? 8

Introduce un número: 1

Introduce un número: 3

Introduce un número: 5

Introduce un número: 8

Introduce un número: 7

Introduce un número: 4

Introduce un número: 6

Introduce un número: 5

[1, 3, 5, 8, 7, 4, 6, 5]

[8, 4, 6]


Process finished with exit code 0
```

2.7.4 Ejercicio 4

Diseña un programa que lea una lista de la longitud que deseemos, la imprima por pantalla, posteriormente nos preguntará que elemento queremos eliminar y lo eliminará. Finaliza mostrando la lista una vez eliminado el elemento.

```
¿Que longitud tendrá la lista? 4

Introduce un elemento: Ana

Introduce un elemento: Eva

Introduce un elemento: Blanca

Introduce un elemento: Mercedes

['Ana', 'Eva', 'Blanca', 'Mercedes']

¿Cual es el elemento a buscar? Ana

['Eva', 'Blanca', 'Mercedes']


Process finished with exit code 0
```

2.7.5 Ejercicio 5

Diseña un programa que lea una lista de la longitud que deseemos, la imprima por pantalla, posteriormente eliminará los elementos repetidos e imprimirá de nuevo la lista.

```
¿Que longitud tendrá la lista? 5

Introduce un elemento: Ana

Introduce un elemento: Eva

Introduce un elemento: Paula

Introduce un elemento: Eva

Introduce un elemento: Ana

['Ana', 'Eva', 'Paula', 'Eva', 'Ana']

['Ana', 'Eva', 'Paula']


Process finished with exit code 0
```

3 Diccionarios

3.1 Definición

Son variables semejantes a las listas de datos. Relacionan una clave con un valor, la clave será el identificador de cada entrada del diccionario y el valor será almacenado como una lista de datos, de este modo el acceso a los datos se hace a través de la clave en lugar del índice como sucedía en las listas. Como es evidente no se podrán repetir las claves dentro del diccionario.

Hay que tener en cuenta que las entradas de los diccionarios no estarán necesariamente ordenadas según un orden, por lo que intentar acceder a ellos con un índice de posición no tendría mucho sentido, por esto, si intentamos acceder con un índice de posición nos devolverá un error. Al acceder por una clave en lugar de una posición encontraremos más rápido los datos, ya que no tenemos que recorrer una lista entera para encontrar los datos que buscamos, esto hará que sean muy útiles y aumenten la eficiencia frente a las listas cuando tenemos que buscar en grandes volúmenes de datos.

Para definir un diccionario, utilizaremos los caracteres de llaves. Lo podemos definir vacío o con valores, en cuyo caso los valores se escribirán de la forma "*clave: valor*", pudiendo el valor ser cualquier tipo de variable (listas incluidas).

```
>>> elip = {'Hayford': [6378388.0, 297.0], 'WGS84': [6378137.0, 298.257223563],
'Struve': [6378298.0, 294.73], 'Helmert': [6378200.0, 298.3]}
>>> print (elip)
{'WGS84': [6378137.0, 298.257223563], 'Hayford': [6378388.0, 297.0],
'Helmert': [6378200.0, 298.3], 'Struve': [6378298.0, 294.73]}
```

3.2 Métodos de acceso a los datos

Para acceder a los elementos, lo haremos como hemos dicho, con la clave en lugar del índice, podremos hacerlo de dos maneras, o bien con el nombre de la clave entre corchetes o bien con el comando *get()*.

```
>>> print (elip['Hayford'])
[6378388.0, 297.0]
>>> print (elip.get('Hayford'))
[6378388.0, 297.0]
```

Sin embargo, si usamos el acceso por la clave entre corchetes y dicha clave no está en la lista se nos retornará un error, mientras que si utilizamos el comando *get()* obtendremos un *None* como respuesta, lo que nos ayudará a controlar posibles errores.

```
>>> print (elip['GRS80'])
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
KeyError: 'GRS80'
>>> print (elip.get('GRS80'))
None
```

Para recorrer todos los valores de un diccionario podemos utilizar un bucle que recorra sus claves.

```
>>> for elemento in elip:
...     print(elemento, ":", elip[elemento])
...
WGS84: [6378137.0, 298.257223563]
Hayford: [6378388.0, 297.0]
Helmert: [6378200.0, 298.3]
Struve: [6378298.0, 294.73]}
```

También podemos recorrer el diccionario de forma que sus elementos nos sean devueltos como tuplas.

```
>>> for nombre, valores in elip.items():
...     print(nombre, ":", valores)
...
WGS84: [6378137.0, 298.257223563]
Hayford: [6378388.0, 297.0]
Helmert: [6378200.0, 298.3]
Struve: [6378298.0, 294.73]}
```

Otras opciones que tenemos es solicitar la lista de claves mediante el comando *"keys()"*. Se devuelven como un objeto de tipo *"dict_keys"*.

```
>>> print(elip.keys())
dict_keys(['WGS84', 'Hayford', 'Helmert', 'Struve'])
```

O a la lista de valores mediante el comando *"values()"* que se devuelven como un objeto de tipo *"dict_values"*.

```
>>> print(elip.values())
dict_values([[6378137.0, 298.257223563], [6378388.0, 297.0], [6378200.0, 298.3],
[6378298.0, 294.73]])
```

3.3 Métodos de inserción y modificación

Para insertar un nuevo elemento en un diccionario tendremos que utilizar la notación *"diccionario[clave] = valor"* el cual nos insertará el nuevo elemento o actualizará uno existente si ya existe la clave en el diccionario.

```
>>> elip['WGS84'] = [6378137.0, 298.258]
>>> print (elip)
{'WGS84': [6378137.0, 298.258], 'Hayford': [6378388.0, 297.0],
'Helmert': [6378200.0, 298.3], 'Struve': [6378298.0, 294.73]}

>>> elip['GRS80'] = [6378137.0, 298.257222100882711243]
>>> print (elip)
{'GRS80': [6378137.0, 298.2572221008827], 'WGS84': [6378137.0, 298.258], 'Hayford':
[6378388.0, 297.0], 'Helmert': [6378200.0, 298.3], 'Struve': [6378298.0, 294.73]}
```

3.4 Métodos de eliminación

Para eliminar entradas de un diccionario, podemos usar el método *"pop(clave)"* que nos elimina el elemento cuya clave hayamos introducido, o el método *"clear()"* que nos vacía el diccionario.

```
>>> elip.pop('GRS80')
>>> print (elip)
{'WGS84': [6378137.0, 298.258], 'Hayford': [6378388.0, 297.0],
'Helmert': [6378200.0, 298.3], 'Struve': [6378298.0, 294.73]}
>>> elip.clear()
>>> print (elip)
{}
```

3.5 Ejercicios propuestos

A continuación, se propone una serie de ejercicios para aplicar los conceptos estudiados sobre diccionarios. Las soluciones a dichos ejercicios, se encuentran al final del documento, pero se recomienda al alumno que trate de desarrollarlos.

3.5.1 Ejercicio 6

Programa que lea un diccionario de la longitud que deseemos en la que almacenará datos de personas. La clave será el DNI y cada uno tendrá asignado una lista de datos de longitud variable. Finalmente se imprimirá el diccionario por pantalla.

```
¿Que longitud tendrá el diccionario? 3

Introduce la clave del nuevo elemento: 35232323Z

¿Cuantos datos va a tener el elemento? 4

Introduce un nuevo valor: Alba

Introduce un nuevo valor: García López

Introduce un nuevo valor: Madrid

Introduce un nuevo valor: 915555555

Introduce la clave del nuevo elemento: 21474747J

¿Cuantos datos va a tener el elemento? 3

Introduce un nuevo valor: Fernando

Introduce un nuevo valor: Martínez Ruiz

Introduce un nuevo valor: Getafe

Introduce la clave del nuevo elemento: 88889999T

¿Cuantos datos va a tener el elemento? 2

Introduce un nuevo valor: Luis

Introduce un nuevo valor: 9122222222

{'35232323Z': ['Alba', 'García López', 'Madrid', '915555555'], '21474747J': ['Fernando',
'Martínez Ruiz', 'Getafe'], '88889999T': ['Luis', '9122222222']}


Process finished with exit code 0
```

4 Soluciones a los ejercicios

Nota: Cómo en todos los problemas de programación, las soluciones no son únicas. En este apartado se plantean posibles soluciones. Al tratarse de ejemplos de carácter didáctico no siempre serán las soluciones óptimas.

4.1 Ejercicios de listas

4.1.1 Ejercicio 1

Diseña un programa que pida la longitud de una lista nueva y a continuación permita introducir por teclado ese número de elementos. El programa imprimirá la lista antes de finalizar.

```
1  longitud = int(input("¿Que longitud tendrá la lista? "))
2
3  lista = []
4  for i in range (0, longitud):
5      lista.append(input('Introduce un elemento: '))
6
7  print(lista)
```

```
¿Que longitud tendrá la lista? 3
Introduce un elemento: Alberto
Introduce un elemento: Luis
Introduce un elemento: Mercedes
['Alberto', 'Luis', 'Mercedes']

Process finished with exit code 0
```

En este fragmento de código se ha hecho lo siguiente:

- La línea 1 crea la variable longitud, que es donde se va a almacenar el número de elementos de la lista. Como la instrucción "*input()*" nos devuelve siempre una cadena de texto, utilizamos el conversor explícito "*int()*" para convertir la entrada en una cifra entera.
- En la línea 3 se crea una lista vacía.
- En la línea 4 se monta un bucle for que iterará desde la posición cero, hasta la posición longitud – 1.
- La línea 5, que se encuentra dentro del bucle, añade directamente el elemento a la lista. Podría hacerse en dos líneas, en una primera asignar el elemento a una variable y a continuación añadir a la lista. Cualquiera de las dos opciones sería válida.
- La línea 7 imprime la lista en pantalla.

4.1.2 Ejercicio 2

Tomando como referencia el programa del ejercicio 1. Diseña un programa que pida longitud y elementos de una lista y a continuación permita contar las apariciones de un elemento.

```
1  longitud = int(input("¿Que longitud tendrá la lista? "))
2
3  lista = []
```

```
4  for i in range (0, longitud):
5      lista.append(input('Introduce un elemento: '))
6
7  busca = input('¿Cual es el elemento a contar? ')
8  print(lista.count(busca))
```

```
¿Que longitud tendrá la lista? 4
Introduce un elemento: Luis
Introduce un elemento: Mercedes
Introduce un elemento: Ana
Introduce un elemento: Luis
¿Cual es el elemento a buscar? Luis
2

Process finished with exit code 0
```

Nos hemos apoyado en el programa anterior añadiendo la línea 7 que pide un elemento a buscar y en la línea 8 utilizamos el comando "*count()*" dentro del comando "*print()*" para que nos muestre el número de apariciones del elemento a buscar.

4.1.3 Ejercicio 3

Diseña un programa que lea una lista de números de la longitud que deseemos, la imprima por pantalla, posteriormente eliminará de la lista todos los números impares y mostrará la lista resultante por pantalla.

```
1  longitud = int(input("¿Que longitud tendrá la lista? "))
2
3  lista = []
4  for i in range (0, longitud):
5      lista.append(int(input('Introduce un número: ')))
6  print(lista)
7
8  lista_borrar = []
9  for el in lista:
10     if el%2 != 0:
11         lista_borrar.append(el)
12
13 for el in lista_borrar:
14     lista.remove(el)
15
16 print(lista)
```

```
¿Que longitud tendrá la lista? 8
Introduce un número: 1
Introduce un número: 3
Introduce un número: 5
Introduce un número: 8
Introduce un número: 7
Introduce un número: 4
Introduce un número: 6
```

```
Introduce un número: 5
[1, 3, 5, 8, 7, 4, 6, 5]
[8, 4, 6]

Process finished with exit code 0
```

Has la línea 6, el código es igual al del ejercicio 1. A partir de ahí se hacen una serie de modificaciones:

- En la línea 8 creamos una segunda lista vacía para almacenar los elementos que queremos borrar.
- En la línea 9 comenzamos un bucle que lee la lista entera.
- En la línea 10 evaluamos si son impares (resto de división entre 2 distinto de 0) y si dicha evaluación es positiva, en la línea 11 se añade a la lista.
- A continuación, en la 13 se crea un nuevo bucle que recorre la lista de elementos a borrar y en la línea 14 borramos de la lista original los elementos impares.
- Por último, imprimimos por pantalla la lista de números pares.

4.1.4 Ejercicio 4

Diseña un programa que lea una lista de la longitud que deseemos, la imprima por pantalla, posteriormente nos preguntará que elemento queremos eliminar y lo eliminará. Finaliza mostrando la lista una vez eliminado el elemento.

```
1  longitud = int(input("¿Que longitud tendrá la lista? "))
2
3  lista = []
4  for i in range (0, longitud):
5      lista.append(input('Introduce un elemento: '))
6  print(lista)
7
8  borrar = input('Introduce el elemento a borrar: ')
9
10 if borrar in lista:
11     lista.remove(borrar)
12 print(lista)
```

```
¿Que longitud tendrá la lista? 4
Introduce un elemento: Ana
Introduce un elemento: Eva
Introduce un elemento: Blanca
Introduce un elemento: Mercedes
['Ana', 'Eva', 'Blanca', 'Mercedes']
¿Cual es el elemento a buscar? Ana
['Eva', 'Blanca', 'Mercedes']

Process finished with exit code 0
```

La solución a este ejercicio es muy semejante a la del ejercicio 3. La diferencia es que en el final del código, en lugar de utilizar el elemento leído para contar su número de apariciones, lo utiliza con el comando "*remove()*" para borrarlo.

En la línea 10 hemos introducido una comprobación para saber si existe del elemento a borrar. Esto es importante, ya que como hemos comentado anteriormente si intentamos borrar de la lista algo que no existe obtendríamos un error que detendría la ejecución del programa. Aunque este caso, solo le pedimos que compruebe si existe, también podríamos añadir una instrucción para que en caso de que no exista el elemento a borrar mostrase un mensaje al usuario avisándole de que no se va a borrar nada.

4.1.5 Ejercicio 5

Diseña un programa que lea una lista de la longitud que deseemos, la imprima por pantalla, posteriormente eliminará los elementos repetidos e imprimirá de nuevo la lista.

```
1  longitud = int(input("¿Que longitud tendrá la lista? "))
2
3  lista = []
4  for i in range (0, longitud):
5      lista.append(input('Introduce un elemento: '))
6  print(lista)
7
8  for i in range (len(lista)-1, -1, -1):
9      if lista[i] in lista[:i]:
10         lista.pop(i)
11
12 print(lista)
```

```
¿Que longitud tendrá la lista? 5
Introduce un elemento: Ana
Introduce un elemento: Eva
Introduce un elemento: Paula
Introduce un elemento: Eva
Introduce un elemento: Ana
['Ana', 'Eva', 'Paula', 'Eva', 'Ana']
['Ana', 'Eva', 'Paula']

Process finished with exit code 0
```

En este ejemplo hemos introducido un concepto nuevo, un bucle invertido. Vamos a intentar explicar paso a paso el porqué de dicho bucle.

En primer lugar, debemos de tener claro que el comando "*range()*" que usamos en los bucles for, aunque generalmente lo usemos con 1 o dos parámetros, realmente tiene 3 parámetros. Los parámetros son:

- Entero de Inicio (el primero que se usa). Si es cero puede omitirse, ya que es su valor por defecto.
- Entero de fin (el primero que no se usa). No se puede omitir.
- Entero de incremento. Es el valor de incremento de valores al recorrer un bucle, en general se usa el valor por defecto "1" y por tanto no se pone.

Por ejemplo, sin utilizamos un "*range(0, 5)*" nuestro bucle pasará por los valores 0, 1, 2, 3 y 4, es decir incrementando de 1 en 1. Si utilizásemos un "*range(0, 5, 2)*" nuestro bucle pasaría por los valores 0, 2 y 4.

En el caso de nuestro ejercicio, queremos buscar y borrar los elementos repetidos y la forma más conveniente de hacerlo es recorrer la lista al revés, por tanto, utilizamos un "*range(len(lista)-1, -1, -1)*" siendo "*len(lista) – 1*" igual a 4. De este modo, nuestro bucle pasaría por los valores 4, 3, 2, 1 y 0.

Ahora bien ¿por qué utilizar un bucle inverso? La mejor manera de verlo es con un ejemplo. Supongamos la siguiente lista:

```
1  ['a', 'b', 'a', 'a', 'b', 'a', 'b', 'b']
```

Es una lista de 8 elementos por lo que en circunstancias normales utilizaríamos un bucle como el siguiente:

```
8  for i in range (0, len(lista)):
9      if lista[i] in lista[i+1:]:
10         lista.pop(i)
```

Al recorrer en este bucle pasaría lo siguiente:

- 1º Iteración:
  - i = 0 → lista[0] = 'a'
  - lista[i+1:] = lista[1:8] →['b', 'a', 'a', 'b', 'a', 'b', 'b']
  - ¿'a' dentro de lista[1:8]? → Si → borramos posición 0
  - Lista = →['b', 'a', 'a', 'b', 'a', 'b', 'b']
- 2º iteración:
  - I = 1 → lista[1] = 'a'
  - lista[i+1:] = lista[2:7] →['a', 'b', 'a', 'b', 'b']
  - ¿'a' dentro de lista[2:7]? → Si → borramos posición 1
  - Lista = →['b', 'a', 'b', 'a', 'b', 'b']
- 3º Iteración:
  - I = 2 → lista[2] = 'b'
  - lista[i+1:] = lista[3:6] →['a', 'b', 'b']
  - ¿'b' dentro de lista[3:6]? → Si → borramos posición 2
  - Lista = →['b', 'a', 'a', 'b', 'b']
- 4º Iteración:
  - I = 3 → lista[3] = 'b'
  - lista[i+1:] = lista[4:5] →['b']
  - ¿'b' dentro de lista[4:5]? → Si → borramos posición 3
  - Lista = →['b', 'a', 'a', 'b']

En estos momentos ya podemos observar un problema, al ir borrando posiciones según vamos avanzando en la lista y el bucle for avanzar de manera lineal, en cuanto borramos alguna posición intermedia y cambia la longitud de la lista y por tanto se van saltando valores.

A priori podríamos plantear otra opción, que es establecer una resta al valor i, cada vez que borre, de tal modo que el iterador no se salte posiciones. Es decir:

```
8   for i in range (0, len(lista)):
9       if lista[i] in lista[i+1:]:
10          lista.pop(i)
11          i -= 1
```

Con este bucle pasaría lo siguiente:

- 1º Iteración:
  - i = 0 → lista[0] = 'a'
  - lista[i+1:] = lista[1:8] →['b', 'a', 'a', 'b', 'a', 'b', 'b']
  - ¿'a' dentro de lista[1:8]? → Si → borramos posición 0
  - Lista = →['b', 'a', 'a', 'b', 'a', 'b', 'b']
  - i = -1
- 2º iteración:
  - I = 0 → lista[0] = 'b'
  - lista[i+1:] = lista[1:7] →['a', 'a', 'b', 'a', 'b', 'b']
  - ¿'b' dentro de lista[1:7]? → Si → borramos posición 0
  - Lista = →['a', 'a', 'b', 'a', 'b', 'b']
  - I = -1
- 3º Iteración:
  - I = 0 → lista[0] = 'a'
  - lista[i+1:] = lista[1:6] →['a', 'b', 'a', 'b', 'b']
  - ¿'a' dentro de lista[1:6]? → Si → borramos posición 0
  - Lista = ['a', 'b', 'a', 'b', 'b']
  - I = -1
- 4º Iteración:
  - I = 0 → lista[0] = 'a'
  - lista[i+1:] = lista[1:5] →['b', 'a', 'b', 'b']
  - ¿'a' dentro de lista[1:5]? → Si → borramos posición 0
  - Lista = → ['b', 'a', 'b', 'b']
  - I = -1
- 5º Iteración:
  - I = 0 → lista[0] = 'b'
  - lista[i+1:] = lista[1:4] →['a', 'b', 'b']
  - ¿'b' dentro de lista[1:4]? → Si → borramos posición 0
  - Lista = → ['a', 'b', 'b']
  - I = -1
- 6º Iteración:
  - I = 0 → lista[0] = 'a'
  - lista[i+1:] = lista[1:3] →['b', 'b']
  - ¿'a' dentro de lista[1:3]? → No → Continuamos
  - Lista = → ['a', 'b', 'b']
  - I = 0
- 7º Iteración:
  - I = 1 → lista[1] = 'b'
  - lista[i+1:] = lista[2:3] →['b']
  - ¿'b' dentro de lista[2:3]? → Si → borramos posición 1
  - Lista = → ['a', 'b']
  - I = 0
- 8º Iteración:
  - I = 1→ lista[1] = 'b'
  - lista[i+1:] = lista[2:2] →[]

    - ¿'b' dentro de lista[2:2]? → No → Continuamos
    - Lista = → ['a', 'b']
    - I = 1
- 9º Iteración:
    - I = 2→ lista[2] = ¡¡ERROR!!!

¿Qué ha pasado? Que a medida que hemos ido borrando duplicados, hemos ido acortando la lista y como el bucle tenía establecido que debía de parar en i = 7, cuando intenta acceder a un valor que ya no existe (por acortarse la lista) nos devuelve error.

Hagamos el mismo ejemplo con el bucle inverso:

```
8  for i in range (len(lista)-1, -1, -1):
9      if lista[i] in lista[:i]:
10         lista.pop(i)
```

Pasaría lo siguiente:

- 1º Iteración:
    - i = len(lista) - 1 → i = 7 → lista[7] = 'b'
    - lista[:i] = lista[0:7] →['a', 'b', 'a', 'a', 'b', 'a', 'b']
    - ¿'b' dentro de lista[0:7]? → Si → borramos posición 7
    - Lista = →['a', 'b', 'a', 'a', 'b', 'a', 'b']
    - i = 7
- 2º Iteración:
    - i = 7 - 1 → i = 6 → lista[6] = 'b'
    - lista[:i] = lista[0:6] →['a', 'b', 'a', 'a', 'b', 'a']
    - ¿'b' dentro de lista[0:6]? → Si → borramos posición 6
    - Lista = →['a', 'b', 'a', 'a', 'b', 'a']
    - i = 6
- 3º Iteración:
    - i = 6 - 1 → i = 5 → lista[5] = 'a'
    - lista[:i] = lista[0:5] →['a', 'b', 'a', 'a', 'b']
    - ¿'a' dentro de lista[0:5]? → Si → borramos posición 5
    - Lista = →['a', 'b', 'a', 'a', 'b']
    - i = 5
- 4º Iteración:
    - i = 5 - 1 → i = 4 → lista[4] = 'b'
    - lista[:i] = lista[0:4] →['a', 'b', 'a', 'a']
    - ¿'b' dentro de lista[0:4]? → Si → borramos posición 4
    - Lista = →['a', 'b', 'a', 'a']
    - i = 4
- 5º Iteración:
    - i = 4 - 1 → i = 3 → lista[3] = 'a'
    - lista[:i] = lista[0:3] →['a', 'b', 'a']
    - ¿'a' dentro de lista[0:3]? → Si → borramos posición 3
    - Lista = →['a', 'b', 'a']
    - i = 3
- 6º Iteración:
    - i = 3 - 1 → i = 2 → lista[2] = 'a'
    - lista[:i] = lista[0:2] →['a', 'b']
    - ¿'a' dentro de lista[0:2]? → Si → borramos posición 2

    - Lista = →[‘a’, ‘b’]
    - i = 2
- 6º Iteración:
    - i = 2 - 1 → i = 1 → lista[1] = ‘b’
    - lista[:i] = lista[0:1] →[‘a’]
    - ¿‘b’ dentro de lista[0:1]? → No → Continuamos
    - Lista = →[‘a’, ‘b’]
    - i = 1
- 7º Iteración:
    - i = 1 - 1 → i = 0 → lista[0] = ‘a’
    - lista[:i] = lista[0:0] →[]
    - ¿‘a’ dentro de lista[0:0]? → No → Continuamos
    - Lista = →[‘a’, ‘b’]
    - i = 0

Y finalizaría el bucle. Al hacerlo de atrás adelante, las variaciones de longitud no afectan al bucle y puede terminar correctamente sin saltarse ningún duplicado.

4.2 Ejercicios de diccionarios

4.2.1 Ejercicio 6

Programa que lea un diccionario de la longitud que deseemos en la que almacenará datos de personas. La clave será el DNI y cada uno tendrá asignado una lista de datos de longitud variable. Finalmente se imprimirá el diccionario por pantalla.

```
1  longitud = int(input("¿Que longitud tendrá el diccionario? "))
2
3  diccionario = {}
4  for i in range (0, longitud):
5      clave = input('Introduce la clave del nuevo elemento: ')
6      long_lista = int(input('¿Cuantos datos va a tener el elemento? '))
7      lista_datos = []
8      for j in range (0, long_lista):
9          lista_datos.append(input('Introduce un nuevo valor: '))
10     diccionario[clave] = lista_datos
11
12 print(diccionario)
```

```
¿Que longitud tendrá el diccionario? 3
Introduce la clave del nuevo elemento: 35232323Z
¿Cuantos datos va a tener el elemento? 4
Introduce un nuevo valor: Alba
Introduce un nuevo valor: García López
Introduce un nuevo valor: Madrid
Introduce un nuevo valor: 915555555
Introduce la clave del nuevo elemento: 21474747J
¿Cuantos datos va a tener el elemento? 3
Introduce un nuevo valor: Fernando
Introduce un nuevo valor: Martínez Ruiz
Introduce un nuevo valor: Getafe
Introduce la clave del nuevo elemento: 88889999T
¿Cuantos datos va a tener el elemento? 2
Introduce un nuevo valor: Luis
Introduce un nuevo valor: 9122222222
{'35232323Z': ['Alba', 'García López', 'Madrid', '915555555'], '21474747J':
['Fernando', 'Martínez Ruiz', 'Getafe'], '88889999T': ['Luis', '9122222222']}

Process finished with exit code 0
```

Es semejante a los ejemplos de listas:

- La línea 1 sirve para que el usuario establezca el número de elementos que va a tener el diccionario.
- La línea 3 crea un diccionario vacío.
- En la línea 4 se define el bucle para dar tantas entradas al diccionario como se haya establecido previamente.
- La línea 5 recoge el valor que se usará como clave del diccionario para la entrada actual.
- La línea 6 recoge el número de elementos que va a tener esa entrada de diccionario.

- La línea 7 crea una lista vacía.
- Las líneas 8 y 9 crean un bucle y recogen los elementos de la entrada del diccionario.
- La línea 10 crea una nueva entrada en el diccionario con la clave y los valores proporcionados.
- Finalmente, la línea 12 imprime el diccionario por pantalla.

5 Bibliografía

Bahit, E. (s.f.). *Python para principiantes*. Obtenido de https://librosweb.es/libro/python/

Foundation, P. S. (2013). *Python.org*. Obtenido de https://www.python.org/

Suárez Lamadrid, A., & Suárez Jiménez, A. (Marzo de 2017). *Python para impacientes*. Obtenido de http://python-para-impacientes.blogspot.com.es/2017/03/el-modulo-time.html