Programación 1

Tecnicatura Universitaria en Inteligencia Artificial

2022

Apunte 5

1. Estructuras de datos

Hasta el momento hemos visto tipos de datos simples. Cada variable que utilizamos tenia un nombre y un tipo y almacenaba solo un valor. Sin embargo en ocasiones es necesario almacenar muchos datos relacionados entre si, con lo cual resulta conveniente darles cierta organización.

Por ejemplo considere el siguiente problema:

Problema

Dada la nota del primer parcial de los 250 alumnos de Programación I, mostrar el promedio y cuantos alumnos superaron el mismo

Con los tipos de datos vistos hasta el momento una estrategia posible sería cargar la nota de cada alumno en una variable e ir sumándola en una variable auxiliar, luego calcular el promedio y volver a cargar todas las notas para contar cuantos superaron el promedio. A continuación mostramos una solución siguiendo esa linea.

```
acum=0.0
cont=0
cant=0

nota=float(input("Ingrese primer nota (-1 para terminar) "))

while (nota !=-1 and cont<250):
  acum=nota+acum
  cont=cont+1
  if(cont<250):
    nota=float(input("Ingrese nota (-1 para terminar) "))

if (cont>0):
  prom=(acum/cont)

for i in range(cont):
  nota=float(input("Ingrese nota nuevamente "))
```

```
  if (nota>prom):
    cant=cant+1

print("Promedio: ",prom, " superaron la nota",cant, " estudiantes")
```

Esta solución tiene el inconveniente de que el usuario tiene que ingresar dos veces los mismos datos.

Otra estrategia podría ser cargar las 250 variables en sendas variables, calcular el promedio y luego recorrer las variables para contar las que superan el promedio. Lo cual resultaría en un código muy poco óptimo.

Ambas estrategias tienen sus dificultades. Para este tipo de casos se recurren a las estructuras de datos que nos permiten almacenar información relacionada bajo un mismo nombre.

Antes de ver las mismas pensemos cómo resolveríamos este problema si lo hiciéremos en papel (método que muchas veces nos ayuda a razonar los algoritmos).

En papel seguramente armaríamos una tabla (llamémosle **NOTAS**) en la cual en cada fila de la misma anotaríamos una nota.

	NOTAS
1	10
2	8
3	4
...	...
250	7

Nuestra tabla tiene un nombre (**NOTAS**), cada fila tiene un numero entero que identifica la nota de un alumno en particular a modo de **índice**, y tiene un **valor** de nota. Estos 3 elementos (Nombre, índice y valor) los encontraremos en las estructuras de datos que veremos.

En papel sumaríamos las notas de la tabla, dividiríamos la suma por la cantidad de notas y luego recorremos nuevamente la tabla para contar los que superan el promedio.

Es así como llegamos a las estructuras de datos (que veremos a continuación) y que definiremos como:

Estructuras de datos

Una **estructura de datos** es un conjunto de datos agrupados bajo un mismo **nombre**, que se caracteriza por tener una **organización determinada** y ciertas **operaciones** que se pueden realizar con ellos o entre sus elementos.

2. Listas

Las listas son estructuras de datos que nos servirán para modelar datos compuestos cuya cantidad y valor pueden variar durante la ejecución del programa. Es una agrupación de elementos separados por coma y entre corchetes. Tienen la particularidad de que pueden almacenar datos de distintos tipos. Y al poder agregar y quitar elementos o modificar el valor de los mismos se dice que son **mutables**.

2.1. Declaración

Declaramos una lista asignándole una serie de valores separados por coma encerrados entre corchetes.

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

También es posible declarar una lista vacía.

```
lista = []
lista
[]
```

La lista puede contener valores de distinto tipo. Las listas no son uniformes en cuanto a tipos de datos.

```
alumno = [7, "Jose", 24, 2022]
alumno
[7, 'Jose', 24, 2022]
```

Incluso un lista puede contener otra lista como elemento.

```
notas = [7, "Jose", [8,9,7]]
notas
[7, 'Jose', [8, 9, 7]]
```

También podemos usar la función list para generar una lista a partir de un iterable.

```
lista = list("1234")
lista
['1', '2', '3', '4']

lista = list(range(1,5))
lista
[1, 2, 3, 4]
```

Nótese que las dos listas generadas son distintas, ya que la primera es una lista cuyos elementos son caracteres en tanto los elementos de la segunda son enteros.

2.2. Acceso

Podemos acceder a los elementos de una lista a través de su índice mediante el operador []. Valen aquí muchas de las opciones que vimos para el acceso de strings.

Recordar que el indizado comienza en 0 y podemos usar un indizado negativo para recorrer la lista hacia atrás.

```
alumno = [7, "Jose", 24, 2022]
alumno[1]
'Jose'

alumno[-1]
2022
```

También podemos usar slicing para recuperar una porción de la lista (sublista, en este caso lo que obtenemos es una lista y no un elemento):

```
alumno[-2:]
[24, 2022]

alumno[1:3]
['Jose', 24]
```

El poder obtener un elemento a través de su índice también nos permite modificarlo.

```
alumno = [7, "Jose", 24, 2022]
alumno[0] = 1999
alumno
[1999, "Jose", 24, 2022]
```

También es posible modificar sublistas de una lista con slicing.

```
alumno = [7, "Jose", 24, 2022]
alumno[0:3] = [0, "Manuel", False]
alumno
[0, "Manuel", False, 2022]
```

2.3. Metodos

Veamos cómo operar con listas.

Vimos que es posible declarar una lista vacía. Una lista sin elementos no es muy interesante. Podemos agregarles elementos con el método `append(<elem>)`. Los métodos de una estructura de datos se invocan con un punto `.` luego del nombre de una variable de dicha estructura de datos (en este caso listas).

```
lista = []
lista
[]
lista.append(1)
lista.append(2)
lista.append("ABC")
lista.append([1,2,3])
lista
[1, 2, "ABC", [1,2,3]]
```

Notar que el método `append` añade elementos de a uno. En el último caso, tal vez, era mas deseable agregar todos los elementos de la lista pasada como argumento. Esto es posible con el método `extend(<iterable>)`.

```
lista = []
lista.extend([1,2,3])
lista
[1, 2, 3]
```

Asimismo, tal vez sea preferible no agregar elementos al final de la lista sino en una posición determinada. El método `insert(<index>, <elem>)` nos facilita esto.

```
lista = [1, 2, 3]
lista.insert(2, 15)
lista
[1,2,15,3]

lista.insert(0, ["a"])
lista
[["a"], 1, 2, 15, 3]
```

También es posible concatenar listas con el operador `+`. Agrega todos los elementos de la segunda lista (a la derecha del `+`) al final de la primera.

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
lista = lista1 + lista2
```

```
lista
[1, 2, 3, 4, 5, 6]

lista = lista2 + lista1
lista
[4, 5, 6, 1, 2, 3]
```

Hasta ahora vimos distintas formas de añadir datos a una lista. ¿Es posible eliminar elementos? Por supuesto que si. Primero veamos `remove(<elem>)` análogo a `append` pero quitando el elemento que se pasa como argumento. Notar que si el elemento que pasamos como parámetro para eliminar no se encuentra en la lista s el método `remove` devuelve una excepción (error).

```
lista = list(range(5))
lista.remove(3)
lista
[0, 1, 2, 4]

lista.remove(-1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    lista.remove(-1)
ValueError: list.remove(x): x not in list
```

Luego esta `pop(<index>)` analogo a `insert` que elimina el elemento que se encuentra en el índice que pasamos como argumento. Es posible utilizar este método sin argumento, en cuyo caso eliminará por defecto el ultimo elemento de la lista.

```
lista = list(range(5,10))
lista.pop(3)
lista
[5, 6, 7, 9]

lista.pop()
lista
[5, 6 ,7]
```

Para obtener el largo de una lista utilizamos la función `len(<iterable>)`.

```
lista = ['a', 'b', 1, 2, 3]
len(lista)
5
```

Es posible que un problema demande encontrar el índice en el que se encuentra un elemento en una lista. Esto se puede resolver con el método `index(<elem>)`

```
lista = ['a', 'b', 1, 2, 3, 2]
lista.index('b')
1

lista.index(2)
3
```

Es importante observar que el método `index` devuelve el índice de la primer aparición del elemento que estamos buscando. Tener cuidado que en caso que el elemento no se encuentra en la lista el método retornará una excepción (error).

```
lista = ['a', 'b', 1, 2, 3, 2]
lista.index(10)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    lista.index(10)
ValueError: 10 is not in list
```

En caso que simplemente queramos determinar si un elemento se encuentra dentro de una lista, Python nos ofrece el operador `in` que devuelve `True` en caso afirmativo y `False` en caso contrario.

```
lista = ['a', 'b', 1, 2, 3]
2 in lista
True

'c' in lista
False
```

Luego de agregar muchos elementos a una lista de manera aleatoria, para organizar mejor los datos es necesario ordenar la lista. El método `sort()` ordena por el operador `'<'`, es decir, de menor a mayor. Tener cuidado que es necesario que los elementos de la lista tengan el mismo tipo para que puedan ser comparados, en caso contrario devolverá una excepción (error).

```
lista = [6, 9, 0 , 3, -10, 33]
lista.sort()
lista
[-10, 0, 3, 6, 9, 33]

lista = ['a', 'b', 1, 2, 3]
lista.sort()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    lista.sort()
TypeError: '<' not supported between instances of 'int' and 'str'
```

2.4. Iterar sobre listas

Existen dos formas de recorrer una lista. Iterar sobre sus elementos o iterar sobre sus índices. Para el primer caso:

```
lista = ['a', 'b', 'c', 'd', 'e']
for elemento in lista:
    print(elemento)
'a'
'b'
'c'
'd'
'e'
```

Para iterar sobre los índices realizamos lo siguiente:

```
lista = ['a', 'b', 'c', 'd', 'e']
for i in range(len(lista)):
    print(lista[i])
'a'
'b'
'c'
```

```
'd'
'e'
```

A menos que el problema que estamos intentando resolver lo requiera, siempre es preferible utilizar la primera forma de iteración ya que provee una clara sintaxis de lo que ocurre en el bucle y dota al código de intención.

Python también provee una forma de poder iterar sobre ambos gracias a la función `enumerate`.

```
lista = ['a', 'b', 'c', 'd', 'e']
for index, value in enumerate(lista):
    print(index, value)
0 'a'
1 'b'
2 'c'
3 'd'
4 'e'
```

En la sección de Tuplas veremos mas en detalle este comportamiento.

3. Tuplas

Las tuplas son otra estructura de datos que nos permite agrupar datos relacionados en una misma variable. Poseen algunas similitudes con las listas, por ejemplo, pueden almacenar datos de distintos tipos y los datos se encuentran ordenados por índice. Sin embargo, las tuplas tienen un tamaño fijo una vez declaradas, es decir, no es posible agregar nuevos elementos ni eliminar elementos. Más aún, no es posible modificar los elementos que la componen, es decir, son inmutables.

3.1. Declaración

Una tupla se declara entre parentesis y seperando sus elementos con comas.

```
tupla = (1,2,3,4)
tupla
(1,2,3,4)

tupla = ("abc", True, (1,2))
tupla
("abc", True, (1,2))
```

Es posible declarar una tupla vacía aunque no tiene ninguna utilidad como tambien crear una tupla de un único elemento.

```
tupla = ()
tupla
()

tupla = (10,)
tupla
(10,)
```

Al igual que las listas, existe la función `tuple` para convertir un iterable a una tupla (análoga a `list`)

```
tupla = tuple("1234")
tupla
('1','2','3','4')
```

```
tupla = tuple(range(1,5))
tupla
(1,2,3,4)
```

3.2. Acceso

Para obtener un elemento de una tupla, accedemos de la misma forma que con listas, a través de índices.

```
tupla = ("abc", True, (1,2))
tupla[0]
"abc"
tupla[-1]
(1, 2)
```

También podemos utilizar slicing para obtener subtuplas de una tupla.

```
tupla = ("abc", True, (1,2), 3, 'd')
tupla[1:3]
(True, (1, 2))
tupla[-1:2]
('d', "abc", True)
```

3.3. Métodos

Igual que en listas, es posible determinar en que índice se encuentra un elemento que busquemos con el método `index(<elem>)`.

```
tupla = ("abc", True, (1,2), 3, 'd')
tupla.index(True)
1

tupla.index((1, 2))
2

tupla.index('a')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    tupla.index('a')
ValueError: tuple.index(x): x not in tuple
```

Tambien existe el método `count(<elem>)` que calcula la cantidad de apariciones en la tupla del elemento que pasemos como argumento.

```
tupla = (1,2,1,1,6,'a','a')
tupla.count(1)
3
tupla.count('a')
2
tupla.count([])
0
```

3.4. Iterar sobre tuplas

Para recorrer una tupla lo hacemos de la misma forma que en listas, sobre sus elementos o sobre sus índices o bien ambos con la función `enumerate`.

```
tupla = ('a', 'b', 'c', 'd', 'e')

for elemento in tupla:
    print(tupla)
'a'
'b'
'c'
'd'
'e'

for i in range(len(tupla)):
    print(tupla[i])
'a'
'b'
'c'
'd'
'e'

for index, value in enumerate(tupla):
    print(index, value)
0 'a'
1 'b'
2 'c'
3 'd'
4 'e'
```

3.5. Desempaquetado

Desempaquetar una tupla es separar sus componentes en variables individuales, Se realiza asignando a las variables la tupla. Hay que tener en cuenta que la cantidad de variables debe coincidir con la cantidad de elementos en la tupla.

```
tupla = ( 'a ' , 'b' , 'c ')
x,y,z = tupla
x,y,z
'a ' 'b' 'c '
```

3.6. Uso

Con lo visto hasta ahora las tuplas parecen una versión limitada, menos flexible que las listas. Entonces, ¿Cuándo es conveniente usar tuplas?.

Su utilidad esta en empaquetar datos y asegurar que no serán modificados durante la ejecución del programa. Sirven para representar objetos del mundo real que no pueden ser descriptos con único tipo de datos. Veamos esto con un ejemplo.

Alguien podría estar interesado en registrar distintos datos de los alumnos de una comisión como pueden ser el nombre, el apellido, el teléfono y el correo electrónico. Esta claro que no es posible representar toda esta información simplemente con un `int` o `string`. Entonces el programador puede proponer empaquetar todos estos datos en una tupla que representarán a un alumno.

```
# alumno = (nombre, apellido, telefono, email)
```

Esto quiere decir que en este programa las tuplas compuestas por 4 elementos serán interpretadas como alumnos donde la primer componente es el nombre del alumno, la segunda su apellido, el tercero su número de teléfono y cuarto su correo electrónico.

¿Por qué no utilizar una lista? Porque las listas son modificables, agregando o quitando elementos. Una tupla asegura que la representación de un alumno no cambiará.

Luego, como se esta registrando una comisión de alumnos, si conviene utilizar una lista ya que a priori no se sabe cuantos alumnos son. Con lo cual, se obtendría una lista de este estilo.

```
alumnos = [("Martín", "Fernandez", "341556690", "mf10@gmail.com"),\
("Juan Manuel", "Lopez", "113451098", "jml1999@gmail.com"), ...]
```

Una vez armada nuestra lista de alumnos, ya es posible empezar a calcular información adicional de la comisión como ¿Cuántos alumnos se llaman igual? ¿Cuántos alumnos tienen teléfono pertenecientes a Rosario? ¿Cuántos mails no son @gmail.com?. Para poder responder estar preguntas es necesario recorrer la lista con una iteración del estilo:

```
for alumno in alumnos:
    ...
```

Sin embargo, rápidamente el programador se puede dar cuenta que es algo engorroso tener que acceder a los distintos elementos de la tupla para obtener la información deseada. Por ejemplo, con la pregunta de los teléfonos quedaría algo así:

```
telefonosRosario = 0
for alumno in alumnos:
    if (alumno[2][0:3] == "341"):
        telefonosRosario += 1
```

El código no resulta claro a la vista si no poseemos el contexto de la información representada. Acá entra en juego otra característica importante de las tuplas. Tal como es posible empaquetar información, también es posible desempaquetarla y mejorar la legibilidad del código. Veamos

```
telefonosRosario = 0
for nombre, apellido, telefono, mail in alumnos:
    if (telefono[0:3] == "341"):
        telefonosRosario += 1
```

Notar como ahora es mucho mas claro que estamos comparando el número de teléfono de un alumno.

También observar que esto es lo que ocurre cuando utilizamos la función `enumerate`. Esta función crea una lista de tuplas de 2 elementos donde el primer elemento es el índice y el segundo el valor del iterable que pasamos como argumento en dicho índice. Luego al iterar, desempaquetamos esta información y la manipulamos separadamente.

En resumen, con tuplas podemos empaquetar información relacionada, usualmente heterogénea, para representar objetos de la vida diaria. Ofrecen una sintaxis y operaciones sencillas, además de garantizar la inmutabilidad de los datos almacenados.

Cabe preguntarse en este ejemplo, ¿Cualquier tupla de 4 elementos es un alumno? ¿Es posible asegurar eso? La respuesta es no. Las tuplas son muy sencillas en su construcción para empaquetar información pero no constituyen un nuevo Tipo único e identificable. Más adelante veremos mecanismos más avanzados para lograr esta característica.

4. Diccionarios

Retomemos el ejemplo de la sección anterior. Imaginemos que quisiera encontrar el número de teléfono y correo de un alumno llamado Matías Gaboto. Por el momento, tenemos almacenada una lista de tuplas que representan los alumnos de la comisión. ¿Cómo podría encontrar la información del alumno Matías? Las listas son muy flexibles en su inserción, eliminación y/o modificación de datos pero para buscar datos dentro de ellas no queda otra que recorrerla secuencialmente hasta encontrar la información deseada. Esto tal vez no sea computacionalmente costoso con un lista de 100 alumnos. Sin embargo, a medida que aumenta el tamaño de lista, la operación de buscar se vuelve cada vez mas lenta. Piense que sucedería si requerimos buscar un alumno en una lista de más de 100000 alumno.

Es claro que las listas no pueden ayudarnos mucho. ¿Hay alguna estructura de datos que nos permita buscar/obtener información más rápido? Hablemos de Diccionarios en Python.

Los Diccionarios son otra estructura de datos como las Listas o Tuplas que ofrece Python. Su particularidad radica en que en vez guardar simplemente datos (`int`, `strings`, etc) almacena pares de valores que se conocen como `clave:valor` (`key:value pairs` en inglés). Decimos que cada clave dentro de un diccionario tiene asociado un valor. Para visualizarlo mejor, piense en un diccionario de la vida diaria donde cada palabra del idioma Español por ejemplo tiene una definición asociada. Esto también evidencia su propósito, los diccionarios sirven para realizar consultas de información de manera eficiente a través de sus claves. Sin embargo, los diccionarios de Python no se encuentran ordenados. Se dice que son una colección de datos no-ordenada. Es decir, no existe el primer elemento de un diccionario (o como hacíamos en listas y tuplas accediendo al índice 0).

4.1. Declaración

Un diccionario en Python se declara entre llaves y separando los pares clave-valor entre comas.

```
diccVacio = {}
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
```

Por ejemplo, definimos un diccionario vacío y otro donde asociamos los campos Nombre, Telefono y Edad, y les asignamos valores correspondientes.

Similar a las funciones `list` o `tuple`, existe la función `dict` pero en vez de recibir como argumento un iterable cualquiera, recibe un iterable de pares clave-valor

```
dicc = dict([('jorge', 4139), ('guido', 4127), ('hugo', 4098)])
dicc
{'jorge': 4139, 'guido': 4127, 'hugo': 4098}

dicc = dict([['jorge', 4139], ['guido', 4127], ['hugo', 4098]])
dicc
{'jorge': 4139, 'guido': 4127, 'hugo': 4098}

dict([3, 6, 8, 9])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

4.2. Acceso

Como se menciono antes, en un diccionario se acceden a los valores a través de las claves. Formalmente, se pone la clave entre corchetes

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
```

```
dicc["Nombre"]
"Matías"
dicc["Edad"]
23
```

Utilizar la clave entre corchetes también sirve para modificar el valor asociado a dicha clave.

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc["Nombre"] = "Lautaro"
dicc
{"Nombre": "Lautaro", "Telefono": "3411038003", "Edad": 23}
```

También es posible agregar un nuevo par clave-valor.

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc["Legajo"] = 4763927
dicc
{"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23,
"Legajo": 4763927}
```

Si intentamos acceder con una clave que no existe dentro del diccionario resultará en una excepción (error)

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc["Correo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Correo'
```

En este punto resulta necesario aclarar, ¿Qué valores pueden ser clave en un diccionario? Las claves deben ser únicas y no pueden ser modificadas. Formalmente, cualquier tipo de dato **inmutable** puede ser clave en un diccionario, esto incluye `int`, `string`, `float` y `bool`. Las listas no pueden ser clave debido a que pueden ser modificadas. Si lo intentáramos, ocurrirá lo siguiente:

```
d = {}
d[[1,2]] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Opcional: investigar qué es un `hash` o función hash.

Las tuplas pueden ser clave en un diccionario siempre y cuando no incluyan datos mutables.

```
d = {}
d[(1,2)] = 2
d
{(1,2): 2}
d[(0, [0])] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

4.3. Métodos

Como vimos, podemos obtener un valor si poseemos una clave valida utilizando `[]` pero en caso que la clave no exista el programa fallará. Para poder hacer una consulta más segura existe el método `get(<key>)` que en caso de no existir dicha key retornará `None`

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc.get("Nombre")
"Matías"
dicc.get("Correo")
None
```

Si quisieramos obtener solo las claves de un diccionario, es posible utilizando `list`

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
list(dicc)
["Nombre", "Telefono", "Edad"]
```

O bien podemos utilizar el método `keys`.

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc.keys()
dict_keys(['Nombre', 'Telefono', 'Edad'])
```

También podemos obtener los valores de un diccionario con `values`.

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc.values()
dict_values(['Matías', '3411038003', 23])
```

Incluso obtener todos los pares clave-valor con `items`.

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc.items()
dict_items([('Nombre', 'Matías'), ('Telefono', '3411038003'),
('Edad', 23)])
```

Es facil ver que ninguno de estos métodos retorna una lista como tal vez uno intuiría. Por el momento no se preocupe por este tipo de dato, sepa que pueden ser iterados y preguntar si un elemento pertenece a ellos.

Para preguntar si un diccionario posee una clave podemos utilizar el método `has_key()`.

```
d = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
d.has_key("Nombre")
True
d.has_key("Legajo")
False
```

O bien utilizar el operador `in` como en listas o tuplas.

```
d = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
"Nombre" in d
True
"Legajo" in d
False
```

El método `pop(<key>)` elimina el par clave-valor del diccionario. En caso de no existir la clave pasada como argumento, el programa fallará.

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc.pop("Telefono")
{"Nombre": "Matías", "Edad": 23}
```

```
dicc.pop("Correo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Correo'
```

Otro método para eliminar es `popitem` que quitará el último par clave-valor agregado al diccionario.

```
dicc = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
dicc["Legajo"] = 12345
dicc
{"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23,
"Legajo": 12345}
dicc.popitem()
("Legajo", 12345)
```

Puede ser que sea necesario actualizar muchos pares clave-valor a la vez y el operador `[]` solo puede hacerlo de a uno. Para esto está el método `update(<dict>)` que toma como argumento otro diccionario. Este método modifica el primer diccionario del siguiente modo: si una clave se encuentra en ambos diccionarios, entonces toma el valor del diccionario pasado como argumento y si no se encuentra en el primer diccionario es agregada.

```
d1 = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23}
d2 = {"Nombre": "Francisco", "Edad": 20, "Apellido": "Suarez"}
d1.update(d2)
d1
{"Nombre": "Francisco", "Telefono": "3411038003",
"Edad": 20, "Apellido": "Suarez"}
```

4.4. Iterar sobre diccionarios

Se puede iterar sobre diccionarios de 3 maneras distintas. La opcion por defecto, y la mas usual, es iterar sobre las claves del diccionario.

```
d = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23,
"Apellido": "Suarez"}
for clave in d:
    print(clave)

'Nombre'
'Telefono'
'Edad'
'Apellido'
```

Esto es equivalente a

```
d = ...
for clave in d.keys():
    print(clave)
```

Para iterar sobre los valores, podemos utilizar las clave para acceder a sus respectivos valores asociados.

```
d = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23,
"Apellido": "Suarez"}
for clave in d:
    print(d[clave])
```

```
'Matias'
'3411038003'
23
'Suarez'
```

O utilizando el método `values`

```
d = ...
for value in d.values():
    print(value)
```

Para iterar sobre los pares clave-valor

```
d = {"Nombre": "Matías", "Telefono": "3411038003", "Edad": 23,
"Apellido": "Suarez"}
for clave in d:
    print(clave, d[clave])

'Nombre' 'Matias'
'Telefono' '3411038003'
'Edad' 23
'Apellido' 'Suarez'
```

O utilizando el método `items`

```
d = ...
for key, value in d.items():
    print(key, value)
```

5. Bibliografía

- Apunte de la materia Algoritmos y Programación 1, primera materia de programación de las carreras de Ingeniería en Informática y Licenciatura en Análisis de Sistemas dela Facultad de Ingeniería de la UBA: http://materias.fi.uba.ar/7501/apunte%20PYTHON.pdf
- "Fundamentos de programación: Algoritmos, estructura de datos y objetos."(4ta Edición). Luis Joyanes Aguilar
- "Python for Everybody: Exploring Data Using Python 3". Charles R. Severance
- Python Data Structures Documentation: https://docs.python.org/3/tutorial/datastructures.html#
- El Libro de Python. Libro digital: https://ellibrodepython.com/