4. Estructuras de Datos en Python

Estructuras de Datos

En este tema vamos a repasar las principales estructuras de datos que ya incorpora el lenguaje python.
En concreto, vamos a ver cinco clases que tienen la característica común de ser iterables.

- Cadenas. ' '
- Listas. [ ]
- Tuplas. ( )
- Diccionarios. { : }
- Conjuntos. { }

Cadenas :

- Una cadena es una secuencia de caracteres
- Se pueden acceder de uno en uno con el operador corchete **[]**

```
>>> fruta = 'naranjas'
>>> letra = fruta[1]
```

La expresión entre corchetes se denomina índice y debe ser un entero. Pero ….¿qué devuelven las siguientes sentencias?

```
>>> print(letra)

>>> fruta[-1]
```

Longitud de las cadenas. ***len()***

```
>>> fruta = 'naranja'
>>> len(fruta)
7

>>> ultima = fruta[len(fruta)-1]
>>> print(ultima)
a
```

Con la función **str()** podemos crear y convertir a cadenas

```
>>> vacia=str()
>>> len(vacia)
0

>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Una cadena vacía no contiene caracteres y tiene una longitud de 0

Recorrido de una cadena con un bucle.

```
fruta = 'naranja'

indice = 0
while indice < len(fruta):
      letra = fruta[indice]
      print (letra)
      indice = indice +1
```

Esta segunda forma es más simple , elegante y "pythonica":

```
for car in fruta:
      print (car)
```

Ej.4.1- Escribe un bucle **while** que comience en el último carácter de la cadena y haga su recorrido hacia atrás hasta el primero, mostrando cada letra en la misma línea.

Rebanado o slicing:

```
>>> s = 'Monty Python'
>>> print ( s[0:5] )
Monty
>>> print ( s[6:12])
Python
```

El formato es ***cadena[inicio : final]***
Inicio es 0, para la primera posición de la cadena
La última posición devuelta es **final – 1**

Python utiliza el rebanado en otros tipos de datos como listas o tuplas

Rebanado o slicing

```
>>> fruta = ’banana’
>>> fruta[:3]
’ban’
>>> fruta[3:]
’ana’
```

El formato ***cadena[: final]*** indica desde la posición 0, de la cadena, hasta ***final***
[inicio :] indica desde **inicio** hasta que la cadena se acabe.

Cadenas inmutables

```
>>> saludo = ’Hola mundo’
>>> saludo[0] = ’J’
TypeError: 'str' object does not support item assignment
```

No podemos modificar un carácter de una cadena.
Para hacer eso necesitamos crear una cadena nueva:

```
>>> nuevo_saludo = ‘J’ + saludo[1:]
```

Operador *in* en la cadenas

```
>>> ‘a’ in ‘Hola mundo’
True
>>> ‘cruel’ in ‘Hola mundo’
False
```

Devuelve True si el primer operando es una subcadena del segundo

Comparación de cadenas

```
if palabra == ‘botella’:
if palabra < ‘botella’:
```

Utiliza los operadores de comparación de python:
== , != , < , >, <= , >= , is , is not
Devuelven True o False siguiendo un orden alfabético

```
>>> 'ab' < 'aaaaaaa'
False
>>> 'ab' < 'Aa'
False
>>> 'Ab' < 'aa'
True
>>>
```

```
>>> ord('A')
65
>>> ord('a')
97
>>> chr(48)
'0'
>>> chr(49)
'1‘
```

La función **ord()** devuelve el ordinal entero del carácter indicado y justo lo contrario hace la función **chr()** que devuelve el carácter (Unicode) que representa al número indicado.

Métodos(1) de cadenas:

```
>>>palabra = 'botella'
>>>nueva_palabra =  palabra.upper()
>>> print (nueva_palabra)
```

La notación con el punto "**.**" especifica el nombre del método. Los paréntesis vacíos indican que el método no toma argumentos.
La llamada a un método se denomina invocación

---

(1) Un Método en POO, es un conjunto de instrucciones que realizan una determinada tarea, invocada por el objeto de la clase.

Métodos de cadenas:

```
>>>indice = 'botella'.find('lla')
>>> print (indice)
4
```

El método ***find*** puede llevar un segundo argumento para indicar la posición de comienzo de la búsqueda.
OJO ! Distingue entre mayúsculas y minúsculas.

```
>>> palabra = 'botella'
>>> palabra.find('l',4)
```

Métodos de cadenas:

Ej.4.2- ¿Qué devuelve **find** si no encuentra nada?

Ej.4.3- Supongamos que debemos localizar la dirección de procedencia de correos con el siguiente formato:

“From stephen.marqu@uct.ac.za Sat Jan 5 09:14:16 2017”

¿Cómo obtendría, la subcadena “**uct.ac.za**”, de la cadena anterior?

Ej.4.4- Extraer la probabilidad de spam, de la siguiente cadena, y convertirla a punto flotante.

"X-DSPAM-Confidence: 0.8475"

Métodos de cadenas:

```
>>>linea = '    todo esta listo    '
>>>linea.strip()
'todo esta listo'
```

Strip y startswith también son métodos útiles

```
>>> linea = 'Que tengas buen día'
>>> linea.startswith('Que')
True
>>> linea.startswith('q')
False
```

Si la cadena está vacía, **startswith** también devuelve **False.**

Ej.4.5-Depura el siguiente código para que no de error cuando se introduce una línea vacía

```
while True:
        linea =input('>')
        if linea[0] == '#':
                continue
        if linea == 'fin':
                break
        print (linea)
print ('Terminado')
```

Métodos útiles de cadenas:

cadena.isalpha()
Devuelve verdadero (True) si todos los caracteres en la cadena son alfabéticos. Los espacios en blanco no lo son.

cadena.isalnum()
Devuelve verdadero (True) si todos los caracteres en la cadena son alfanuméricos. Los espacios en blanco no lo son.

cadena.isdecimal(), cadena.isdigit(), cadena.isnumeric()
Devuelve verdadero (True) si todos los caracteres en la cadena son números.

cadena.isspace()
Devuelve verdadero (True) si todos los caracteres son espacios en blanco.

cadena.islower(), cadena.isupper()
Devuelve verdadero (True) si todos los caracteres son minúsculas o mayúsculas, respectivamente.

cadena.istitle()
Devuelve verdadero (True) si el primer carácter de la cadena es mayúsculas y el resto minúsculas; o en el caso de que haya palabras separadas por espacios en blanco que cumplan la misma regla.

Ayuda de python

La funcion **dir()** nos muestra los métodos disponibles para un objeto de una clase

```
>>>dir (str)
[.......
__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'capitalize', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'islower', 'isnumeric', 'isprintable',
....]
```

función **help()** muestra ayuda del método concreto

```
>>>help (str.strip)
```

Operador de formato de cadenas. %

```
>>> camellos = 42
>>> ‘he divisado %d camellos.’    %    camellos
‘he divisado 42 camellos’
>>> 'a = %d y b = %d  %s'    %    (3, 4, ’camellos’)
'a = 3 y b = 4  camellos'
>>>
```

- El resultado del operador es una cadena.
- Las secuencias de formato deben coincidir con los elementos de la tupla.
- Las secuencia ***%d*** formatea un enero, ***%g*** un número en punto flotante y ***%s*** una cadena

Listas. []

- Serie de elementos separados por comas, encerrados entre corchetes. []
- Los elementos pueden ser de distinto tipo
- Dinámicas: número variable de elementos
- Mutables

```
>>> v = [1, 2, 3]
>>> v
[1, 2, 3]
>>> v = [1, 2.0, ‘tres’]
>>> print (v)
[1, 2.0, ’tres’]
```

Listas. []

- Creación de una lista vacía

```
>>> v = []
>>> v
[]
>>> v = list()
```

- Las listas vacías se evalúan a False

```
>>> x = []
>>> bool(x)
False
```

Listas.[]

- Acceso a elementos

```
>>> lista = [1, 2.0 ,'tres', 4]
>>> print (lista[2])
tres
```

```
>>> print (lista[1:3])
[2.0, 'tres']
>>> print (lista[2:])
```

- Modificación de elementos

```
>>> v = [4, 7, 3, 8]
>>> v[-1] = 0
>>> v
[4, 7, 3, 0]
```

- Tamaño de un lista. **len()**

```
>>> len(v)
4
```

Listas. []

- Las listas son mutables

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> b = a
>>> b is a
True
```

Para que a apunte a una copia de la lista b

```
>>> a=b[:]
```

Listas. []

- Métodos de Listas. **append() , insert()**

```
>>> v = [1, 3, 4]
>>> v.append(5)
>>> v
[1, 2, 3, 5]
>>> v = [1, 3, 4]
>>> v.insert(2,9)
>>> v
[1, 3, 9, 4]
```

- append() Añade al final
- insert(index , object) Añade en la posición de index

- Operadores **+** y ***** en las listas (concatenación):

```
>>> v + [2,0,2]
[1, 3, 9, 4, 2, 0, 2]
>>>v
?
```

```
>>> v*3
[1, 3, 9, 4, 1, 3, 9, 4, 1, 3, 9, 4]
>>>v
?
```

Listas.[]

- Métodos de Listas. Borrado por valor. **remove()**

```
>>> v = [1, 3, 4]
>>> v.remove(3)
>>> v
[1, 4]
```

La mayoría de los métodos de lista no devuelven nada, modifican la lista y devuelven ***None***

- Borrado por posición. **del()** ó método **.pop()**

```
>>> v = [9, 5, 8, 0, 9]
>>> del (v[3])
>>> v
[9, 5, 8, 9]
>>> del (v)
?   ¿qué sucede  con v ?
```

```
>>> v = [9, 5, 8, 0, 9]
>>> v.pop(3)
0
>>> v
[9, 5, 8, 9]
```

- Índice conocido
- Sin índice borra el último de la lista

Listas.[]

Ojo con el slice (rebanado) para borrar elementos de las listas

```
>>> t=['a','b','c','d']
>>> c= t
>>> c is t
?   # analice el resultado
>>> t=t[1:]    # borrar el primero de t
>>> t is c
?   # analice el resultado
```

¿Qué está sucediendo al borrar un elemento de la lista como en el ejemplo ?

```
>>> t= ['b', 'k', 'a', 'b', 'k']
>>> t[0:2] ='‘
>>> t
```

Formas de borrar con slicing

```
>>> t= ['b', 'k', 'a', 'b', 'k']
>>> del(t[0:2])
>>> t
```

RECOMENDACIÓN: Elige un estilo y ajústate al él.

Listas.[]

Métodos de listas.

- Comprobar si un elemento está en la lista con operador **in**
- Buscar la posición de un valor en la lista. **.index()**

```
>>> v = [7, 0, 7]
>>> 7 in v
True
>>> v.index(7)
0
>>> v.index(7,1)
2

>>> v.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 4 is not in list
>>>
```

Busca a partir de la posición 1

¡OJO !, si no lo encuentra dará un error

Listas. []

- Concatenación en Listas **.append()** vs **.extend()**

```
>>> x = [1, 2, 3]
>>> x.append([9, 8])
>>> x
[1, 2, 3, [9, 8]]
>>> x=x.append(5)
>>> x
>>>?   #¿qué sucece con x?
```

Un único elemento

¡ OJO, error !

```
>>> x = [1, 2, 3]
>>> x.extend([9, 8])
>>> x
[1, 2, 3, 9, 8]
>>>
```

- Método de cadenas **.join()** **s.join(secuencia)**

Retorna una CADENA resultante de concatenar la cadena de "**secuencia**", separada por la cadena **(s)** sobre la que se llama al método

```
>>> list1 = ['a', 'b', 'c']
>>> '-'.join(list1)
'a-b-c'
>>>
```

```
>>> list1 = ['a', 'b', 'c']
>>> cad= '\n'.join(list1)
>>> print(cad)
```

Listas. []

- Método **.sort()**

```
>>> t=['d','c','e','b','a']
>>> t.sort()
>>> print (t)
['a', 'b', 'c', 'd', 'e']
```

- ¡ OJO! sort() devuelve None. Si hacemos : t = t.sort() es INCORRECTO
- sort(reverse = True) ordena la lista en orden descendente
- No ordena entre tipos de datos diferentes.

- **Función sorted()**

```
>>> t=['d','c','e','b','a']
>>> sorted(t)
['a', 'b', 'c', 'd', 'e']
```

- sorted() devuelve la lista ordenada en una copia nueva

- Funciones **max, min, sum** en listas.

```
>>> max([7, 2, 5])
7
>>> min(([7, 2, 5])
2
>>> sum([7, 2, 5])
14
```

- sum() No suma datos no numéricos

Listas. []

- Las función **list()**. Fabricando listas.

```
>>> s = 'spam'
>>> t = list(s)
>>> print (t)
['s', 'p', 'a', 'm']

>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]

>>> list(range(3,7))
[3, 4, 5, 6]
>>>
```

Listas. []

- Obtener una lista de una cadena con el método **split()** de cadenas.

s.split([sep, [, maxsplit]])

Retorna una LISTA a partir de la cadena "**s**", la cual se delimita por los índices que indique "**sep**". Si no se especifica "**sep**" se usan los espacios. Si se especifica "**maxsplit**", éste indica el número máximo de índices a realizar.

```
>>> lista = 'python, 3, curso, iniciación'.split(', ')
>>> print (lista)
['python', '3', 'curso', 'iniciación']
```

Ej.4.6 – Modifique el ejercicio 4.4 para usar **split()** en lugar de **find()**

Ej.4.7 – Quitar todos los espacios en blanco de la cadena :

s = "Programando en \nPython"

Ej.4.8 – Separar cada palabra por un espacio en la cadena:

s = " Programando en \nPython "

Listas por comprensión. []

- Si tratamos de obtener una cadena de una lista por el método ya visto anteriormente **.join()** o usar la función **str()** sucede esto:

```
lista = ['1',' 2', '3']
str1 = ''.join(lista)
print (str1)
```

OK

```
lista = [1, 2, 3]
str1 = ''.join(lista)
print (str1)
```

¡ ERROR !

```
>>> list1 = [1,2,3]
>>> cad= str(list1)
>>> cad
'[1, 2, 3]'
```

No está limpio →

```
>>> lista = cad[1:-1].split(', ')
>>> cad3 = ''.join(lista)
>>> cad3
'123'
```

¿Qué pasaría si en la lista list1 tenemos enteros y cadenas?

Listas por comprensión. []

- Las comprensiones de listas ofrecen una manera concisa de crear listas. Sus usos comunes son para hacer nuevas listas donde cada elemento es el resultado de algunas operaciones aplicadas a cada miembro de otra secuencia o iterable

```
>>> cuadrados = []
>>> for x in range(10):
...     cuadrados.append(x**2)
...
>>> cuadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> cuadrados = [ x ** 2 for x in range(10) ]
>>> cuadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Una lista de comprensión consiste de corchetes **[]** rodeando una expresión seguida de la declaración **for** y luego cero o más declaraciones for o if.

EJEMPLO: Resolviendo de un plumazo el problema anterior, para pasar una lista con valores numéricos a una cadena.

```
>>> list1 = [1, 2, 3, 'aa']
>>> str1 = ''.join( [ str(i) for i in list1 ] )
>>> print (str1)
```

Listas. []

Ej.4.9 – Recorre una lista de números y actualiza su contenido multiplicando cada elemento por 2.

Ej.4.10 – Crea un programa que almacene en una lista los números, introducidos por el usuario, hasta que se introduzca la palabra "fin". Al final el programa debe mostrar la media de esos números.

Ej.4.11- A partir de dos listas de enteros, 'numeros1' y 'numeros2', crea una lista que contiene aquellos valores de la primera que también están en la segunda e imprímela por pantalla. Es decir, calcula la intersección de ambas listas.

Listas. []

Iterando en una lista con **enumerate().**

En pocas palabras, es una función que permite iterar de manera simultánea sobre dos variables: el índice y el valor.

```
mi_lista = ['a', 'b', 'c', 'd']
# Caso 1:
indice = 0
for valor in mi_lista:
    print(indice , valor)
    indice += 1

# O también, caso 2:
for indice in range( len(mi_lista) ):
    print(indice, mi_lista[indice])
```

```
mi_lista = ['a', 'b', 'c', 'd']

for indice, valor in enumerate(mi_lista):
        print(indice,valor)
```

Lo que realmente hace la función internamente es generar 4 tuplas, cada una de las cuáles contiene los correspondientes pares índice-valor

Ej.4.12 – Modifica el ejercicio 4.9 para usar enumerate()

Ej.4.13 - Para cada una de las cadenas de texto almacenadas en una lista, imprimir por pantalla el índice y la cadena en si, indicando si la palabra es demasiado corta (cinco o menos caracteres) o larga (más de cinco caracteres)

TUPLAS. (,)

- Sintácticamente, son una serie de elementos separados por comas “,”
- Son inmutables. Los métodos como append o remove no exiten en las tuplas.
- Se representan con los paréntesis **( )**, en lugar de los corchetes **[ ]**.

```
>>> t= 2 , 3 , 4
>>> t
(2, 3, 4)

>>> t = (3 , 'a‘ , 2 , [4,5,6] , (3,4))
>>> type(t)
<type 'tuple'>

>>> type(3 , 'a‘ , 2 , [4,5,6] , (3,4))       # ERROR
```

TUPLAS (,)

- Tuplas vacías:

```
>>> t = ()
>>> type(t)
<class 'tuple‘>
```

```
>>> t = tuple()
>>> len(t)
0
```

```
>>> bool(t)
False
>>>
```

La función tuple() puede crear tupla partiendo de listas o cadenas

- ¿Cómo definimos una tupla de un único elemento?

```
>>> t = (1, )
```

- ¿Por qué es necesaria la coma?

TUPLAS. (,)

- Manejo de tuplas:

Podemos reemplazar una tupla con otra, lo mismo que en las cadenas o listas, usando el operador **+**

```
>>> x = (1, 8)
>>> x[0]
1
>>> len(x)
2
>>>  x[1:]
(8, )

>>>('A', ) + x[1:]
```

Uso de [ ] para acceso a los índices

Podemos usar Slicing

- Acceso inmutable vs mutable

```
>>> t = (2 , ['a','b'])
>>> t[1] = ‘a‘                  #  ¿ qué sucede ?

>>> t[1]
['a', 'b']
>>> t[1][1] = 0                 # ¿ y en este caso ?
```

TUPLAS (,)

- Funciones, max, min, sum

```
>>> t = (3, 4, 7, 1)
>>> max (t)
7
```

```
>>> min(t)
1
>>>
```

```
>>> sum(t)
15
>>>
```

- Método **.index()** y operador **in**

```
>>> t = (3, 4, 7, 1)
t.index(7)
2
```

```
>>> 2 in t
False
>>>
```

El método index devuelve error si no encuentra el valor

TUPLAS (,)

- Método **.count()**

```
>>> t = (3, 4, 7, 3, 1)
>>> t.count(3)
2
```

- Comparación de tuplas.

```
>>> (0,1,2) < (0,3,4)
True
>>> (0,1,200000) < (0,3,4)
True
```

La comparación usa orden lexicográfico: primero se comparan los dos primeros ítems, si son diferentes esto ya determina el resultado de la comparación; si son iguales, se comparan los siguientes dos ítems

Las tuplas no cuentan con el método **.sort()** , pero es posible, como ya vimos en las listas, utilizar la función **sorted()** y convertir la lista devuelta con la función **tuple()**

TUPLAS (,)

- Gracias a las tuplas podemos hace cosas como

```
>>> a,b = b,a

>>> t=(4,5,6)
>>> a,b,c = t
>>> a
4
>>> tu= a,b,c
>>> tu
(4, 5, 6)
```

Intercambio de variables

Desempaquetado de tupla

Empaquetado de tupla

TUPLAS (,)

- Desempaquetado de tuplas con cadenas o listas

```
>>> correo = 'monty@pyhton.org'
>>> nombre, dominio = correo.split('@')
>>> listado=['oso','pato','perro']
>>> a,b,c = listado
>>> a
'oso'
>>> b
'pato'
>>> c
'perro'
```

Ej.4.14 - Partiendo de una frase (cadena de caracteres), crea una tupla con cada una de las palabras. Dada una palabra, encuentra cuántas veces se repite en la tupla.

Diccionarios. { : }

- Un conjunto de pares clave-valor, que define una relación uno a uno entre claves y valores.

  Por ejemplo: { ‘one’: ‘uno’ , ‘two’: ’dos’ }

- A diferencia de las secuencias, donde los índices son números enteros, los diccionarios están indexados por claves.
- Los diccionarios son mutables
- Las claves han de ser inmutables (no podría poner una lista); los valores pueden ser de cualquier tipo
- El orden de los elementos es impredecible. No tiene sentido un método .sort()

Diccionarios. { : }

```
>>> diccionario = {}
>>> diccionario = dict()
```

Creación de un diccionario vacío

```
>>> diccionario = {'uno':1, 'dos':2, 'tres':3}
>>> diccionario
{'dos': 2, 'tres': 3, 'uno': 1}
>>>
```

En la creación de un diccionario, clave y valor se separan con **:**

```
>>> diccionario['dos']
2
```

El acceso a elementos es con **[clave]**
No se admite **slicing** . Ejemplo [0:2]

Diccionarios. { : }

Inserción

```
>>> vacaciones = {}
>>> vacaciones[‘Javi’] = 3
>>> vacaciones[‘Mario’] = 5
>>> vacaciones[‘Trini’] = 4

>>> vacaciones
{‘Javi': 3, 'Mario': 5, ‘Trini': 4}
```

Diccionarios. { : }

Extracción

```
>>> vacaciones['Javi']
3
>>> vacaciones['Javi'] -= 1
>>> vacaciones['Javi']
2

>>> vacaciones['Carlos']
KeyError: 'Carlos'
```

Si la clave no existe, da un error.
Antes podemos usar el operador **in**

Diccionarios { : }

Búsquedas. Operador **in** . Busca una clave

```
>>> “Mario” in vacaciones
True
>>> 2 in vacaciones
False
```

- El operador ***in*** en los diccionarios. Funciona con un algoritmo de tabla hash (de dispersión) , al contrario que en las listas, donde el algoritmo de búsqueda es lineal y menos eficiente.

Diccionarios. { : }

Eliminación de elementos. Tamaño del diccionario

```
>>> del vacaciones[‘Javi’]
>>> vacaciones
{‘Mario': 5, ‘Trini': 4}

>>> vacaciones
{‘Mario': 5, ‘Trini': 4}
>>> len(vacaciones)
2
```

Si no encuentra la clave a eliminar, da un error.

La función **len()** devuelve el número de parejas clave-valor

Diccionarios. { : }

Métodos keys() y values(). Listas de claves y valores

```
>>> vacaciones.keys()
dict_keys([‘Mario', ‘Trini'])

>>> vacaciones.values()
dict_values([5, 4])
```

Puedo usar la función list() para convertir los tipos devueltos a listas

```
>>> 2 in vacaciones.values()
False
```

Diccionarios. { : }
Métodos:

- **items().** Devuelve una lista de tuplas, cada tupla se compone de dos elementos: el primero será la clave y el segundo, su valor.

```
>>> dic =   {‘a’ : 1, ’b’ : 2, ‘c’ : 3 , ‘d’ : 4}
>>> dic.items()
dict_items([('c', 3), ('a', 1), ('b', 2), ('d', 4)])
```

- **clear().** Elimina todos los ítems del diccionario dejándolo vacío.
- **copy().** Retorna una copia del diccionario original

```
>>> dic2 = dic.copy()
>>> dic2
{‘c': 3, ’a': 1, ‘b': 2, ‘d': 4}
```

Diccionarios. { : }

Métodos:

•**get().** Recibe como parámetro una clave, devuelve el valor de la clave. Si no lo encuentra, devuelve un objeto **None,** o lo que se le ponga en un segundo argumento

```
>>> dic.get('e') is None
True
```

```
>>> dic.get('e',0)
0
```

• **pop().** Recibe como parámetro una clave, elimina esta y devuelve su valor. Si no lo encuentra, devuelve error.

```
>>> dic.pop('b')
2
>>> dic
{'a': 1, 'd': 4, 'c': 3}
```

Diccionarios. { : }
Métodos:

- **update().** Recibe como parámetro otro diccionario. Si se tienen claves iguales, actualiza el valor de la clave repetida; si no hay claves iguales, este par clave-valor es agregado al diccionario.

```
>>> dic1 = {‘a’ : 1, ’b’ : 2, ‘c’ : 3 , ‘d’ : 4}
>>> dic2 = {‘c’ : 6, ’b’ : 5, ‘e’ : 9 , ‘f’ : 10}
>>> dic1.update(dic2)
>>> dic1
{'c': 6, 'd': 4, 'e': 9, 'f': 10, 'a': 1, 'b': 5}
```

Diccionarios. { : }
Iteraciones.

En las lista podíamos usar **enumerate().** En los diccionarios se puede usar el método **.items()**

```
>>> vacaciones={'Javi': 3, 'Mario': 5, 'Trini': 4}
>>> for k, v in vacaciones.items():
...     print (k, v)
...
Mario 5
Trini 4
```

Diccionarios. { : }

Creación de Diccionarios por Comprensión.

En los diccionarios también podemos usar el mecanismo de Comprensión :

Inversión de clave-valor en un diccionario:

```
>>> dict((v, k) for k, v in vacaciones.items())
{3: 'Javi', 5: 'Mario', 4: 'Trini'}

>>> {v:k for k, v in vacaciones.items()}
{3: 'Javi', 5: 'Mario', 4: 'Trini'}
>>>
```

Diccionarios. { : }

Ej.4.15 - Escribe un programa que lea una frase y contabilice las palabras usando un diccionario. Cada vez que se repita una clave, se incrementa su valor.
Luego puedes preguntar por una palabra para saber cuantas veces aparece en la frase.

Ej.4.16 – Imprime en orden alfabético un listado de claves y valor usando el diccionario del ejercicio anterior

Ej.4.17 - Imprime la palabra que más se repite y el total de palabras, usando el diccionario del ejercicio anterior.

Conjuntos. set { }

- Colección no ordenada(1) de objetos únicos.
- Son mutables.
- Sus elementos pueden ser de diversos tipos.
- No pueden incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos.

```
>>> s = {1, 2, 3, 4}
>>> type(s)
<class 'set'>
>>> s = {True, 3.14, None, False, "Hola mundo", (1, 2)}
>>>
```

---

(1) Al ser no ordenada significa que no soporta indexación. Ej: s[1] # ERROR

Conjuntos. set { }

- **Creación de conjuntos:**

Creación de conjuntos vacios. Uso de la función set()

```
>>>s = {}
>>>s
>>>?    # ¿qué sucede ?

>>> s = set()
>>> s
set()
>>> s1 = set([1, 2, 3, 4])
>>> s1
{1, 2, 3, 4}
>>>
```

La función set() puede convertir objetos iterables, como listas o cadenas, a conjuntos

```
>>> di={'a':2,'b':4,'c':6}
>>> set(di)
{'b', 'c', 'a'}
>>> set(di.values())
{2, 4, 6}
```

Conjuntos. set { }

- **Algunos métodos de los conjuntos son:**
  .add() .discard() . remove() .clear() . pop() .issubset() :

```
>>> s={'b', 'c', 'a'}
>>> while s:
...     print(s.pop())
...
b
c
a
```

El método pop() elimina y retorna un elemento en forma arbitraria. Lanzan la excepción KeyError cuando un elemento no se encuentra en el conjunto o bien éste está vacío, respectivamente.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3}
>>> b.issubset(a)
True
```

Relación: subconjunto de....

Conjuntos. set { }

- **Algunos de las propiedades más interesantes de los conjuntos radican en sus operaciones principales:**

Pertenencia (**in**), unión (**|**), intersección (**&**) y diferencia (**-**)

```
>>> 2 in {1, 2, 3}
True
>>> 4 in {1, 2, 3}
False
```

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5, 6}
>>> a & b
{3, 4}
```

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5, 6}
>>> a | b
{1, 2, 3, 4, 5, 6}
```

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3}
>>> a - b
{1, 4}
```

Conjuntos. set { }

Ej.4.18 - Realiza un programa que baraje un mazo de cartas. Como pista utiliza el método pop() de los conjuntos.

Nota: no es necesario que representes la baraja completa.