Listas, Conjuntos y Diccionarios

08

Curso Básico python

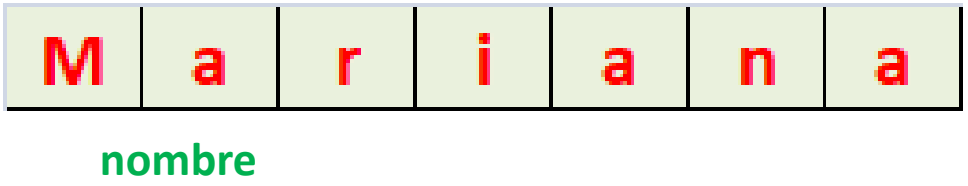
Tipos de Datos

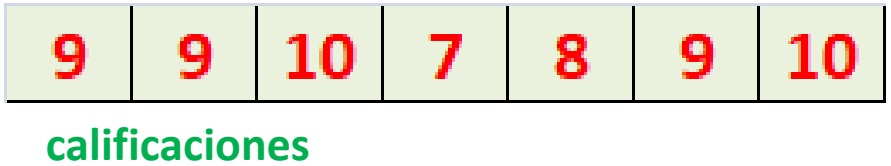
Conceptos generales

Tanto los datos simples como compuestos en Python son tratados como un objeto

- Simples. Tiene asociado un único valor: un entero, un real o un booleano. Son objetos escalares por ser indivisibles, es decir no tienen una estructura interna accesible.

- Compuestos. como los textos (string), las secuencias en forma de listas. Se pueden dividir en elementos y acceder a ellos, son **datos estructurados**.





Datos estructurados o compuestos

- Clasificación de acuerdo a la característica de si sus elementos pueden o no ser cambiados, reducidos o ampliados

- Inmutables (estáticos) valores/tamaño fijos, sus elementos no pueden ser cambiados ni eliminados, tampoco se pueden añadir elementos nuevos. Si se quiere modificar este tipo de datos se utiliza el recurso de crear un nuevo valor.

- Mutables (dinámicos): sus elementos pueden cambiar de valor y se puede añadir o eliminar elementos

Datos estructurados o compuestos

- Inmutables (estáticos):
  - Cadenas de caracteres (string)
  - Tuplas (tuple)
  - Conjuntos congelados (frozenset)
- Mutables (dinámicos):
  - Listas (list)
  - Conjuntos (set)
  - Diccionarios (dict)

Listas
(list)

Listas

- Una lista en Python es una estructura de datos formada por una secuencia ordenada de objetos. Es una secuencia de valores encerrados entre corchetes y separados por comas.
- Los elementos de una lista pueden accederse mediante su índice, siendo 0 el índice del primer elemento.
- **heterogéneas**: pueden estar conformadas por elementos de distintos tipo, incluidos otras listas.
- **mutables**: sus elementos pueden modificarse.
- Es como una tupla mutable.
- Los elementos de las listas pueden ser datos simples (numéricos o booleanos), strings, tuplas u otras listas.

Listas

```
>>> v1 = [2, 4, 6, 8, 10]
>>> type(v1)
<class 'list'>
>>> v2 = [7, 8.5, 'a', 'Hola', (2, 3), [11, 12]]
>>> v2
[7, 8, 'a', 'Hola', (2, 3), [11, 12]]
>>> juegos = ['tennis','baseball','football','voleyball','natación']
>>> juegos
['tennis', 'baseball', 'football', 'voleyball', 'natación']
```

Listas, a partir de string y de tuplas

```
>>> cad = "hola mundo"
>>> cad
'hola mundo'
>>> l1 = list(cad)
>>> l1
['h', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']
```

```
>>> t1 = 2,7,3.7,'hola', 56
>>> t1
(2, 7, 3.7, 'hola', 56)
>>> lista2 = list(t1)
>>> lista2
[2, 7, 3.7, 'hola', 56]
```

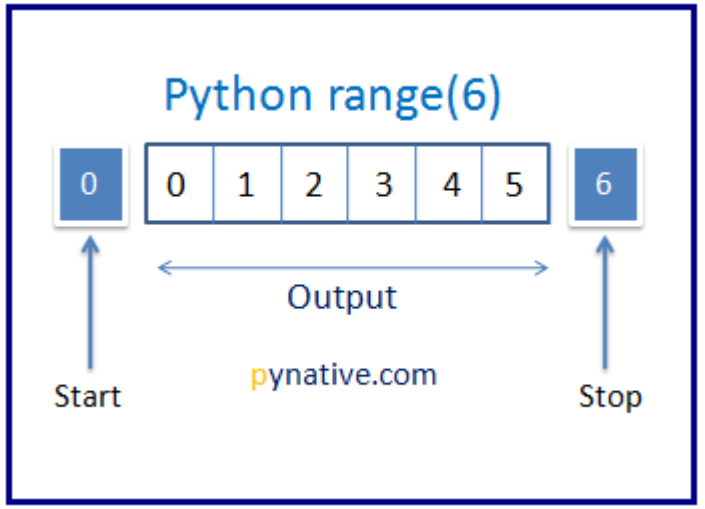
Listas a partir de strings con split()

- Para separar una cadena en frases, los valores pueden separarse con la función integrada

```
>>> mensaje = "Hola, como estas tu?"
>>> mensaje.split() # retorna una lista
['Hola,', 'como', 'estas', 'tu?']
```

Listas a partir de range()

- El tipo range es una lista inmutable de números enteros en sucesión aritmética.



Syntax:

```
range([start,] stop [, step]) -> range object
```

PARAMETER	DESCRIPTION
`start`	(optional) Starting point of the sequence. It defaults to `0`.
`stop` (required)	Endpoint of the sequence. This item will not be included in the sequence.
`step` (optional)	Step size of the sequence. It defaults to `1`.

```
>>> range(4)
range(0, 4)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(range(3,6))
[3, 4, 5]
>>> list(range(1,13,2))
[1, 3, 5, 7, 9, 11]
```

Turbo Código
https://www.youtube.com/user/juarezefren

Listas (Operaciones)

Operación	Resultado
`x in s`	Indica si la variable `x` se encuentra en `s`
`s + t`	Concantena las secuencias `s` y `t`
`s * n`	Concatena `n` copias de `s`
`s[i]`	Elemento `i` de `s`, empezando por `0`
`s[i:j]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive)
`s[i:j:k]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive), con paso `k`
`len(s)`	Cantidad de elementos de la secuencia `s`
`min(s)`	Mínimo elemento de la secuencia `s`
`max(s)`	Máximo elemento de la secuencia `s`

```
>>> lista=[4,7,2,67,4]
>>> min(lista)
2
>>> max(lista)
67
>>> lista[-1]
4
```

lista[1] = 44.78

Las listas si son mutables

Acceso a elementos de una lista anidada

```
>>> lista1 = [23,True,67,13.8,'sol']
>>> lista2= [90,45,lista1,100,234]
>>> lista2
[90, 45, [23, True, 67, 13.8, 'sol'], 100, 234]
>>> lista2[2][1]
True
```

Listas (Métodos)

- count(x) # cuenta la cantidad de veces que x aparece en la lista
- index(x [,ini,fin]) # devuelve el índice de la primera aparición de x en la lista. Opcionalmente se puede indicar donde iniciar ini y donde terminar fin la búsqueda. Devuelve un excepción ValueError si el elemento x no se encuentra en la lista, o en el entorno definido.
- append(x) # agrega un elemento x al final de una lista.
- insert(i , x) #inserta el elemento x en la lista, en el índice i.
- extend(x) # extiende una lista agregando un iterable x al final.
- pop([ind]) #devuelve el último elemento de la lista, y lo borra de la misma. Opcionalmente puede recibir un argumento numérico ind, que funciona como índice del elemento (por defecto, -1)
- remove(x) #recibe como argumento un elemento x, y borra su primera aparición en la lista, devuelve un excepción ValueError si el elemento no se encuentra en la lista
- reverse() #invierte el orden de los elementos de una lista.
- sort([reverse=True]) #ordena los elementos de una lista, admite la opción reverse, por defecto, con valor False. De tener valor True, el ordenamiento se hace en sentido inverso.
- clear() # remueve todos los elementos de la lista
- copy() # # copia la lista en otro objeto, tambien existe deepcopy() # con sub-listas

Objetos, valores y referencias

Caso: string

- El operador **is** indica si dos variables están referidas al mismo objeto o no.
- El método id() devuelve la dirección de memoria

```
>>> a = 'casa'
>>> b = 'casa'
>>> id(a)
123917904
>>> id(b)
123917904
>>> a is b
True
```

Ambas variables a y b están referidas al mismo objeto, que tiene valor 'casa' y ocupa la posición de memoria 123917904 (esta posición es arbitraria). La instrucción a is b es cierta.

En los tipos de dato string, al ser inmutables, Python crea solo un objeto por economía de memoria y ambas variables están referidas al mismo objeto. Sin embargo con las listas, al ser mutables, aunque se formen dos listas con los mismos valores, Python crea dos objetos, que ocupan diferente posición de memoria.

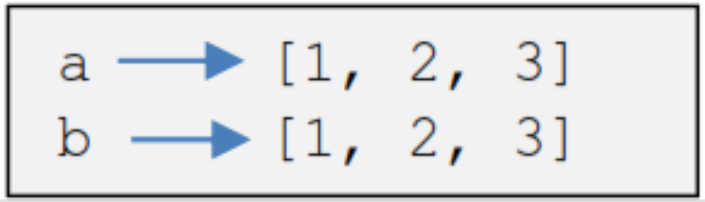
Objetos, valores y referencias

Caso:
list

- El operador **is** indica si dos variables están referidas al mismo objeto o no.
- El método id() devuelve la dirección de memoria

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
123921992
>>> id(b)
123923656
>>> a is b
False
```

Las listas asignadas a las variables a y b, aunque con el mismo valor, son objetos diferentes.



Pero hay que tener cuidado con las asignaciones de variable al mismo objeto mutable b=a ya que no se crea otro objeto sino que el copiado se refiere al mismo objeto.

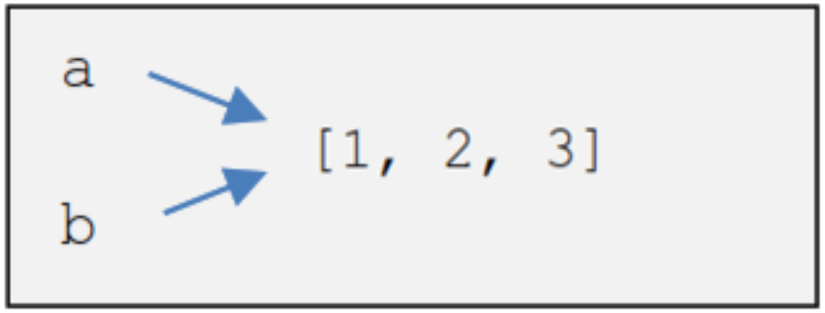
Objetos, valores y referencias

Caso:
list

- El operador **is** indica si dos variables están referidas al mismo objeto o no.
- El método id() devuelve la dirección de memoria

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(b)
123921992
>>> a is b
True
```

Al copiar (o asignar) una variable a otra, con el operador igual (=) no se crea otro objeto sino que el copiado se refiere al mismo objeto.




```
>>> b[0] = 15
>>> a
[15, 2, 3]
```

Por lo tanto si modificamos o añadimos un valor al objeto [1, 2, 3], a través de una de las variables, entonces modificamos la otra.

Métodos de separar (*split) y unir (joint) string-listas:*

```
>>> str = '10/04/2018'
>>> lst = str.split('/')
>>> lst
['10', '04', '2018']
>>> str2 = '/'.join(lst)
>>> str2
'10/04/2018'
```

Listas anidadas (tablas o matrices)

Una lista anidada es una lista donde sus elementos son a su vez listas. Este tipo de objeto es útil para representar tablas o matrices de datos. Por ejemplo, la matriz

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Se puede escribir en Python como

```
>>> m = [[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12]]
```

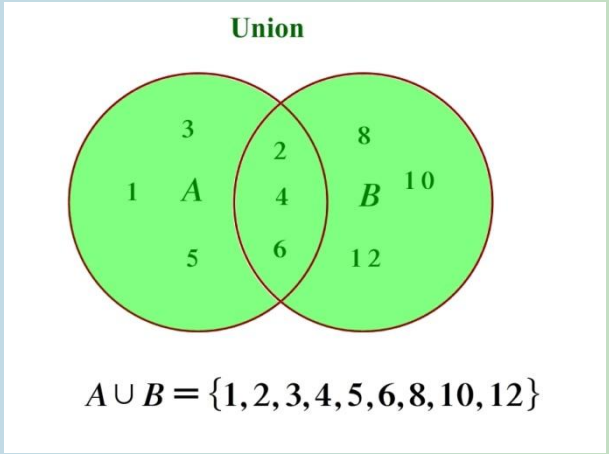
En un editor de Python se puede escribir saltando a la siguiente línea después de la coma que separa cada elemento de la lista (que representa una fila de la matriz):

```
m = [[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12]]
```

NumPy

- Los arreglos de números o tablas se pueden trabajar mucho más eficientemente con los objetos *array del paquete (package) de NumPy*
- Estos módulos de cálculo numérico son una extensión del Python, pero muy útiles para el tratamiento de vectores y matrices, de forma similar al Matlab.

Conjuntos (Set)



Conjuntos (Set)

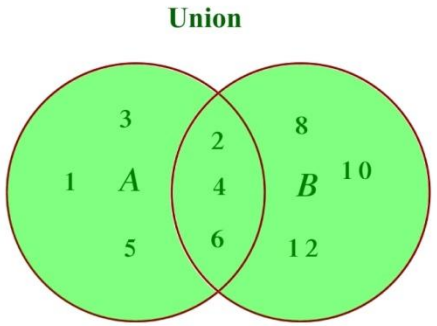
- Los conjuntos son una colección de elementos únicos e inmutables que no están ordenados. Siguen la idea de los conjuntos en matemáticas, donde los elementos no deben repetirse.
- Pueden contener números, string, tuplas, pero no listas.
- Los conjuntos no pueden indexarse ni recortarse (slice), pero se le pueden añadir o quitar elementos aunque solo a través de sus métodos asociados.

Conjuntos (Set)

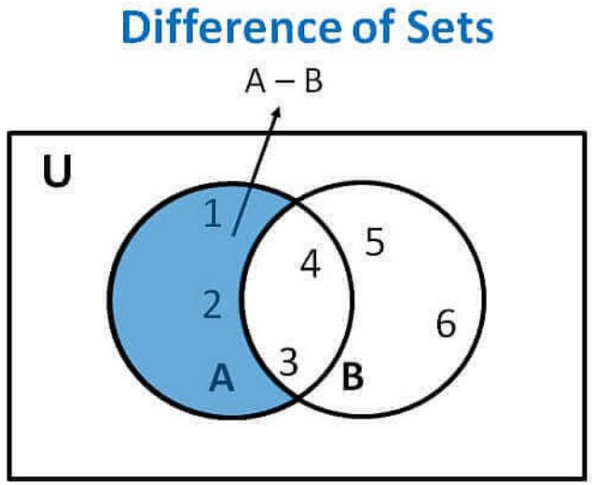
```
>>> a=set('hola mundo')
>>> b = set([4,6,7])
>>> a
{'h', 'a', 'o', 'l', 'd', ' ', 'm', 'u', 'n'}
>>> b
{4, 6, 7}
>>> len(a)
9
```

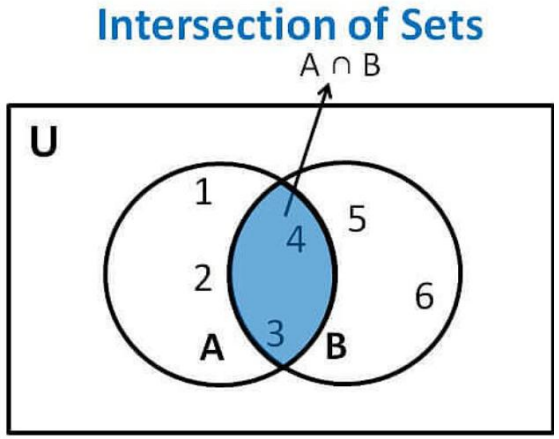
```
>>> S1 = {25, 4, 'a',2, 25, 'casa', 'a'}
>>> S1
{'casa', 'a', 2, 4, 25}          # los elementos repetidos fueron desechados
>>> type(S1)
<class 'set'>
>>> S1.add('b')
>>> S1
{'casa', 'a', 2, 4, 'b', 25}
>>> S1.remove('a')
>>> S1
{'casa', 2, 4, 'b', 25}
```

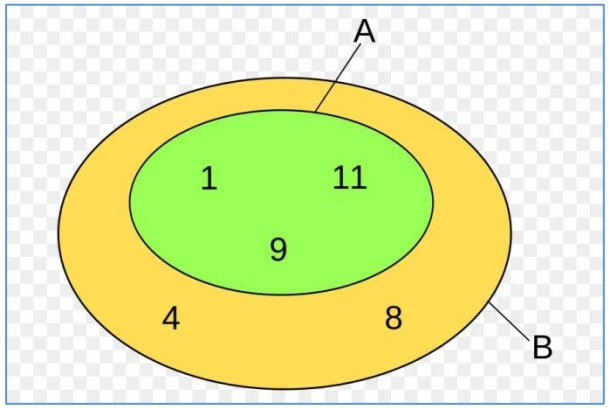
Conjuntos (operaciones)



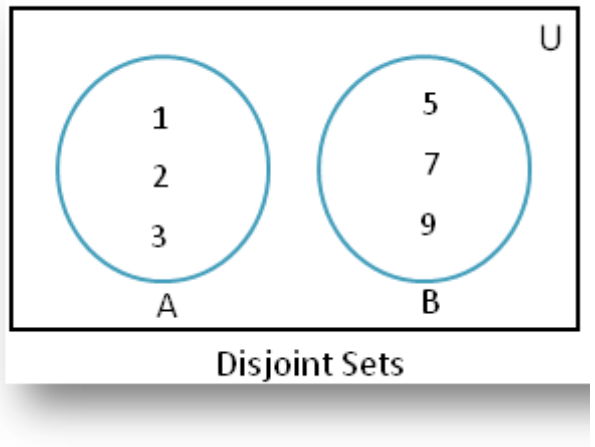
$A \cup B = \{1, 2, 3, 4, 5, 6, 8, 10, 12\}$







Subconjunto



Turbo Código
https://www.youtube.com/user/juarezefren

Conjuntos (Métodos)

- add(x) #agrega el elemento x en el conjunto
- clear() # remueve todos los elementos
- copy() # copia superficial del conjunto
- a.difference(b) # igual a-b
- a.difference_update(b) # # igual a = a-b
- a.discard(elem) # remueve elem de a
- a.intersection(b)
- a.intersection_update(b)

- a.union(b)
- a. isdisjoint(b)
- a. issubset(b) # True si a es sub-conjunto de b
- a. issuperset(b) # True si a es super-conjunto de b
- a.pop() # remueve un elemento arbitrario de a #el primero
- a.remove(elem) # remueve elem de a
- a.update(x) # Agrega x , siendo x un iterable

Diccionarios (dict)

Diccionarios (dict)

- El diccionario, define una relación uno a uno entre claves y valores.

{ ‘clave1’ : valor1 , ‘clave2’ : valor2}

Clase	Tipo	Notas	Ejemplo
`dict`	Mapeos	Mutable, sin orden.	`{'cms':"Plone", 'version':5}`

Diccionarios (dict)

- Los diccionarios pueden ser creados colocando una lista separada por coma de pares “key:value” entre {}, por ejemplo: {'python': 27, 'plone': 51} o {27:'python', 51:'plone‘} o por el constructor “*dict()*

```
>>> diccionario = {
      "clave1":234,
      "clave2":True,
      "clave3":"Valor 1",
      "clave4":[1,2,3,4]
 }
>>> diccionario
{'clave1': 234, 'clave2': True, 'clave3': 'Valor 1', 'clave4': [1, 2, 3, 4]}
>>> type(diccionario)
<class 'dict'>
```

{ ‘clave1’ : valor1 , ‘clave2’ : valor2}

Diccionarios (acceso a los elementos)

- Se utiliza la clave

```
>>> diccionario
{'clave1': 234, 'clave2': True, 'clave3': 'Valor 1', 'clave4': [1, 2, 3, 4]}
>>> diccionario['clave1']
234
>>> diccionario['clave2']
True
>>> diccionario['clave3']
'Valor 1'
>>> diccionario['clave4']
[1, 2, 3, 4]
```

```
>>> type(diccionario['clave1'])
<type 'int'>
>>> type(diccionario['clave2'])
<type 'bool'>
>>> type(diccionario['clave3'])
<type 'str'>
>>> type(diccionario['clave4'])
<type 'list'>
```

{ ‘clave1’ : valor1 , ‘clave2’ : valor2}

Diccionarios (creacion con dic())

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1, django=2.1)
>>> versiones
{'python': 2.7, 'zope': 2.13, 'plone': 5.1, 'django': 2.1}
>>> versiones['zope']
2.13
>>> type(versiones['zope'])
<class 'float'>
```

Operación in

Este operador es el mismo operador integrado *in* en el interprete Python pero aplicada al uso de la secuencia de tipo **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1, django=2.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1, 'django': 2.1}
>>> 'plone' in versiones
True
>>> 'flask' in versiones
False
```

En el ejemplo anterior este operador devuelve `True` si la clave esta en el diccionario `versiones`, de lo contrario devuelve `False`.

Métodos

clear()

Este método remueve todos los elementos desde el **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.clear()
>>> print versiones
{}
```

copy()

Este método devuelve una copia superficial del tipo **diccionario**:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> otro_versiones = versiones.copy()
>>> versiones == otro_versiones
True
```

Métodos

fromkeys()

Este método crea un nuevo **diccionario** con *claves* a partir de un tipo de dato *secuencia*. El valor de `value` por defecto es el tipo *None*.

```
>>> secuencia = ('python', 'zope', 'plone')
>>> versiones = dict.fromkeys(secuencia)
>>> print "Nuevo Diccionario : %s" %  str(versiones)
Nuevo Diccionario : {'python': None, 'zope': None, 'plone': None}
```

En el ejemplo anterior inicializa los valores de cada clave a `None`, mas puede inicializar un *valor* común por defecto para cada *clave*:

```
>>> versiones = dict.fromkeys(secuencia, 0.1)
>>> print "Nuevo Diccionario : %s" %  str(versiones)
Nuevo Diccionario : {'python': 0.1, 'zope': 0.1, 'plone': 0.1}
```

get()

Este método devuelve el valor en base a una coincidencia de búsqueda en un diccionario mediante una clave, de lo contrario devuelve el objeto *None*.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.get('plone')
5.1
>>> versiones.get('php')
>>>
```

Métodos

has_key()

Este método devuelve el valor True si el diccionario tiene presente la clave enviada como argumento.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.has_key('plone')
True
>>> versiones.has_key('django')
False
```

items()

Este método devuelve una lista de pares de diccionarios (clave, valor), como 2 tuplas.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.items()
[('zope', 2.13), ('python', 2.7), ('plone', 5.1)]
```

Métodos

iteritems()

Este método devuelve un *iterador* sobre los elementos (clave, valor) del diccionario. Lanza una excepción *StopIteration* si llega al final de la posición del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.iteritems()
<dictionary-itemiterator object at 0x7fab9dd4bc58>
>>> for clave,valor in versiones.iteritems():
...     print clave,valor
...
zope 2.13
python 2.7
plone 5.1
>>> versionesIterador = versiones.iteritems()
>>> print versionesIterador.next()
('zope', 2.13)
>>> print versionesIterador.next()
('python', 2.7)
>>> print versionesIterador.next()
('plone', 5.1)
>>> print versionesIterador.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Métodos

iterkeys()

Este método devuelve un *iterador* sobre las claves del diccionario. Lanza una excepción *StopIteration* si llega al final de la posición del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.iterkeys()
<dictionary-keyiterator object at 0x7fab9dd4bcb0>
>>> for clave in versiones.iterkeys():
...     print clave
...
zope
python
plone
>>> versionesIterador = versiones.iterkeys()
>>> print versionesIterador.next()
zope
>>> print versionesIterador.next()
python
>>> print versionesIterador.next()
plone
>>> print versionesIterador.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

itervalues()

Este método devuelve un *iterador* sobre los valores del diccionario. Lanza una excepción *StopIteration* si llega al final de la posición del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.itervalues()
<dictionary-valueiterator object at 0x7fab9dd4bc58>
>>> for valor in versiones.itervalues():
...     print valor
...
2.13
2.7
5.1
>>> versionesIterador = versiones.itervalues()
>>> print versionesIterador.next()
2.13
>>> print versionesIterador.next()
2.7
>>> print versionesIterador.next()
5.1
>>> print versionesIterador.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Métodos

keys()

Este método devuelve una lista de las claves del diccionario:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.keys()
['zope', 'python', 'plone']
```

pop()

Este método remueve específicamente una clave de **diccionario** y devuelve valor correspondiente. Lanza una excepción *KeyError* si la **clave** no es encontrada.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.pop('zope')
2.13
>>> versiones
{'python': 2.7, 'plone': 5.1}
>>> versiones.pop('django')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'django'
```

Métodos

popitem()

Este método remueve y devuelve algún par (clave, valor) del **diccionario** d
Lanza una excepción *KeyError* si el **diccionario** esta vació.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.popitem()
('zope', 2.13)
>>> versiones
{'python': 2.7, 'plone': 5.1}
>>> versiones.popitem()
('python', 2.7)
>>> versiones
{'plone': 5.1}
>>> versiones.popitem()
('plone', 5.1)
>>> versiones
{}
>>> versiones.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

update()

Este método actualiza un **diccionario** agregando los pares clave-valores en un segundo diccionario. Este método no devuelve nada.

El método `update()` toma un diccionario o un objeto iterable de pares clave/valor (generalmente tuplas). Si se llama a `update()` sin pasar parámetros, el diccionario permanece sin cambios.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones_adicional = dict(django=2.1)
>>> print versiones_adicional
{'django': 2.1}
>>> versiones.update(versiones_adicional)
```

Como puede apreciar este método no devuelve nada, más si muestra de nuevo el diccionario `versiones` puede ver que este fue actualizado con el otro diccionario `versiones_adicional`.

```
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1, 'django': 2.1}
```

Métodos

values()

Este método devuelve una lista de los valores del diccionario:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.values()
[2.13, 2.7, 5.1]
```

Métodos

- viewkeys() #Este método devuelve un objeto proveyendo una vista de las claves del diccionario.
- viewitems() #Este método devuelve un objeto como un conjunto mutable proveyendo una vista en los elementos del diccionario:
- viewvalues() #Este método devuelve un objeto proveyendo una vista de los valores del diccionario.
- cmp(dic1,dic2) #compara dos diccionarios -> 0 si son iguales

Operación	Descripción
len(d)	Devuelve el número de ítems o pares clave-valor
del d(k)	elimina el ítem con clave k
k **in** d	True, si la clave k existe en el diccionario d
k **not in** d	True, si la clave k no existe en el diccionario d

Listas, Conjuntos y Diccionarios

08

Curso Básico python