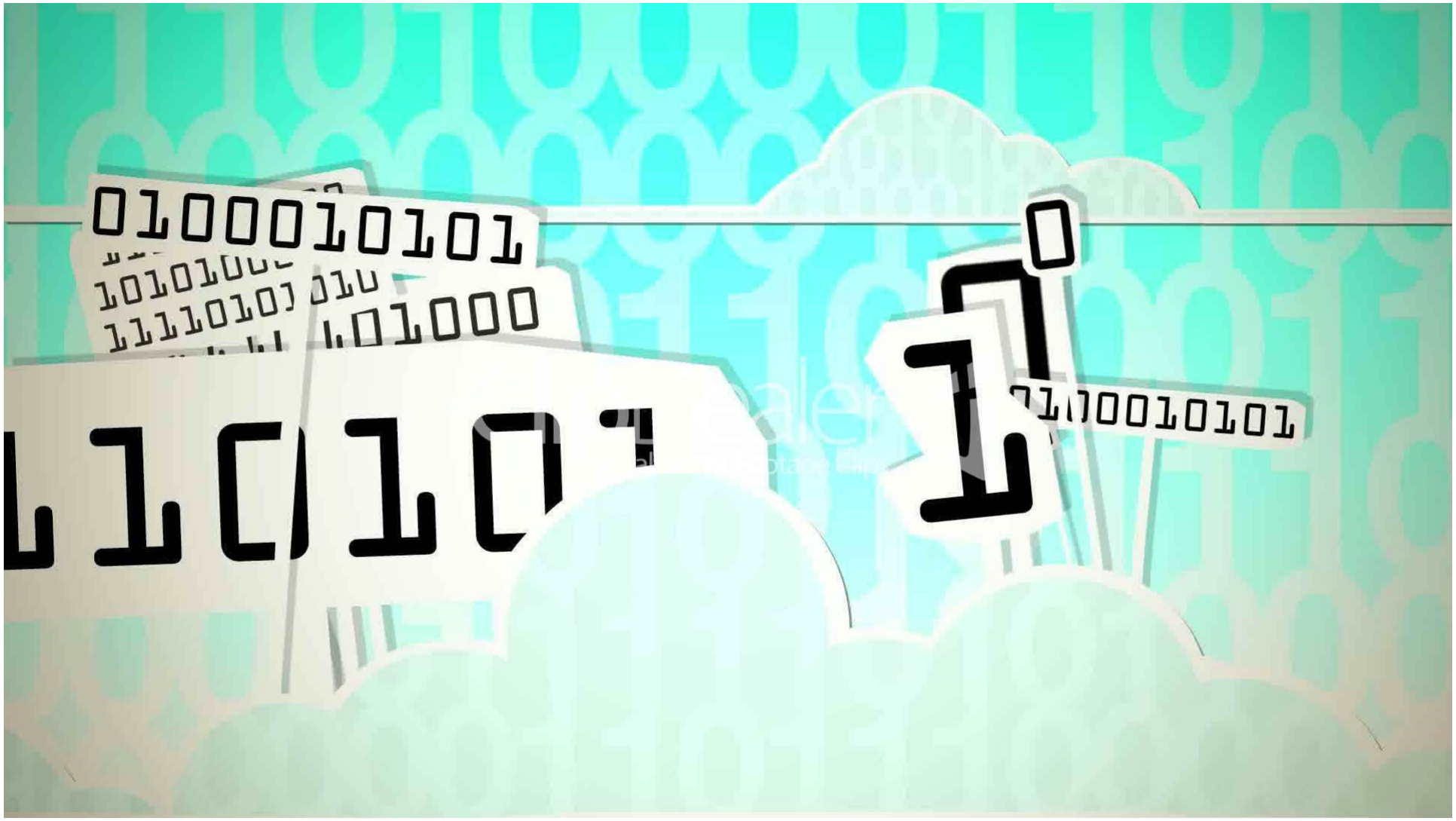
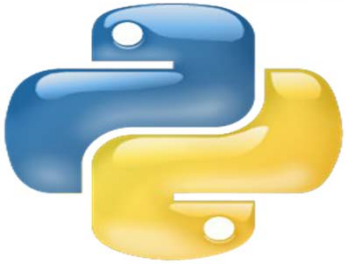


Introducción a Python 3



COLECCIONES

- **Tupla ( )**
  Una secuencia ordenada e ***inmutable*** de elementos. Los elementos pueden ser de diferentes tipos, incluyendo otras colecciones.

- **Cadenas (Strings) “ ”**
  Inmutable. Conceptualmente iguales a las Tuplas cuyos elementos son caracteres alfanuméricos.

- **Listas [ ]**
  Secuencia ordenada ***Mutable*** de elementos de diferentes tipos.

- **Diccionarios { }**
  Conjunto ***Mutable*** no ordenado de ***claves:valor*** de elementos de diferentes tipos.

COLECCIONES TIPO SECUENCIA

- Las **tuplas** se definen como una lista de elementos ordenados separados por comas, entre paréntesis .

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]          # Segundo elemento de la tupla.
 'abc'
```

- Las **listas** igual, solo que entre corchetes.

```
>>> li = ["abc", 34, 4.34, 23]
>>> li[1] # Segundo elemento de la lista.
 34
```

- Las **cadenas** se escriben entre comillas (", ', """").

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """Hello esta es una multi-línea que utiliza triple comillas
dobles."""
>>> st[1] # Segundo elemento de la cadena.
 'e'
```

CORTES (SLICING)

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Devuelve una copia del contenedor con un subconjunto de los miembros originales. Empieza a copiar desde el primer índice y se detiene antes del segundo.

```
>>> t[1:4]
(‘abc’, 4.56, (2,3))
```

- También se pueden usar índices negativos.

```
>>> t[1:-1]
(‘abc’, 4.56, (2,3))
```

COPIANDO TODA LA SECUENCIA

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Para devolver una *copia* de toda la secuencia, puedes usar `[:]`.

```
>>> t[:]
(23, ‘abc’, 4.56, (2,3), ‘def’)
```

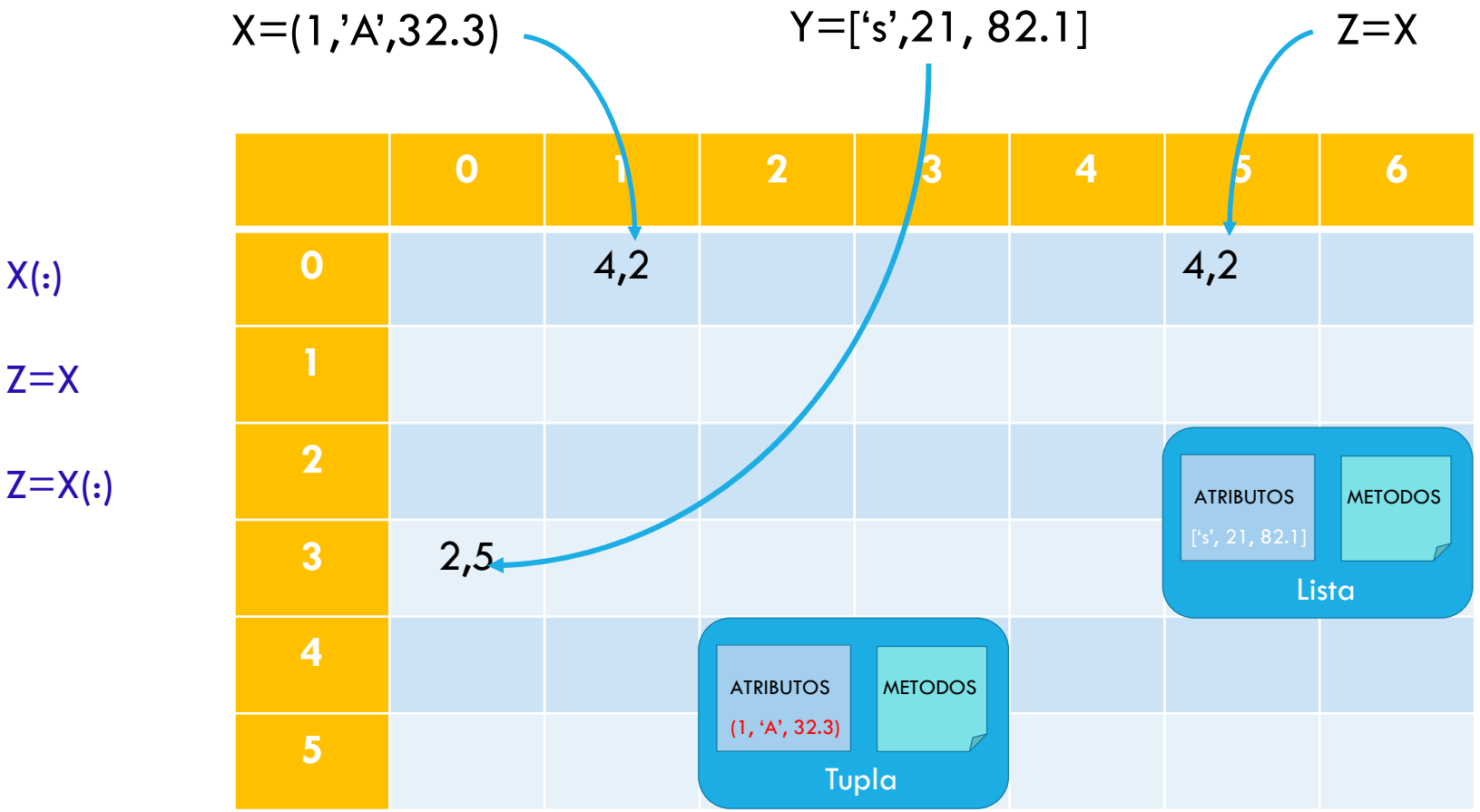
- La diferencia entre estas dos líneas es muy importante:

```
>>> list2 = list1 # 2 nombres hacen referencia a la misma lista
                  # Si cambiamos una se cambian ambas

>>> list2 = list1[:] # Dos copias diferentes, dos refs
```

RECORDANDO ASIGNACIÓN

Tabla Asignación Memoria	
X	0,1
Y	3,0
Z	0,5



OPERADORES 'IN' Y 'NOT IN'

- Una prueba booleana para ver si un elemento está en la secuencia:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- En las cadenas, comprueba si una subcadena está en la secuencia

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

EL OPERADOR +

- El operador + produce una *nueva* tupla, lista o cadena cuyos valores son la concatenación de los argumentos.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]


>>> “Hello” + “ ” + “World”
 ‘Hello World’
```

EL OPERADOR *

- El operador * produce una *nueva* tupla, lista o cadena cuyos valores son la repetición de los argumentos.

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

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
 HelloHelloHello’
```

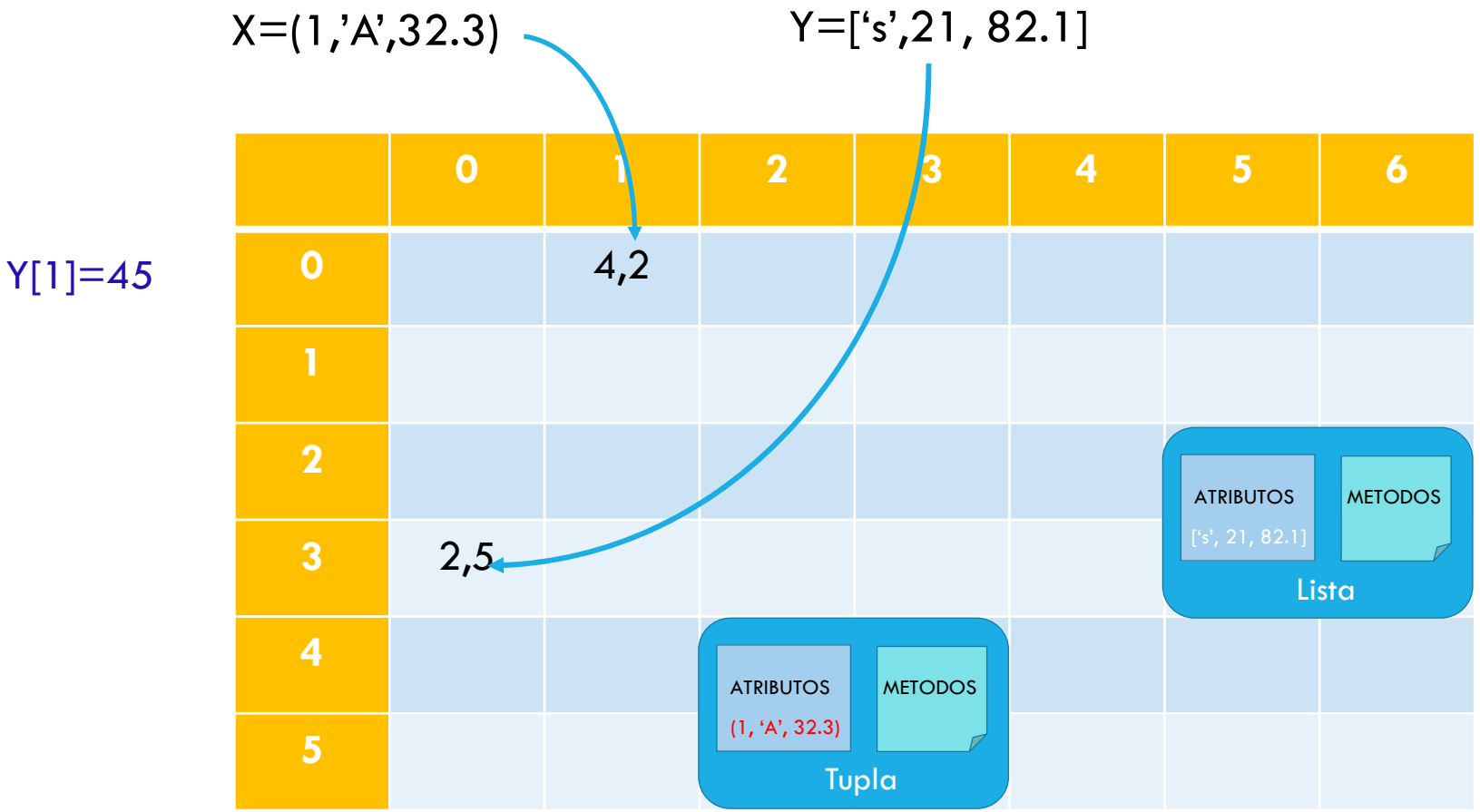
LISTAS: *MUTABLES*

```
>>> li = [‘abc’, 23, 4.34, 23]
>>> li[1] = 45
>>> li
   [‘abc’, 45, 4.34, 23]
```

- Podemos modificar la lista *in situ*.
- El nombre *li* sigue apuntando a la misma dirección de memoria después de la actualización.

EN MEMORIA

Tabla Asignación Memoria	
X	0,1
Y	3,0
Z	0,5



MÉTODOS SOLO PARA LISTAS (1)

```
>>> li = [1, 11, 3, 4, 5]
```

append( )

```
>>> li.append('a')    # Se utiliza un método de la lista
>>> li
[1, 11, 3, 4, 5, 'a']
```

insert( )

```
>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

MÉTODOS SOLO PARA LISTAS (2)

extend()

- *El operador* + crea una nueva lista (con una nueva referencia)

extend altera la lista `li` in place.

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Cuidado:
extend toma como argumento una lista.
append toma un solo elemento como argumento.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

MÉTODOS SOLO PARA LISTAS (3)

```
>>> li = [‘a’, ‘b’, ‘c’, ‘b’]
```

index()

```
>>> li.index(‘b’)      # indice de primera ocurrencia*
1
```

*existen otras formas

count()

```
>>> li.count(‘b’)      # numero de ocurrencias
2
```

remove()

```
>>> li.remove(‘b’)     # elimina la primera ocurrencia
>>> li
  [‘a’, ‘c’, ‘b’]
```

MÉTODOS DE LISTAS

```
>>> li = [5, 2, 6, 8]
```

reverse()

```
>>> li.reverse()    # ordena al revés   la lista *in place*
>>> li

  [8, 6, 2, 5]
```

sort()

```
>>> li.sort()       # ordena la lista *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(alguna_funcion)
    # se ordena utilizando la función recibida
```

DE CADENA A LISTA A CADENA

join pasa una lista de cadenas a una sola cadena. (*es un método de la cadena separadora*)

```
<cadena_separadora>.join( <lista> )
```

```
>>> ";".join( ["abc", "def", "ghi"] )
"abc;def;ghi"
```

split convierte una cadena a una lista de cadenas. (*es un método de la lista a separar*)

```
<alguna_cadena>.split( <cadena_separadora> )
```

```
>>> "abc;def;ghi".split( ";" )
["abc", "def", "ghi"]
```

Fíjate en el cambio de responsable de la operación

CONVERTIR ENTRE LISTAS Y TUPLAS

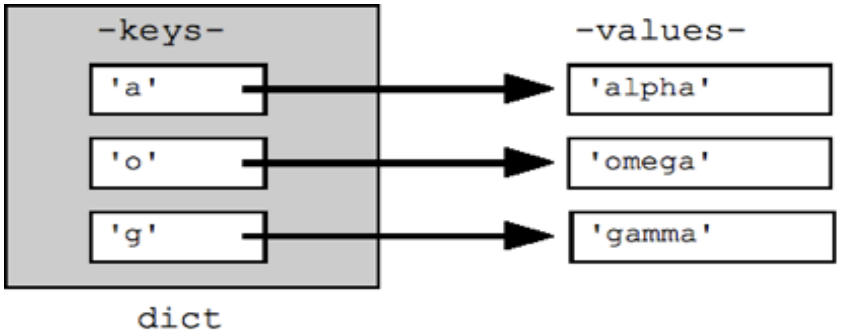
Las listas son más lentas pero más potentes que las tuplas.

Las listas se pueden modificar y permiten más operaciones. Las tuplas son mas rápidas pero son inmutables.

Para convertir entre ellas utiliza las funciones list() y tuple():

```
li =list(tu)
tu =tuple(li)
```

DICCIONARIOS



Los diccionarios contienen entre llaves {} un conjunto desordenado de parejas clave:valor.

d = {‘usuario’:‘bozo’, ‘pswd’:1234}

- Las claves pueden ser cualquier tipo de dato inmutable.
- Los valores pueden ser de cualquier tipo.
- Un diccionario puede almacenar diferentes tipos de datos.
- Los diccionarios no tienen orden.

DEFINIR Y RECUPERAR EN DICCIONARIOS

```
>>> d = {'usuario':'bozo', 'pswd':1234}
>>> d['usuario']
>>> 'bozo'

>>> d['pswd']
1234
>>> d['bozo']
Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

ACTUALIZANDO DICCIONARIOS

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

Las claves deben de ser únicas.
Al asignar a una clave existente, remplaza el valor.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

ELIMINANDO ELEMENTOS DE LOS DICCIONARIOS

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
```

del

```
>>> del d['user']              # Elimina un par clave-valor.
>>> d
{'p':1234, 'i':34}
```

clear()

```
>>> d.clear()                  # Elimina todos los pares.
>>> d
{}
>>> a=[1,2]
>>> del a[1]                   # (del también se usa en listas)
>>> a
[1]
```

MÉTODOS ÚTILES PARA ACCEDER A DICCIONARIOS

```
>>> d = {'user':'bozo','p':1234, 'i':34}
```

keys()

```
# Lista de claves
>>> d.keys()
['user', 'p', 'i']
```

values()

```
>>> d.values()          # Lista de valores.
['bozo', 1234, 34]
```

items()

```
>>> d.items()      # Diccionario como una lista de tuplas.
[('user','bozo'), ('p',1234), ('i',34)]
```

BUCLES FOR

- Un bucle for recorre cada uno de los elementos de una colección, o cualquier objeto “iterable”

```
for <elemento> in <colección>:
    <sentencias>
```

- Si <colección> es una lista o tupla, el for recorre cada elemento de la colección.

- Si <colección> es una cadena, entonces el ciclo recorre cada carácter de la cadena.

```
for caracter in “Hello World”:
    print (caracter)
```

BUCLES FOR: PLANTILLAS O PATRONES

```
for <elemento> in <colección>:
    <sentencias>
```

- **<elemento>** puede ser complejo.

  Cuando los elementos de una <colección> son a su vez secuencias, entonces <elemento> puede ser una “plantilla” o patrón de la estructura de los elementos. Esto facilita el acceso a los elementos internos.

```
for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:
  print (x)
```

DICCIONARIOS Y FOR

```
>>> edades = { "Sam " :4, "Mary " :3, "Bill " :2 }

>>> for nombre in edades.keys():
        print (nombre, edades[nombre])

Bill 2
Mary 3
Sam 4

>>> for key in sorted(edades.keys()):
        print (key, edades[key])
```

LISTAS POR COMPRENSIÓN

Una característica poderosa del lenguaje.

Generas una nueva lista aplicando una función a cada elemento de la lista original.

[expresión `for` nombre `in` lista]

- La expresión es alguna operación sobre nombre.
- Cada elemento de la lista, se asigna a nombre. y se calcula el nuevo elemento utilizando la expresión.
- Los elementos resultantes se van recolectando en una nueva lista la cual se devuelve como resultado de comprensión.

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

LISTAS POR COMPRENSIÓN

[expresión for nombre in lista]

- Si la lista contiene elementos de distintos tipos, entonces la expresión debe ser capaz de operar correctamente con todos los elementos de la lista.
- Si los elementos de la lista son a su vez contenedores, entonces el nombre puede consistir de patrones de nombres que “empaten” con los elementos de la lista.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

LISTAS POR COMPRENSIÓN

`[ expresión for nombre in lista ]`

La expresión puede contener funciones.

```
>>> def subtract(a, b):
        return a - b
>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

LISTAS POR COMPRENSIÓN: FILTROS

```
[ expresión for nombre in lista if  filtro  ]
```

La expresión booleana filtro determina si el elemento se evaluará o no por la expresión.

Si filtro es *False* entonces el elemento se omite de la lista antes de que se evalúe la comprensión.

CON FILTROS Y ANIDADAS

[expresión for nombre in lista if filtro]

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

Como la operación toma una lista como entrada y produce una lista como salida, éstas se pueden anidar fácilmente:

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
          [item+1 for item in li] ]
[8, 6, 10, 4]
```

ARCHIVOS

Un **archivo** en Python es un **objeto** al que tenemos que asignar una variable.

- **Abrir archivos** fileObject = **open**(file_name [, access_mode][, buffering])

  open - Devuelve un objeto tipo archivo
  Buffering – Si es 0 el se escribe en el archivo inmediatamente, si no esto no es así.

  Algunos modos de acceso:

  - "**r**" abre un archivo solo para lectura (si no aparece ningún modo de acceso es lectura).
  - "**w**" abre un archivo solo para escritura. Sobreescribe el archivo si ya existe. Si no, crea un nuevo archivo.
  - "**a**" Abre un archivo para añadir (append). Si el archivo no existe, crea uno nuevo para escribir en él.
  - Añadiendo una 'b' se leen o escriben **bytes strings** en binario ("**rb**", "**wb**", "**ab**")

- **Cerrar archivos** fileObject.**close**()

  El método close() escribe cualquier información que aún no haya sido escrita y cierra el objeto archivo.

OPERACIONES SOBRE ARCHIVOS

r	Lectura
r+	Lectura/Escritura
w	Sobreescritura. Si no existe archivo se creará
a	Añadir. Escribe al final del archivo
b	Binario
+	Permite lectura/escritura simultánea
U	Salto de línea universal: win cr+lf, linux lf y mac cr
rb	Lectura binaria
wb	Sobreescritura binaria
r+b	Lectura/Escritura binaria

MANEJO DE ARCHIVOS

Los archivos son objetos de la clase File y por lo tanto tienen accesibles todos sus métodos

- **Leer de un archivo** fileObject.**read**([count])
  - El método **read**() lee el archivo completo de una pasada.
  - El método **readline**() lee una línea del archivo cada vez.
  - El método **readlines**() lee todas las líneas de un archivo y las pone en una lista.

  Todo se lee como cadenas de caracteres.

- **Escribir en un archivo** fileObject.**write**(string)
  - El método **write()** escribe una cadena en un archivo abierto.
  - El método **writelines()** escribe toda una lista en un archivo abierto

EJEMPLOS

```
# Abre archivo en modo lectura
archivo = open('archivo.txt','r')

# Lee los 9 primeros bytes
cadena1 = archivo.read(9)

# Lee la información restaste
cadena2 = archivo.read()

# Muestra la primera lectura
print(cadena1)

# Muestra la segunda lectura
print(cadena2)

# Cierra el archivo
archivo.close()
```

```
cadena1 = 'Datos' # declara cadena1
cadena2 = 'Secretos' # declara cadena2

# Abre archivo para escribir
 archivo = open('datos1.txt','w')

# Escribe cadena1 añadiendo salto de línea
archivo.write(cadena1 + '\n')

# Escribe cadena2 en archivo
archivo.write(cadena2)

# cierra archivo
archivo.close()

# Declara lista
lista = ['lunes', 'martes', 'miercoles', 'jueves', 'viernes']

# Abre archivo en modo escritura
archivo = open('datos2.txt','w')

# Escribe toda la lista en el archivo
archivo.writelines(lista)

# Cierra archivo
archivo.close()
```

MANEJO DE ARCHIVOS (SALTO Y LECTURA)

- El método **seek()** desplaza el puntero a una posición del archivo
- El método **tell()** devuelve la posición del puntero en un momento dado (en bytes).

```
# Abre archivo en modo lectura
archivo = open('datos2.txt','r')

# Mueve puntero al quinto byte
archivo.seek(5)

# lee los siguientes 5 bytes
cadena1 = archivo.read(5)

# Muestra cadena
print(cadena1)

# Muestra posición del puntero
print(archivo.tell())

# Cierra archivo
archivo.close()
```

MANEJO DE ARCHIVOS (PICKLE)

Para leer y escribir cualquier tipo de objeto Python podemos importar el modulo **pickle** y usar sus métodos **dump**() y **load**() para leer y escribir los datos.

```
# Importa módulo pickle
import pickle

# Declara lista
lista = ['Perl', 'Python', 'Ruby']

# Abre archivo binario para escribir
archivo = open('lenguajes.dat', 'wb')

# Escribe lista en archivo
pickle.dump(lista, archivo)

# Cierra archivo
archivo.close()

# Borra de memoria la lista
del lista

# Abre archivo binario para leer
archivo = open('lenguajes.dat', 'rb')

# carga lista desde archivo
lista = pickle.load(archivo)

# Muestra lista
print(lista)

# Cierra archivo archivo.close()
```

IMPORT Y MÓDULOS

Sirve para utilizar clases y funciones definidas en otros archivos.

Un módulo en Python es un archivo del mismo nombre con extensión .py.

Tres maneras de utilizar el comando:

```
import algun_archivo
from algun_archivo import *
from algun_archivo import className
```

¿Cuál es la diferencia?

Qué se importa del archivo y qué nombre tienen las referencias después de ser importadas.

IMPORT

```
import algun_archivo
```

Todo lo que se encuentra en archivo.py es importado.
Para referirse a los elementos importados se debe agregar el nombre del módulo antes del nombre:

```
algun_archivo.className.method("abc")
algun_archivo.myFunction(34)
```

```
from archivo import *
```

También se importa todo, pero ahora no es necesario agregar el nombre del módulo antes ya que todo se importó al espacio de nombres actual.

```
className.method("abc")
myFunction(34)
```

¡Cuidado! Esto puede redefinir funciones o clases que se llamen igual en tu programa y en el módulo.

IMPORT

```
from algun_archivo import className
```

- Solo el elemento *className* de algun_archivo.py es importado.
- Después de importar *className*, se puede utilizar sin necesidad de agregar el prefijo del módulo, ya que se trajo al espacio de nombres actual.

```
className.method("abc")      #Esto se importó

myFunction(34)               #Esta función no
```

- Podemos importar la función también:

```
from algun_archivo import className, myFunction
```

RUTAS DE BÚSQUEDA DE MÓDULOS

Cuando importamos un modulo por ejemplo spam, el intérprete inicia una búsqueda:

Primero busca si hay un built-in con ese nombre.

Si no se encuentra busca un archivo llamado spam.py en una lista de directorios dada por la variable **sys.path**.

sys.path se inicializa desde con estas localidades:

- El directorio que contiene el script de entrada (el directorio actual).
- La variable de entorno PYTHONPATH (una lista de directorios, utiliza la misma sintaxis que la variable PATH).
- Después de la inicialización, los programas de Python pueden modificar la variable sys.path.

El directorio actual es insertado al principio de las rutas, delante de la ruta de librerías estándar.