Clase_2_Variables

August 26, 2023

1 Variables más complejas (Contenedores)

En Python existen otros tipo de variables aparte de las que vimos en el capítulo anterior. Estas variables aprovechan la programación orientada a objetos y son estructuras que engloban a las variables básicas. Actúan no manejando datos individuales sino agrupaciones de ellos y se las conoce como contenedores. En palabras más simples, los contenedores se los utiliza para manejar conjuntos de elementos de datos de cualquier tipo de los que vimos en el capítulo anterior. Los elementos pueden o no estar numerados, pueden o no estar repetidos y pueden o no pueden ser inmutables. Con inmutables se indica que una vez cargados con un valor estas variables **NO** pueden ser modificadas durante el transcurso del programa. Si bien esto parece algo que no aporta y que obliga a consumir más recursos de la computadora, las variables inmutables son más "livianas" a la hora de ser usadas, dado que se trata de objetos sin mucha carga computacional. Los contenedores están pensados para que el usuario los utilice sobre grandes cantidades de datos.

En estas variables se debe prestar atención si se las está usando con paréntesis ( ), llaves { } o corchetes [ ] ya que estos son indicadores del tipo de contenedor.

Las variables guardadas en un contenedor se la considera **iterables** es decir, existen formas de recorrer con alguna operación a todos los elementos guardados en el contenedor.

No debe confundirse a los contenedores con variables aptas para álgebra vectorial, aunque se les parecen y podrían llegar a usarse, no son convenientes. Para el uso de cálculos en este campo deben utilizarse arreglos de la librería *Numpy* que serán descriptos en un capítulo siguiente.

Los principales tipos de contenedores son los siguientes.

1.1 Tipos de datos Iterables:

- Listas
- Tuplas
- Sets
- Frozensets
- Diccionarios

---

2 Listas

- Son una colección de objetos, pueden ser de tipos diferentes. Tienen orden y este orden está numerado.

- Las listas pueden mutar, su contenido es modificable.
- Las lista se definen con una colección de datos entre corchetes, separados por ",".
- Las variables que caracterizan lo datos pueden ser de tipos diferentes.

Veamos un ejemplo:

Los elementos no tienen que ser del mismo tipo.

[1]:
```
L = [1, '1', 1.4]
L
```

[1]:
```
[1, '1', 1.4]
```

Veamos otra lista:

[2]:
```
L = ['azul','verde','rojo']
```

En este caso los elementos son una colección de variables de texto Podemos preguntar que tipo de variable es L. Recordemos que para averiguar el tipo de variable usamos la orden type()

[3]:
```
type(L) # Imprime el tipo de variable que es L
```

[3]:
```
list
```

Preguntemos por uno de los elementos de la lista, pero tenemos que recordar que: * Los elementos están numerados * El primer elemento tiene número **0** y el último es el **N-1** de la lista de N elementos

[4]:
```
L[1]
```

[4]:
```
'verde'
```

Si escribo una operación o una variable sola, el notebook considera que hay una función print implícita. Básicamente se usa la celda en modo calculadora. Pero esto sólo sirve para la última variable que se encuentre, las anteriores se ignoran y no se imprimen.

[5]:
```
print(L[0])
```

```
azul
```

Pero también puedo preguntar con índices negativos, que significan que empiezo a contar por el final

[6]:
```
L[-1] # Último elemento
```

[6]:
```
'rojo'
```

[7]:
```
L[-3]
```

[7]:
```
'azul'
```

La idea de que los números negativos sean indicación de que se empiece a contar por el final tienen sentido si yo no se, ni me enteresa averiguar cuantos elementos tiene la lista. Y por otro lado necesito realizar alguna operación con los elementos del final.

Puedo sumar listas, pero no es lo que una se imagina, aunque tiene bastante sentido lo que sucede.

[8]:
```
L=L+['rojo','amarillo']
```

Y si imprimo el resultado

[9]:
```
print(L)
```

```
['azul', 'verde', 'rojo', 'rojo', 'amarillo']
```

Vean que ahora la lista tiene los elementos originales con el agregado de los dos nuevos. En una lista los elementos pueden repetirse. —

Por otro lado, hay maneras de utilizar un loop implícitito para tomar un rango de índices (inicio y fin) y un paso. Se lo construye de la siguiente manera:

Lista[Comienzo : final : paso] donde los elementos de índice i cumplen con *comienzo* <= *i* < *final* Como vimos antes el final no está incluido

Si no figura el comienzo se entiende que empieza la cuenta desde el inicio. Si no figura el final, es hasta el final de la lista. Si el paso no figura, se considera que es 1.

Un índice negativo se cuenta desde el final de la lista donde -1 es el último elemento, y un paso negativo implica que desde el final indicado voy avanzando hacia el comienzo de la lista.

"Slicing" (rebanar) es como se denomina al comando para extraer parte de una lista. Usaremos slicing para entender como se pueden filtrar una lista a través de sus índices.

[10]:
```
L[1:3]
```

[10]:
```
['verde', 'rojo']
```

[11]:
```
L[2:]
```

[11]:
```
['rojo', 'rojo', 'amarillo']
```

[12]:
```
L[-2:]
```

[12]:
```
['rojo', 'amarillo']
```

[13]:
```
L[::2] # L[start:stop:step] cada dos elementos
```

[13]:
```
['azul', 'rojo', 'amarillo']
```

[14]:
```
L[::-1]
```

[14]:
```
['amarillo', 'rojo', 'rojo', 'verde', 'azul']
```

[15]:
```
# Puedo modificar el contenido de la lista

L[2] = 'verde'
print(L)
```

```
['azul', 'verde', 'verde', 'rojo', 'amarillo']
```

Como la lista es un objeto usaremos algunas de las funciones o métodos ya programados en el objeto lista.

En este caso “append( )”, “insert( )”, “extend( )”,“count( )” y “sort( )”

Muchos de estos métodos pueden utilizarse en la forma “punto” (“dot” en inglés) en la cual aplico el método haciendo:

Lista.método() donde dentro de los ( ) puedo agregar argumentos para casos especiales.

Veamos como los puedo usar:

append( ) → agrega un nuevo dato al final y lo uso en modo “dot”.

[16]:
```
L.append('rosa') # agregar un valor al final
L
```

[16]:
```
['azul', 'verde', 'verde', 'rojo', 'amarillo', 'rosa']
```

insert( ) → Agrega un nuevo dato en la posición indicada

[17]:
```
L.insert(2, 'blue')     #  L.insert(índice, objeto) -- insertar un elemento
                        # en el lugar dado por el índice
L
```

[17]:
```
['azul', 'verde', 'blue', 'verde', 'rojo', 'amarillo', 'rosa']
```

extend( ) → agrega un iterable

[18]:
```
L.extend(['magenta', 'violeta'])
L
```

[18]:
```
['azul',
 'verde',
 'blue',
 'verde',
 'rojo',
 'amarillo',
 'rosa',
 'magenta',
 'violeta']
```

sort( ) → Ordena los elementos de la lista

[19]:
```
L2 = L.copy()   # Notar que no se hizo L2=L ¿Cuál es la diferencia entre ambas␣
 ↪operaciones?
L2.sort()
print("Lista no ordenada", L)
print("")
print("Lista ordenada alfabeticamente",L2)

# Pero si quiero ordenar la lista descendiendo
# indico que "reverse" (al revés) sea verdadero
# Desceniente significa que si son números van del más
# grande al más chico. Y si textos Van de la Z a la A.

L2.sort(reverse=True)
print("Lista ordenada descendiente", L2)
```

```
Lista no ordenada ['azul', 'verde', 'blue', 'verde', 'rojo', 'amarillo', 'rosa',
'magenta', 'violeta']

Lista ordenada alfabeticamente ['amarillo', 'azul', 'blue', 'magenta', 'rojo',
'rosa', 'verde', 'verde', 'violeta']
Lista ordenada descendiente ['violeta', 'verde', 'verde', 'rosa', 'rojo',
'magenta', 'blue', 'azul', 'amarillo']
```

count() → cuenta la cantidad de veces que un tipo particular de elemento se encuentra en la lista. Notar que la busqueda se hace indicando el elemento en cuestión como argumento de la función.

[20]:
```
print(L.count('yellow'))
```

```
0
```

[21]:
```
print(L.count('amarillo'))
```

```
1
```

Hay que recordar también que la lista puede contener cualquier variable como elemento, incluyendo otro iterable

[22]:
```
L.append(2)
L
```

[22]:
```
['azul',
 'verde',
 'blue',
 'verde',
 'rojo',
 'amarillo',
 'rosa',
 'magenta',
 'violeta',
```

```
 2]
```

[23]:
```
L = L[::-1] # Orden reverso
L
```

[23]:
```
[2,
 'violeta',
 'magenta',
 'rosa',
 'amarillo',
 'rojo',
 'verde',
 'blue',
 'verde',
 'azul']
```

[24]:
```
L2 = L[:-3] # Elimina ya que corta los últimos 3 elementos
print(L)
print(L2)
```

```
[2, 'violeta', 'magenta', 'rosa', 'amarillo', 'rojo', 'verde', 'blue', 'verde',
'azul']
[2, 'violeta', 'magenta', 'rosa', 'amarillo', 'rojo', 'verde']
```

[25]:
```
#L[25] # Fuera del rango, da error
```

[26]:
```
print(L)
print(L[20:25]) # No hay error cuando rebano (slicing).
print(L[20:])
print(L[2:20])
```

```
[2, 'violeta', 'magenta', 'rosa', 'amarillo', 'rojo', 'verde', 'blue', 'verde',
'azul']
[]
[]
['magenta', 'rosa', 'amarillo', 'rojo', 'verde', 'blue', 'verde', 'azul']
```

Como la lista es un objeto en la clase asociada que la define hay también programadas funciones o métodos. Veamos la funcion *count* que me cuenta los elementos que tengan un valor determinado. En este caso los que tengan la palabra "yellow" y luego la palabra "amarillo".

[27]:
```
print(dir(L))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
```

```
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

También puedo asignar un resultado de filtrar una lista a otra lista.

[28]:
```
L1=L[2:]
L1.sort() # Se puede usar la tecla tab para ver los métodos o funciones
          # del objeto.
          # Esta puede variar según el origen del notebook utilizado
print(L1)
```

```
['amarillo', 'azul', 'blue', 'magenta', 'rojo', 'rosa', 'verde', 'verde']
```

[29]:
```
a = [1,2,3]
b = [10,20,30]
```

[30]:
```
print(a+b) # NO es lo que uno espera (o si?), pero tiene cierta lógica
```

```
[1, 2, 3, 10, 20, 30]
```

Lo que pasó fue que la suma concatenó ambas lista, es decir, pegó **b** después de **a**.

Y que pasará si hago:

c = a*5

[31]:
```
c = a*3
print(c)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Lo que sucedió fue que la lista **a** se concatenó 3 veces, y fue equivalente a hacer $\mathbf{c} = \mathbf{a} + \mathbf{a} + \mathbf{a}$

[32]:
```
#print(a*b) # No se pueden multiplicar listas (Usar Numpy si son números
            # --> lo veremos la clase que viene)
```

2.1 Creando listas con comandos

Hay maneras en Python de crear listas en Python cuando tienen patrones simples de una manera muy rápida y eficiente, veamos una forma simple de hacerlo con la función **range( )**

[33]:
```
L = list(range(4)) # Crea una lista a partir de un definición numérica.
                   # El número es la cantidad de elementos.
                   # pero también indica que el último es el N-1
                   # porque el primer elemento es el 0.
L
```

[33]:
```
[0, 1, 2, 3]
```

Range se usa en su mínima expresión con el número final-1, en este ejemplo 4. Ya que mi valor final era "3". Esto se debe a que empiezo a contar no en "1", sino en el número "0", lo que da a lugar a muchas confusiones.

En la forma más general es **range(inicio,final-paso, paso)**

Como pueden ver en ejemplo anterior, si inicio no está se considera que es "0", y sino está indicado el paso es "1".

Y si quiero una lista de número pares:

[34]:
```
L = list(range(2, 20, 2)) # cada 2 enteros
L
```

[34]:
```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Puedo borrar elementos, en el ejemplo que sigue quito el quinto elemento, pero en general se usa el comando **del( )** para remover el n+1-avo elemento. Es el n+1 porque hay que recordar que el primer elemento es el "0"

[35]:
```
L = list(range(0,20,2))
print(L)
del L[5]
print(L)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 2, 4, 6, 8, 12, 14, 16, 18]
```

3 Listas de listas

Es algo que se puede hacer, ya que una lista puede tener elementos de diferentes tipos de variables También puede tener como elemento a otra lista.

[36]:
```
a = [[1, 2, 3], [10, 20, 30], [100, 200, 300]] # No es 2D (no es una MATRIZ,
                                                # OJO es una tabla de tablas
print(a)
```

```
[[1, 2, 3], [10, 20, 30], [100, 200, 300]]
```

[37]:
```
print(a[0])
```

```
[1, 2, 3]
```

El elemento "0" es una lista

[38]:
```
print(a[1][1])
```

```
20
```

Pero en este caso tomo la lista "1" y de ahí ahora tomo su elemento "1"

[39]:
```
#print(a[1,1]) # NO FUNCIONA, no son matrices
```

[40]:
```
b = a[1]
print(b)
```

```
[10, 20, 30]
```

Lo que hice fue tomar la lista "1" y asignar la lista entera a la variable b Y veo que puedo cambiar un sólo valor en particular de esa lista.

[41]:
```
b[1] = 999
print(b)
```

```
[10, 999, 30]
```

[42]:
```
print(a) # Cambiando b cambiamos a, Cuidado !!!
```

```
[[1, 2, 3], [10, 999, 30], [100, 200, 300]]
```

Los objetos cuando se asignan no se copian sino que se asignan con el mismo nombre a esas variables. No se crea un nuevo objeto, sino que con distintos nombres se llaman al mismo objeto o a parte de él como en este caso. Podemos preguntar con el operador "is" si es el mismo objeto.

[43]:
```
b[1] is a[1][1]
```

[43]:
```
True
```

[44]:
```
c = a[1][::] # copiar en vez de rebanar esto crea un nuevo objeto
print(c)
c[0] = 77777
print(c)
print(a)
a[1] is c
```

```
[10, 999, 30]
[77777, 999, 30]
[[1, 2, 3], [10, 999, 30], [100, 200, 300]]
```

[44]:
```
False
```

Los métodos asociados a las listas son: > >* sort( ): Ordena la lista en orden ascendente. >* type( ): Indica el tipo de Clase de un objeto. >* append( ): Agrega un elemento a la lista. >* extend( ): Agrega varios elementos a la lista. >* index( ): Retorna la primera aparición de un valor en particular. >* max( ): Retorna el item que tiene el valor máximo. >* min( ): Retorna el item que tiene el valor mínimo. >* len( ): Retorna la cantidad de elementos de una lista. >* clear( ): Borra la lista. >* insert( ): Inserta un elemento en la posición requerida. >* count( ): Retorna el número de veces que un elemento tiene el valor indicado. >* pop( ): Borra un elemento en la posición requerida y lo devuelve a una variable. >* remove( ): Borra la primera aparición de

elemento con el valor indicado. >* reverse( ): Da vuelta el orden de los elementos de una lista. >* copy( ): Duplica la lista.

4 Tuplas

Son como las listas, pero sus valores son inmutables (no se pueden modificar durante la corrida del programa)

Usan **( )** para indicar que es una tupla, a diferencia de las listas donde se usan **[ ]**

En muchas maneras las tuplas son parecidas a las listas, pero la diferencia más importante es que las tuplas pueden usarse como índices (keys) en un diccionario y como elementos de un set, mientras las listas NO PUEDEN ser usadas para estas tareas. Veremos que son diccionarios y sets a continuación.

La ventaja de las tuplas con respecto a las listas es que estas últimas tienen una gran cantidad de métodos o funciones programadas por lo cual son un objeto que necesita más recursos computacionales. Por eso se diseño a las Tuplas para que sean más "livianas" y por lo tanto también más rápidas.

Las funciones en Python que pueden devolver más de un valor retornan los resultados en forma de una Tupla.

Dos funciones que tienen las tuplas pre-programadas son count( ) y index( ), la primera cuenta la cantidad de veces que cierto elemento se encuentra en una tupla y la segunda nos da el índice de la primera aparición de un elemento en particular.

Las funciones en Python que pueden devolver más de un valor retornan los resultados en forma de una Tupla.

Ejemplos:

[45]:
```
T = (1,2,3)
print(T.count(1))  # Cuento cuantas veces aparece el número "1"

T2 = (1,2,3,4,5,2,3,4,2,2)
print(T2.count(2))  # Cuento cuantas veces aparece el número "2"
```

```
1
4
```

[46]:
```
T.index(2)  # Retorna donde está el valor buscado, en este caso un "2"
```

[46]: 1

Aunque es poco usado, si cargo varios elementos en una variable en una sola asignación, separados por comas, se presupone que la variable será una tupla:

[47]:
```
T2 = 1, 2, 3
print(T2)
type(T2)
```

```
(1, 2, 3)
```

[47]:
```
tuple
```

[48]:
```
T[1]
```

[48]:
```
2
```

[49]:
```
### Las tuplas son inmutables
```

[50]:
```
#T(1) = 3 # No funciona
```

La única manera de borrar los elementos de una tupla es volver a escribirla de nuevo entera.

[51]:
```
T=(3,2,1)  # la tengo que escribir toda de nuevo
T
```

[51]:
```
(3, 2, 1)
```

Veamos una característica muy especial de las tuplas. Creo varias tuplas de tamaños diferentes y las imprimo:

[52]:
```
t3=(1,2,3)
t2=(1,2)
t1=(22)
t0=()

# imprimo los valores

print(t3)
print(t2)
print(t1)
print(t0)
```

```
(1, 2, 3)
(1, 2)
22
()
```

¿Qué paso aquí?

Por lo que se aprecia según el resultado de la impresión son todas tuplas ya que tienen los ( ), salvo cuando imprimo la que tiene un sólo elemento. El cual aparece como un número sin los ( ) que caracterizan a la Tupla. Es decir que es un número y no una Tupla. Note que incluso la que está vacia se imprimió con ( ), indicando que aún no teniendo ningún elemento sigue siendo una tupla.

¿Porque se diseñaron las tuplas con esta característica?

La razón es para permitir que se pueda acceder rápidamente a los valores guardados para asignarlos a variables individuales en una sola sentencia.

Y esto se realiza de la siguiente forma:

var1,var2,var3,... = Tupla(val1,val2,val3,...)

donde la cantidad de variables debe ser igual a la cantidad de valores en la tupla

Ejemplo:

Tengo la tuppla t_prueba que tiene los siguientes valores

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

Si quisiera pasar esos valores a las variables a,b,c,d la manera más "clasica" sería:

```
a = t_prueba[0]
b = t_prueba[1]
c = t_prueba[2]
d = t_prueba[3]
```

Que es largo y lento, además en un caso real podría tener muchas más variables. Pero como es una tupla podría hacer:

[53]:
```
t_prueba = (1,2,3,4)

a,b,c,d = t_prueba

print(a)
print(b)
print(c)
print(d)

print("T_prueba es:",type(t_prueba)," a es:",type(a))
```

```
1
2
3
4
T_prueba es: <class 'tuple'>  a es: <class 'int'>
```

Para crear una tupla con un solo elemento debe indicarse con (elemento,). Note la ","

Como lo hacemos en el ejemplo sigue:

[54]:
```
T=(22.3,)
print(T)
```

```
(22.3,)
```

5 Sets

Los **sets** son listas sin numerar, es decir los elementos no tienen una posición númerica con la cual el programador podría elegirlos en un orden determinado para una actividad.

Se distinguen de las listas por el uso de los **{ }**.

No son muy usados, pero existen y están disponibles. Su principal ventaja es que al no estar númerados, son más simples y por lo tanto más eficientes en el momento de ser usados. Los elementos en un set no pueden estar repetidos.

Pero si tienen una contraparte matemática clara, son la implementación en Python de la teoría de conjuntos en Python ¿Recuerdan los diagrams de Venn?

Por lo cual, los elementos de set serían equivalentes a los elementos de un conjunto y el set el conjunto.

Ejemplo:

[55]:
```
Mi_set={"gato","perro","loro"}
print(Mi_set)
print(type(Mi_set))
```

```
{'gato', 'loro', 'perro'}
<class 'set'>
```

El método **add( )** de los set permite agregar nuevos elementos a un set ya existente.

[56]:
```
Mi_set.add("elefante")
print(Mi_set)
```

```
{'gato', 'elefante', 'loro', 'perro'}
```

Note que puede no haberse agregado final, ya que los set no tiene un orden establecido y este puede variar entre una computadora y otra cuando se ejecute este comando.

Y si vuelvo a agregar elementos que ya tengo:

[57]:
```
Mi_set.add("elefante")
Mi_set.add("tortuga")
Mi_set.add("gato")

print(Mi_set)
```

```
{'loro', 'gato', 'elefante', 'perro', 'tortuga'}
```

Tal como indicamos de los elementos repetidos sólo se consideró uno como válido

Los elementos de un set pueden ser removidos con los métodos “remove( )” o “discard( )”, el primero da error si elemento no está y el segundo no (y el programa continua corriendo).

También existen métodos para realizar uniones o intersecciones entre dos sets de la forma en que se usan en teoría de Conjuntos.

[58]:
```
set1 = {1,2,3,4,5,6}
set2 = {2,4,6,8,10}

set1.remove(1)
print("Removí el 1 del set, ahora me quedan=", set1)
print("")
```

```
print("La unión de ambos sets seria:",set1.union(set2))
print("La intersección de ambos sets seria:",set1.intersection(set2))
```

```
Removí el 1 del set, ahora me quedan= {2, 3, 4, 5, 6}

La unión de ambos sets seria: {2, 3, 4, 5, 6, 8, 10}
La intersección de ambos sets seria: {2, 4, 6}
```

6 Frozensets

Los **Frozen sets** o "sets congelados" son **sets** pero con la misma características que las tuplas, son inmutables. En otras palabras son un set, pero no puedo reasignar con otros valores a las variables una vez creados.

Se utiliza la función frozenset() para crearlos a partir de una lista o una tupla. a diferencia de un set donde se usan{} para indicar su naturaleza, un frozen set se anota con doble sístema de símbolos de esta manera "frozenset({elementos})"

[59]:
```
ejemplo_de_una_lista = [1,2,3,4,5]
# Convierto la lista en un frozenset

frozen_set = frozenset(ejemplo_de_una_lista)

# Y me fijo al diferencia entre la lista y el frozen set
print(ejemplo_de_una_lista)
print(type(ejemplo_de_una_lista))
print()

print(frozen_set)
print(type(frozen_set))
```

```
[1, 2, 3, 4, 5]
<class 'list'>

frozenset({1, 2, 3, 4, 5})
<class 'frozenset'>
```

También puedo crearlos a partir de una tupla

[60]:
```
ejemplo_tupla = (1,2,3,4,5)
frozen_set2= frozenset(ejemplo_tupla)

print(ejemplo_tupla)
print(type(ejemplo_tupla))
print()

print(frozen_set2)
```

```
print(type(frozen_set2))
```

```
(1, 2, 3, 4, 5)
<class 'tuple'>

frozenset({1, 2, 3, 4, 5})
<class 'frozenset'>
```

6.0.1 Resumiendo hasta ahora

Tipo	Mutabilidad	Elem. Repetidos	Elem. Numerados*	Símbolo para indentificarlos
Listas	Si	Si	Si	[ ]
Tuplas	No	Si	Si	( )
Set	Si	No	No	{ }
frozenset	No	No	No	frozenset({ })

* Indica puedo acceder a un elemento en particular a través del número que señala su posición

7 Diccionarios

Permiten organizar los datos en estructuras donde una palabra llave dispara otro dato guardado en el diccionario. Es decir relaciona una llave determinada con un valor particular. Pero este valor puede ser de cualquier tipo de variable, incluso una lista o una tupla.

Para distinguirlos de las listas y tuplas, los diccionaros utilizan "**{ }**" en su definición y uso.

[61]:
```
a_diccionario = {'uno' : 1.0,
                 'dos' : 2.0,
                 'una_lista' : ['esta', 'es', 'una', 'lista']
                 }
```

Si pregunto con "type( )" que tipo de variable es a_diccionario:

[62]:
```
print(type(a_diccionario))
```

```
<class 'dict'>
```

Me contesta que es de la clase 'dict', un diccionario.

Para acceder a los datos:

[63]:
```
print(a_diccionario['uno'])
print()
print(a_diccionario['una_lista'])
```

```
1.0

['esta', 'es', 'una', 'lista']
```

Y de la misma manera puedo modificarlos o agregar nuevos valores

[64]:
```
a_diccionario['otra_lista']=['esta','es','otra','lista']
a_diccionario['dos']= 'two'

# Y si los imprimo:
print(a_diccionario['otra_lista'])
print(a_diccionario['dos'])
```

```
['esta', 'es', 'otra', 'lista']
two
```

Otra manera de crear un diccionario es con la orden **dict( )** y entre los paréntesis escribo las llaves valores de la siguiente forma:

dicc2 = dict(llave1=‘valor1’, llave2=‘valor2’, llave2=’valor2,. . .)

Ejemplo:

[65]:
```
datos = dict(galaxia1='NGC2366', galaxia2='NGC1672', galaxia3='NGC7949',␣
 ↪galaxia4='NGC7552')

print(datos)
print("")
print(datos['galaxia3'])
```

```
{'galaxia1': 'NGC2366', 'galaxia2': 'NGC1672', 'galaxia3': 'NGC7949',
'galaxia4': 'NGC7552'}

NGC7949
```

Puedo listar todas las llaves de un diccionario en particular con la función keys(), veamos si la uso:

[66]:
```
b = a_diccionario.keys()
print(type(b))
```

```
<class 'dict_keys'>
```