Estructuras de datos en Python

Secuencias: Tuplas y Listas

Python posee además de los tipos de datos básicos, otros tipos de datos más complejos. Se trata de las **tuplas**, las **listas** y los **diccionarios**.

Estos tres tipos, pueden almacenar colecciones de datos de diversos tipos y se diferencian por su sintaxis y por la forma en la cual los datos pueden ser manipulados.
Tanto las tuplas como las listas son conjuntos ordenados de elementos.

> Una **tupla** es una tipo que permite almacenar *datos inmutables* (no pueden ser modificados una vez creados) de tipos diferentes. Las tuplas se encierran entre *paréntesis*.
>
> - Tienen longitud fija
> - Solo tienen una dimensión

> Una **lista** es similar a una tupla con la diferencia fundamental de que puede ser modificada una vez creada. Las listas se encierran sus elementos entre *corchetes*.

En la siguiente celda creamos una tupla de 4 elementos llamada `t`:

```
t = (4, "Hola", 6.0, 99)
print ("Tupla: " , t)
```

```
Tupla:  (4, 'Hola', 6.0, 99)
```

```
# Solo tenemos creada una variable, la variable t
%who
```

```
t
```

¿De que tipo es `t`?

```
type(t)       # t es de tipo tuple
```

```
tuple
```

En la siguiente celda creamos una lista de 5 elementos llamada `m`:

```
m = [ "Hola", "Mundo", 6.0, 99 , "Fin"]
m
```

```
['Hola', 'Mundo', 6.0, 99, 'Fin']
```

```
# Ahora tenemos creadas dos variables, la variable t y la variable m
%who
```

```
m        t
```

```
type(m)     # pregunto al intérprete por su tipo
```

```
list
```

Tanto si se trata de listas como de tuplas podemos:

- Comprobar si un elemento está en la secuencia con el operador **in**:

```
'Fin' in m     # ¿Está el 4 en la lista m? La respuesta es un valor de tipo bool
```

```
True
```

- Podemos saber cuántos elementos tienen con la función **len**:

```
len(m)
```

```
5
```

Acceso a los elementos de las secuencias

- Los elementos de las secuencias pueden ser accedidos mediante el uso de corchetes `[ ]`, como en otros lenguajes de programación.
- Podemos *indexar* las secuencias utilizando la sintaxis:

  `[<inicio>:<final>:<salto>]`

- En Python, la indexación empieza por CERO

```
print ( m )
print ( m[0] )  # Primer elemento de la lista está en la posición 0
```

```
['Hola', 'Mundo', 6.0, 99, 'Fin']
Hola
```

```
print ( m[1] )  # Segundo elemento
```

```
Mundo
```

```
print ( m )
print ( m[0:4] )  # Desde el primer elemento hasta el cuarto, [0,4)
                  # [0,4) son los índices 0, 1, 2, y 3
```

```
['Hola', 'Mundo', 6.0, 99, 'Fin']
['Hola', 'Mundo', 6.0, 99]
```

```
print ( m )
print ( m[:3] )  # Desde el primero hasta el tercero,  [0,3) = 0, 1 y  2
```

```
['Hola', 'Mundo', 6.0, 99, 'Fin']
['Hola', 'Mundo', 6.0]
```

```
print( m )
print( m[:] )  # Desde el primero hasta el último
print( m[::2] )  # Desde el primero hasta el último, saltando 2
```

```
['Hola', 'Mundo', 6.0, 99, 'Fin']
['Hola', 'Mundo', 6.0, 99, 'Fin']
['Hola', 6.0, 'Fin']
```

Otra forma de acceder a una secuencia es de forma inversa (de atrás hacia adelante), colocando un índice negativo.

```
print ( m )
print ( m[-1] )  # El último elemento
print ( m[-2] )  # El penúltimo elemento
```

```
['Hola', 'Mundo', 6.0, 99, 'Fin']
Fin
99
```

```
m[-2:]     # selección de los dos últimos elementos de la lista m
```

```
[99, 'Fin']
```

Los elementos de las secuencias, tanto listas como tuplas, son hetereogéneos, así que es posible definir **listas que contienen otras listas**:

```
enero =[
         [1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]
]
enero
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
enero[1] = [0,0,0]
enero
```

```
[[1, 2, 3], [0, 0, 0], [7, 8, 9]]
```

También es posible añadir nuevos valores a las listas mediante la función `append`.

```
enero.append(7)
enero
```

```
[[1, 2, 3], [0, 0, 0], [7, 8, 9], 7]
```

Podemos **concatenar** secuencias con el operador `+`:

```
n = [1,2] + [3, 4]
n
```

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

El operador multiplicación tiene un efecto un tanto particular:

```
res = n * 4
```

```
res
```

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

Las operaciones **+** y ***** sobre secuencias permiten crear nuevas secuencias sin modificar las originales.

Desempaquetar tuplas

En muchos casos es interesante asignar nombre a cada uno de los elementos de las secuencias para, posteriormente, trabajar con esos elementos.

```
laborales = (1, 2, 3, 4, 5)    # creo una tupla llamada `laborales`de 5 elementos
```

```
lunes, martes, miercoles, jueves, viernes = laborales
```

A esto se le llama **desempaquetado de secuencias**, y funciona para cualquier secuencia en el lado derecho del igual. El desempaquetado de secuencias requiere que la cantidad de variables a la izquierda del signo igual coincida con el tamaño de la secuencia.

Una vez hecho el desempaquetado, podemos acceder a los elementos de la secuencia con un nombre en lugar de por su posición dentro de la secuencia.

```
martes
```

```
2
```

```
viernes
```

```
5
```

El caso particular de las listas

Como hemos dicho anteriormente, las listas pueden tener longitud variable y son mutables. También hemos visto que pueden definirse mediante `[ ]`.

Otra forma de definir las listas es mediante la función **list**.

```
t = 3, 4, 5                 # definimos una tupla  de tres elementos
m = list(t)                 # la función 'list' transforma una tupla en una lista
m
```

```
[3, 4, 5]
```

Podemos modificar el primer elemento de la lista. Asignamos el valor Nulo en Python en la posición 0.

```
m[0] = None     # valor nulo en Python
m
```

```
[None, 4, 5]
```

Añadiendo y eliminando elementos de una lista

La forma más eficiente de añadir elementos a una lista es mediante la función **append**, que añade elementos al final de la lista.

Otra forma de añadir elementos es mediante la función **insert**, que inserta un elemento en una determinada posición.

```
m = ['Lunes', 'Jueves']
m.append('Viernes')
m
```

```
['Lunes', 'Jueves', 'Viernes']
```

```
m.insert(1, 'Martes')    # inserta el elemento 'martes' en la posición 1
m                        # desplaza las posiciones a la derecha del 1
```

```
['Lunes', 'Martes', 'Jueves', 'Viernes']
```

Si queremos añadir múltiples elementos a una lista, podemos utilizar el operador + o el método **extends**. Si la lista que estamos construyendo es muy larga es preferible utilizar **extends** ya que es mucho mas eficiente. La razón es que el operador + debe crear una nueva lista y copiar todos los elementos en la nueva lista.

Veamos un ejemplo de su uso:

```
m.extend([1,2,3,4,5,6,7,8,9,10])
m
```

```
['Lunes', 'Martes', 'Jueves', 'Viernes', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Para eliminar elementos de una lista, podemos utilizar las funciones **pop** y **remove**.

Eliminando por posición

La operación **pop** permite eliminar el elemento de la lista que ocupa una determinada posición.

```
e = m.pop(1)
e
```

```
'Martes'
```

```
m        # la lista tiene un elemento menos
```

```
['Lunes', 'Jueves', 'Viernes', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Eliminación por valor

Puede darse el caso de que necesitemos eliminar un elemento de la lista y que no conozcamos la posición que ocupa. En esos casos utilizaremos el método **remove**.

```
m.remove('Jueves')
m
```

```
['Lunes', 'Viernes', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Ordenando una lista

El método **sort** permite ordenar una lista sin necesidad de crear una lista nueva, por lo que la operación es muy eficiente.

```
lista = [5,7,2,0,4,7,1,5,4,3,4,1,9,0]
lista.sort()
lista
```

```
[0, 0, 1, 1, 2, 3, 4, 4, 4, 5, 5, 7, 7, 9]
```

Generando listas

Python 2.7 proporciona la función predefinida **range(inicio, fin, paso)** para generar listas automáticamente. En Python 3.6 no se genera una lista; se genera un objeto iterable. Decimos que un objeto es **iterable** si se puede recorrer para recuperar cada uno de los elementos que contiene. En este sentido, las listas son iterables y las tuplas también.

- **range** no construye una lista por lo que se ahorra espacio.
- devuelve cada uno de los elementos cuando se recorre.

Veamos algunos ejemplos:

```
# Python 3.6
range(10)         # crea un objeto iterable de 10 números enteros: [0, 10)
```

```
range(0, 10)
```

```
# Python 3.6
r = range(10)     # usamos la función 'list' que me devuelve una lista
list(r)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
#  crea un objeto iterable de números enteros entre el -5 y 5: [-5, 5)
r = range(-5, 5)
list(r)
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

También es posible crear una lista a partir de otra:

- Recordar que la función **sorted** recibe una lista 'm' y devuelve una nueva lista ordenada a partir de 'm'.
- Otra función es la función **reversed** que devuelve un iterador en orden inverso de la lista original.

```
i = reversed(r)
i
```

```
<range_iterator at 0x1756e3f9d30>
```

```
list(i)
```

```
[4, 3, 2, 1, 0, -1, -2, -3, -4, -5]
```

Un caso particular de secuencias: Las secuencias de caracteres- str

Las cadenas son consideradas como una secuencia de caracteres y por tanto pueden ser tratadas como otras secuencias (tuplas o listas).

Por ejemplo, podemos acceder a cada uno de los caracteres de una cadena:

```
a = "Ana"
a, a[0], a[2]
```

```
('Ana', 'A', 'a')
```

Las cadenas en Python son inmutables. Eso quiere decir que no es posible modificar una cadena sin crear otra nueva.

```
# En Python 3.5 strings son Unicode por defecto
mensaje = "Vaya calor que hace"
mensaje[0]
```

```
'V'
```

```
b = mensaje.replace('V', 'v')
b, mensaje
```

```
('vaya calor que hace', 'Vaya calor que hace')
```

Para ver todas las funciones que podemos utilizar con cadenas basta con ejecutar la siguiente celda:

```
dir(str)
```

```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',

 'upper',
```

```
 'zfill']
```

Algunos ejemplos:

```
mensaje.upper()
```

```
'VAYA CALOR QUE HACE'
```

```
mensaje.title()
```

```
'Vaya Calor Que Hace'
```