Características básicas del lenguaje

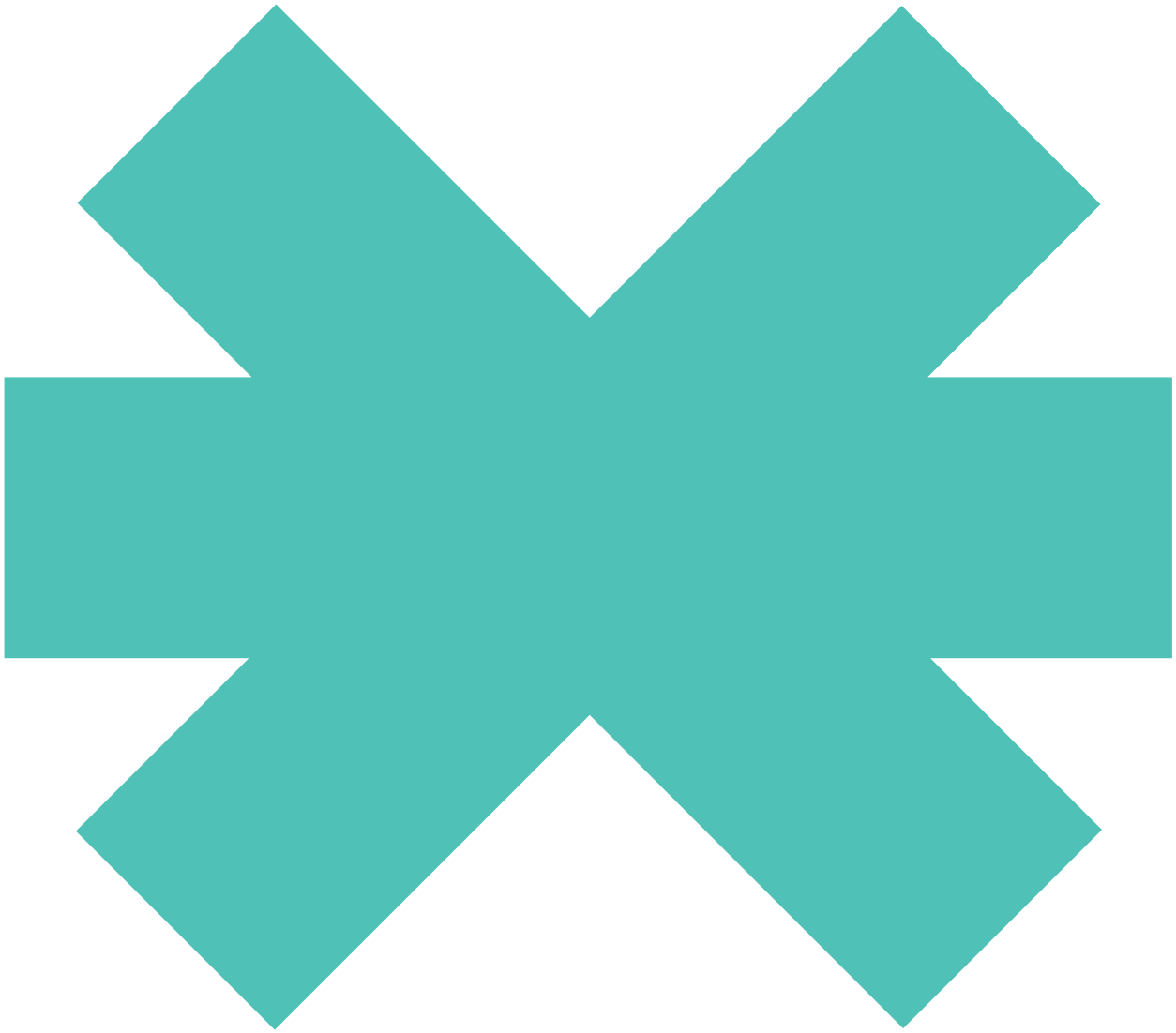
Programación Python

4

Tipos de datos avanzados

New Technology School

Tokio.



4 Tipos de datos avanzados

Sumario

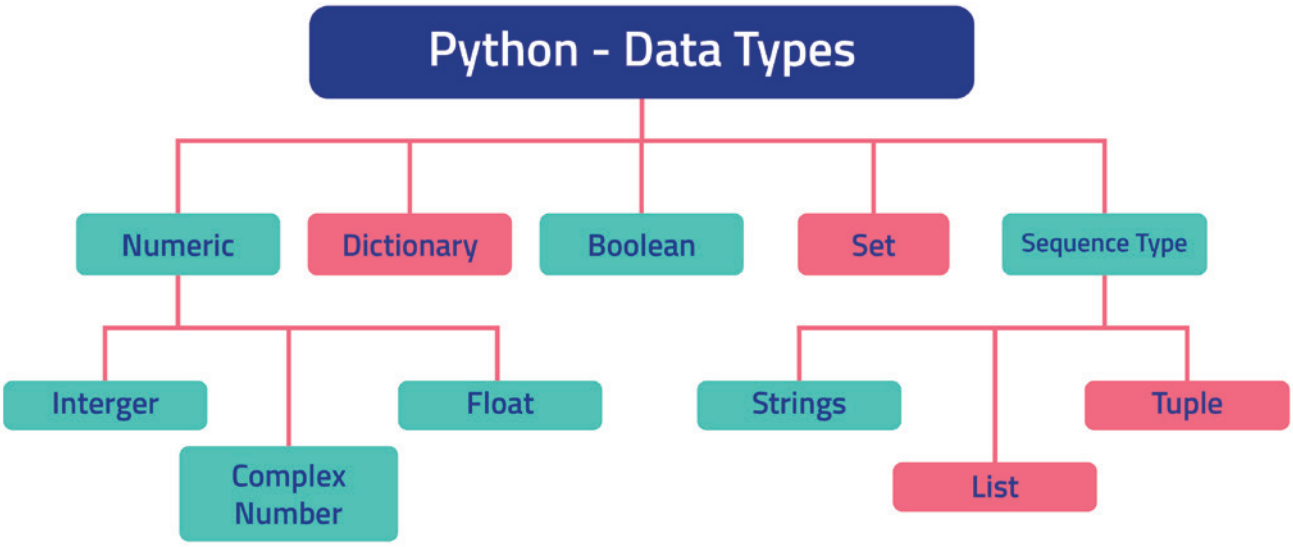

Python - Data Types
Numeric
Dictionary
Boolean
Set
Sequence Type
Interger
Float
Complex Number
Strings
Tuple
List

Figura 5

4.1 Listas

Una **lista** en *Python* es una estructura de datos formada por una secuencia ordenada de objetos, es una colección ordenada y modificable que permite miembros duplicados.

Las listas en *Python* son heterogéneas porque pueden estar formadas por elementos de distintos tipos, incluidos otras listas y mutables, porque sus elementos pueden modificarse.

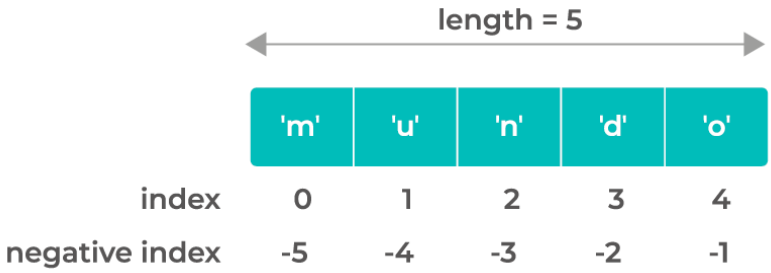


Figura 6

A través de los índices de una lista podemos cambiar el valor de sus elementos, haciendo de esta forma la lista mutable.

Los valores que componen una lista van ordenados entre [ ] y separados por comas. A continuación, se muestra como declarar una lista, como mostrarla y comprobar de qué tipo es el elemento creado:

```
In [3]: numeros = [1,2,3,4]
        print(numeros)
        [1, 2, 3, 4]

In [4]: datos = [4,"Una cadena",-15,3.14,"Otra cadena"]
        print(datos)
        [4, 'Una cadena', -15, 3.14, 'Otra cadena']

In [3]: print(type(datos))
        <class 'list'>
```

Tanto el acceso a los elementos, que se realiza mediante índices, como el *slicing*, se realizan de forma muy similar a las cadenas de caracteres, pudiendo también acceder a los elementos desde el final con índices negativos, como hemos visto con las cadenas:

```
In [5]: print(datos)
        print(datos[0])
        [4, 'Una cadena', -15, 3.14, 'Otra cadena']
        4

In [7]: print(datos[-1])
        Otra cadena

In [8]: print(datos[-2:])
        [3.14, 'Otra cadena']

In [9]: print(datos[1:3])
        ['Una cadena', -15]
```

Las listas también aceptan el operador de suma, cuyo resultado es una nueva lista que incluye todos los ítems:

```
In [11]: numeros = numeros + [5,6,7,8]
         print(numeros)
```

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

Las listas, como bien indicamos anteriormente, son modificables, y para ello modificamos el valor a través de sus índices:

```
In [6]: pares = [0,2,4,5,8,10]
        print(pares)
```

```
[0, 2, 4, 5, 8, 10]
```

```
In [7]: pares[3]= 6
        print(pares)
```

```
[0, 2, 4, 6, 8, 10]
```

Asimismo, integran funcionalidades internas, como el método *.append()* usado para añadir un ítem al final de la lista:

```
In [9]: pares.append(12)
```

```
In [10]: print(pares)
```

```
[0, 2, 4, 6, 8, 10, 12]
```

```
In [11]: pares.insert(0, 7)
```

```
In [17]: pares.append(7*2)
```

```
In [12]: print(pares)
```

```
[7, 0, 2, 4, 6, 8, 10, 12]
```

Y tienen una peculiaridad, que aceptan asignación con *slicing* para modificar varios ítems en conjunto:

```
In [20]: letras = ['a','b','c','d','e','f']
```

```
In [21]: print(letras[:3])
```

```
['a', 'b', 'c']
```

```
In [22]: letras[:3] = ['A','B','C']
```

```
In [23]: print(letras)
```

```
['A', 'B', 'C', 'd', 'e', 'f']
```

Si queremos borrar el contenido de una lista solo tenemos que asignar una lista vacía a la misma:

```
In [26]: letras[:3] = []
```

```
In [27]: print(letras)
```

```
['d', 'e', 'f']
```

```
In [28]: letras = []
```

```
In [29]: print(letras)
```

```
[]
```

Al igual que sucede con las cadenas la función *len*() devuelve la cantidad de elementos que contiene la lista o lo que es lo mismo, su longitud:

```
In [30]: print(len(letras))
         0

In [31]: print(len(pares))
         6
```

Para buscar un elemento dentro de una lista usaremos el operador *in*, que nos devolverá verdadero o falso en función de si encuentra el elemento dentro de la lista o no:

```
In [40]: print(pares)
         [0, 2, 4, 6, 8, 10]

In [41]: 2 in pares
Out[41]: True

In [42]: 7 in pares
Out[42]: False
```

Podemos incluir listas dentro de listas, llamadas listas anidadas, y las manipularemos fácilmente utilizando múltiples índices, como si nos refiriéramos a las filas y columnas de una tabla:

```
In [13]: a = [1,2,3]
         b = [4,5,6]
         c = [7,8,9]
         r = [a,b,c]

In [33]: print(r)
         [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

In [34]: print(r[0])  # Primera sublista
         [1, 2, 3]

In [35]: print(r[-1])  # Última sublista
         [7, 8, 9]

In [14]: print(r[0][1])  # Primera sublista, y de ella, primer ítem
         2

In [37]: print(r[1][1])  # Segunda sublista, y de ella, segundo ítem
         5

In [38]: print(r[2][2])  # Tercera sublista, y de ella, tercer ítem
         9

In [39]: print(r[-1][-1])  # Última sublista, y de ella, último ítem
         9
```

4.2 Tuplas

Una **tupla** en *Python* es una estructura de datos formada por una secuencia ordenada de objetos. Una colección ordenada e inmutable que permite miembros duplicados.

Podemos decir que las tuplas son listas inmutables, que no pueden modificarse después de su creación.

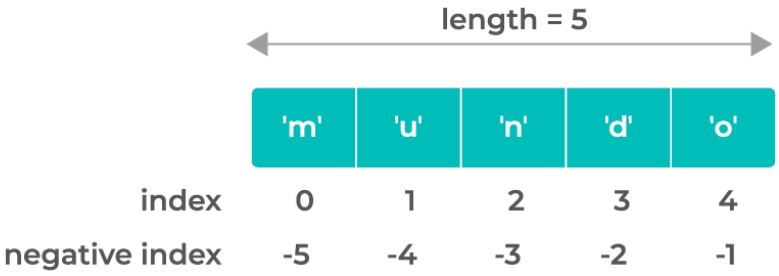


Figura 7

Con las tuplas se trabaja exactamente igual que con las listas. La única diferencia es que las tuplas son inmutables, no se puede modificar su contenido. Los valores que componen una tupla van ordenados entre ( ) y separados por comas. A continuación, se muestra cómo declarar una tupla, cómo mostrarla y comprobar de qué tipo es el elemento creado:

```
In [2]: numeros = (1,2,3,4)
        print(numeros)
        (1, 2, 3, 4)

In [3]: datos = (4,"Una cadena",-15,3.14,"Otra cadena")
        print(datos)
        (4, 'Una cadena', -15, 3.14, 'Otra cadena')

In [4]: print(type(datos))
        <class 'tuple'>
```

Las tuplas, en cuanto a índices y *slicing*, funcionan de una forma muy similar a las cadenas de caracteres y las listas:

```
In [5]: print(datos[0])
        4

In [6]: print(datos[-1])
        Otra cadena

In [7]: print(datos[-2:])
        (3.14, 'Otra cadena')

In [8]: print(datos[1:3])
        ('Una cadena', -15)
```

También podemos realizar la suma de tuplas, que nos dará como resultado una nueva tupla que incluye todos los ítems:

```
In [10]: numeros = numeros + (5,6,7,8)
         print(numeros)

         (1, 2, 3, 4, 5, 6, 7, 8)
```

Es muy importante tener en cuenta que los elementos de las tuplas no son modificables, cualquier intento de modificación de los mismos nos dará error:

```
In [11]: pares = (0,2,4,5,8,10)
         print(pares)

         (0, 2, 4, 5, 8, 10)

In [12]: pares[3]= 6
         print(pares)

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-12-e23cd326fffe> in <module>
         ----> 1 pares[3]= 6
               2 print(pares)

         TypeError: 'tuple' object does not support item assignment
```

Al no ser modificables no incluyen el método *append*:

```
In [13]: pares.append(12)

         ---------------------------------------------------------------------------
         AttributeError                            Traceback (most recent call last)
         <ipython-input-13-441fff7e2acb> in <module>
         ----> 1 pares.append(12)

         AttributeError: 'tuple' object has no attribute 'append'
```

Y tampoco aceptan la asignación con *slicing* para modificar varios ítems en conjunto:

```
In [22]: letras = ('a','b','c','d','e','f')

In [19]: print(letras[:3])

         ('a', 'b', 'c')

In [20]: letras[:3] = ('A','B','C')

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-20-322fe606ae62> in <module>
         ----> 1 letras[:3] = ('A','B','C')

         TypeError: 'tuple' object does not support item assignment
```

La función *len*() también funciona con las tuplas del mismo modo que en las listas y cadenas de caracteres:

```
In [23]: print(len(letras))

         6
```

Al igual que en las listas, podemos buscar con el operador *in*:

```
In [25]: print(pares)
         (0, 2, 4, 5, 8, 10)

In [26]: 2 in pares
Out[26]: True

In [27]: 7 in pares
Out[27]: False
```

Y también podemos manipular fácilmente tuplas anidadas utilizando múltiples índices, como si nos refiriéramos a las filas y columnas de una tabla:

```
In [28]: a = (1,2,3)
         b = (4,5,6)
         c = (7,8,9)
         r = (a,b,c)

In [29]: print(r)
         ((1, 2, 3), (4, 5, 6), (7, 8, 9))

In [30]: print(r[0])  # Primera subtupla
         (1, 2, 3)

In [35]: print(r[-1])  # Última subtupla
         [7, 8, 9]

In [36]: print(r[0][0])  # Primera subtupla, y de ella, primer ítem
         1

In [37]: print(r[1][1])  # Segunda subtupla, y de ella, segundo ítem
         5

In [38]: print(r[2][2])  # Tercera subtupla, y de ella, tercer ítem
         9

In [31]: print(r[-1][-1])  # Última subtupla, y de ella, último ítem
         9
```

4.3 Sets

Un **conjunto o *set***, es una colección desordenada y no indexada en la que no se permiten elementos repetidos. Los usos básicos de estos conjuntos incluyen verificación de pertenencia y eliminación de entradas duplicadas.

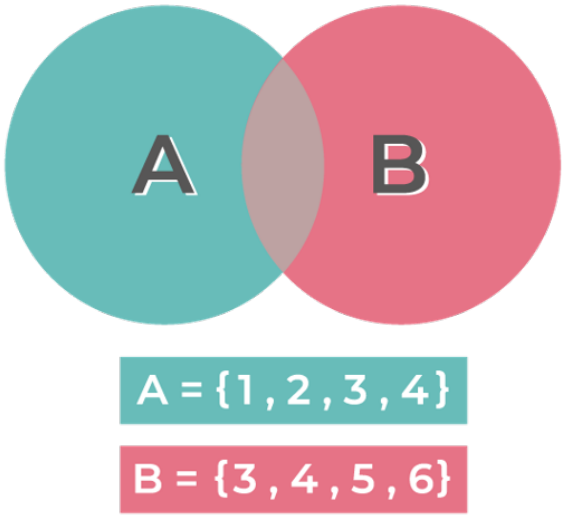


Figura 8

Los valores que componen un conjunto van indicados entre { } y separados por comas. A continuación, se muestra como declarar un conjunto, como mostrarlo y comprobar de qué tipo es el elemento creado:

```
In [1]: numeros = {1,2,3,4}
        print(numeros)
```

```
{1, 2, 3, 4}
```

```
In [5]: datos = {4,"Una cadena",-15,3.14,"Otra cadena"}
        print(datos)
```

```
{3.14, 4, 'Otra cadena', -15, 'Una cadena'}
```

```
In [2]: print(type(numeros))
```

```
<class 'set'>
```

```
In [6]: print(type(datos))
```

```
<class 'set'>
```

No podemos acceder a los elementos de un **conjunto *set*** haciendo referencia a un índice, porque los conjuntos no están ordenados y debido a esto, los elementos no tienen índice. Pero podemos recorrer los elementos del conjunto utilizando un bucle *for*, que veremos más adelante, o preguntar si un valor especificado está presente en un conjunto, utilizando la palabra clave *in*:

```
In [4]: lenguajes = {"Python", "C++", "Java"}
        for x in lenguajes:
            print(x)
```

```
Java
Python
C++
```

```
In [8]: print("Python" in lenguajes)
```

```
True
```

Una vez que se crea un conjunto set, no podemos cambiar sus elementos, pero podemos agregar nuevos elementos mediante el método *add()*, para agregar un elemento a un conjunto o mediante el método *update()*, para agregar más de un elemento a un conjunto. Podemos observar como el orden es totalmente aleatorio y decidido por el lenguaje, y como, al no aceptar elementos repetidos, si añadimos de nuevo un elemento que ya existe, no se añadirá como un elemento nuevo:

```
In [32]: lenguajes = {"Python", "C++", "Java"}
         print(lenguajes)

         {'Java', 'C++', 'Python'}

In [33]: lenguajes.add("C#")
         print(lenguajes) # Como se puede ver, el orden es totalmente aleatorio y decidido por el lenguaje

         {'Java', 'C#', 'C++', 'Python'}
```

```
In [37]: lenguajes.add("C#")
         print(lenguajes) # Como se puede ver, el orden es totalmente aleatorio y decidido por el lenguaje

         {'Java', 'C#', 'C++', 'Python'}

In [16]: lenguajes.update(["Go", "Javascript", "PHP"])
         print(lenguajes)

         {'C#', 'Python', 'Go', 'C++', 'Javascript', 'Java', 'PHP'}
```

La función *len*() también funciona para los conjuntos:

```
In [17]: print(len(lenguajes))

         7
```

Para eliminar elementos del conjunto podemos utilizar dos métodos, *discard()* o *remove()*, indicando entre paréntesis el elemento que queremos eliminar:

```
In [21]: lenguajes = {'C#', 'Python', 'Go', 'C++', 'Javascript', 'Java', 'PHP'}
         print(lenguajes)

         lenguajes.remove("Go")
         print(lenguajes)

         lenguajes.discard("PHP")
         print(lenguajes)

         {'C#', 'Python', 'Go', 'C++', 'Java', 'Javascript', 'PHP'}
         {'C#', 'Python', 'C++', 'Java', 'Javascript', 'PHP'}
         {'C#', 'Python', 'C++', 'Java', 'Javascript'}
```

Si queremos buscar dentro del conjunto lo haremos con el operador *in*:

```
In [22]: print(lenguajes)

         {'C#', 'Python', 'C++', 'Java', 'Javascript'}

In [24]: "C#" in lenguajes

Out[24]: True
```

Y si lo que queremos es eliminar todo el contenido de un conjunto, debemos utilizar el método *clear()*:

```
In [27]: lenguajes = {'C#', 'Python', 'Go', 'C++', 'Javascript', 'Java', 'PHP'}
         print(lenguajes)
         lenguajes.clear()
         print(lenguajes)

         {'C#', 'Python', 'Go', 'C++', 'Java', 'Javascript', 'PHP'}
         set()
```

Por último, debemos saber que los conjuntos no son anidables, de manera que no puede haber conjuntos dentro de otros conjuntos:

```
In [31]: a = {1,2,3}
         b = {4,5,6}
         c = {7,8,9}
         r = {a,b,c}

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)
         <ipython-input-31-c6b53e85afab> in <module>
               2 b = {4,5,6}
               3 c = {7,8,9}
         ----> 4 r = {a,b,c}

         TypeError: unhashable type: 'set'
```

4.4 Diccionarios

Un **diccionario** en *Python* es una colección desordenada, modificable e indexada que no permite miembros duplicados. Un diccionario define una relación uno a uno entre claves y valores.

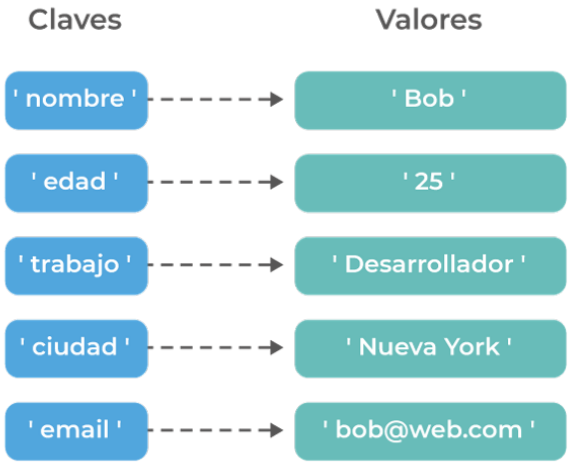


Figura 9

Los valores que componen un diccionario van encerrados entre {} y separados con comas. La estructura principal es *clave* : *valor*. A continuación, se muestra cómo declarar un diccionario, cómo mostrarlo y comprobar de qué tipo es el elemento creado:

```
In [1]: vehiculos = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964
        }
        print(vehiculos)

        {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

In [43]: print(type(vehiculos))

         <class 'dict'>
```

Para acceder a los elementos de un diccionario tenemos dos formas, bien haciendo referencia a su *clave*, o bien utilizando el método *get()*. Ambas formas nos devolverán el valor correspondiente, como se ve a continuación:

```
In [2]: valorQueMeInteresa = vehiculos["model"]
        print(valorQueMeInteresa)

        Mustang

In [49]: valorQueMeInteresa = vehiculos.get("model")
         print(valorQueMeInteresa)

         Mustang
```

Para buscar una clave en un diccionario se utiliza el operador *in* (solo sirve para buscar claves):

```
print("model" in vehiculos)
```

```
True
```

Para modificar un valor haremos referencia a su clave:

```
In [1]: vehiculos = {
            "brand": "Ford",
            "model": "Mustang",
            "year": 1964
        }
        print(vehiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

```
In [3]: vehiculos["year"] = 2020
        print(vehiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Para saber la longitud de un diccionario usaremos la función *len()*, que también funciona con los diccionarios:

```
In [4]: print(len(vehiculos))
```

```
3
```

Para recorrer un diccionario utilizaremos el mismo método que con los conjuntos:

```
for x in vehiculos:
    print(x)
```

```
brand
model
year
```

```
for x in vehiculos:
    print(vehiculos[x])
```

```
Ford
Mustang
2020
```

```
for x in vehiculos:
    print(x, ": ", vehiculos[x])
```

```
brand :  Ford
model :  Mustang
year :  2020
```

```
for x in vehiculos.values():
    print(x)
```

```
Ford
Mustang
2020
```

Existe un método de los diccionarios que nos facilita la lectura en clave y el valor de los elementos, porque devuelve ambos valores en cada iteración automáticamente:

```
In [10]: for x, y in vehiculos.items():
             print(x, ":", y)
```

```
brand : Ford
model : Mustang
year : 2020
```

Para agregar elementos a un diccionario utilizaremos una nueva clave de índice y le asignaremos un valor:

In [4]:
```
vehiculos = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(vehiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

In [5]:
```
vehiculos["color"] = "red"
print(vehiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Para eliminar elementos del diccionario utilizaremos uno de estos tres métodos, según nos sea más conveniente: *clear*() para borrar todo el diccionario, *pop*() para eliminar el elemento con el nombre de clave especificado y *popitem*() para eliminar el último elemento insertado. Hay que tener en cuenta que *popiten()*, en versiones anteriores a la 3.7 del intérprete, elimina un elemento aleatorio del diccionario en lugar de eliminar el último elemento insertado:

In [19]:
```
vehiculos = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(vehiculos)

# Elimina el último elemento insertado
# (en versiones anteriores a 3.7, en su lugar, se elimina un elemento aleatorio):
eliminado = vehiculos.popitem()
print(vehiculos)
print("Se ha eliminado la siguiente pareja de datos:", eliminado)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
{'brand': 'Ford', 'model': 'Mustang'}
Se ha eliminado la siguiente pareja de datos: ('year', 1964)
```

In [20]:
```
vehiculos = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(vehiculos)

eliminado = vehiculos.pop("model") # Elimina el elemento con el nombre de clave especificado
print(vehiculos)
print("Se ha eliminado el valor", eliminado)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
{'brand': 'Ford', 'year': 1964}
Se ha eliminado el valor Mustang
```

In [21]:
```
vehiculos = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(vehiculos)

vehiculos.clear() # Vacia el diccionario
print(vehiculos)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
{}
```

Un diccionario no se puede copiar realizando una asignación entre dos diccionarios de la forma dict2 = dict1, porque el diccionario dict2 solo será una referencia a dict1, y los cambios realizados en dict1 también se realizarán automáticamente en dict2. Para hacer una copia hay que utilizar el método *copy()*:

```
In [18]: vehiculos = {
           "brand": "Ford",
           "model": "Mustang",
           "year": 1964
         }
         print(vehiculos)

         vehiculos_copia = vehiculos.copy()
         print(vehiculos_copia)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

```
In [19]: vehiculos.pop("model") # Elimina el elemento con el nombre de clave especificado
         print(vehiculos)
         print(vehiculos_copia) # Comprobamos que la copia sigue intacta y no se ha modificado con el original
```

```
{'brand': 'Ford', 'year': 1964}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Se pueden anidar diccionarios, incluir diccionarios dentro de otros diccionarios. En la siguiente imagen el diccionario familia contiene otros tres diccionarios:

```
In [2]: familia = {
          "child1" : {
            "name" : "Emil",
            "year" : 2004
          },
          "child2" : {
            "name" : "Tobias",
            "year" : 2007
          },
          "child3" : {
            "name" : "Linus",
            "year" : 2011
          }
        }

        print(familia)
```

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus',
'year': 2011}}
```

Y para acceder a los diccionarios "internos" lo haremos de la siguiente manera:

```
In [3]: valorQueMeInteresa = familia["child1"]
        print(valorQueMeInteresa)
```

```
{'name': 'Emil', 'year': 2004}
```

```
In [4]: valorQueMeInteresa = familia["child3"]["name"]
        print(valorQueMeInteresa)
```

```
Linus
```

Por último, en estas dos imágenes vemos como incluir listas en nuestros diccionarios:

In [3]:
```
diccionario = {
    'nombre' : 'Carlos',
    'edad' : 22,
    'cursos': ['Python','Django','JavaScript']
}

print("Diccionario completo: ")
print(diccionario)

print("\nElementos del diccionario")
print(diccionario['nombre']) # Carlos
print(diccionario['edad']) # 22
print(diccionario['cursos']) #['Python','Django','JavaScript']

print("\nItems de la lista cursos: ")
print(diccionario['cursos'][0]) #Python
print(diccionario['cursos'][1]) #Django
print(diccionario['cursos'][2]) #JavaScript

print("\nRecorriendo el diccionario con un bucle for")
for key in diccionario:
    print(key, ":", diccionario[key])
```

```
Diccionario completo:
{'nombre': 'Carlos', 'edad': 22, 'cursos': ['Python', 'Django', 'JavaScript']}

Elementos del diccionario
Carlos
22
['Python', 'Django', 'JavaScript']

Items de la lista cursos:
Python
Django
JavaScript

Recorriendo el diccionario con un bucle for
nombre : Carlos
edad : 22
cursos : ['Python', 'Django', 'JavaScript']
```

In [18]:
```
clientes = {
    'nombre' : ['Carlos','Cristian','David'] ,
    'edad' : [22,30,32] ,
    'lenguaje_favorito': ['Python','Django','JavaScript']
}

print("Diccionario completo: ")
print(diccionario)

print("\nMostrar todos los datos del primer cliente")
print(clientes['nombre'][0])
print(clientes['edad'][0])
print(clientes['lenguaje_favorito'][0])

print("\nMostrar todos los datos del segundo cliente")
print(clientes['nombre'][1])
print(clientes['edad'][1])
print(clientes['lenguaje_favorito'][1])

print("\nMostrar todos los datos del tercer cliente")
print(clientes['nombre'][2], end=", ")
print(clientes['edad'][2], end=", ")
print(clientes['lenguaje_favorito'][2])
```

```
Diccionario completo:
{'nombre': 'Carlos', 'edad': 22, 'cursos': ['Python', 'Django', 'JavaScript']}

Mostrar todos los datos del primer cliente
Carlos
22
Python

Mostrar todos los datos del segundo cliente
Cristian
30
Django

Mostrar todos los datos del tercer cliente
David, 32, JavaScript
```