

3.11. Tipo diccionarios

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

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

Un objeto *mapping* mapea valores *hashable* a objetos arbitrariamente. Los objetos Mapeos son objetos mutable. El **diccionario** es el único tipo de mapeo estándar actual. Para otros contenedores ver los integrados en las clases «*lista*», «*conjuntos*», y «*tupla*», y el módulo «*collections*».

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

Usted puede acceder a los valores del diccionario usando cada su clave, se presenta unos ejemplos a continuación:

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

Un diccionario puede almacenar los diversos tipos de datos integrados en Python usando la función `type()`, usted puede pasar el diccionario con la clave que usted desea determinar el tipo de dato, se presenta unos ejemplos a continuación:

 v: 3.7 ▼

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

3.11.1. Operaciones

Los objetos de tipo **diccionario** permite una serie de operaciones usando operadores integrados en el interprete Python para su tratamiento, a continuación algunos de estos:

3.11.1.1. Acceder a valor de clave

Esta operación le permite acceder a un valor específico del *diccionario* mediante su clave.

```
>>> versiones = {"python": 3.11, "zope": 5.2, "plone": 6.0, "django": 4.2.7}
>>> versiones["zope"]
5.2
```

3.11.1.2. Asignar valor a clave

Esta operación le permite asignar el valor específico del *diccionario* mediante su clave.

```
>>> versiones = {"python": 3.11, "zope": 5.2, "plone": None}
>>> versiones["plone"]
>>> versiones["plone"] = 6.0
>>> versiones
{'python': 3.11, 'zope': 5.2, 'plone': 6.0}
>>> versiones["plone"]
6.0
```

3.11.1.3. Iteración 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=3.11, zope=5.2, plone=6.0, django=4.2.7)
>>> print(versiones)
{'zope': 5.2, 'python': 3.11, 'plone': 6.0, 'django': 4.2.7}
```

```
>>> "plone" in versiones
True
>>> "flask" in versiones
False
```

 v: 3.7 ▼

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

3.11.2. Métodos

Los objetos de tipo **diccionario** integra una serie de métodos integrados a continuación:

3.11.2.1. clear()

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

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> print(versiones)
{'zope': 5.2, 'python': 3.11, 'plone': 6.0}
>>> versiones.clear()
>>> print(versiones)
{}
```

3.11.2.2. copy()

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

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> otro_versiones = versiones.copy()
>>> versiones == otro_versiones
True
```

3.11.2.3. 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`, más puede inicializar un *valor* común por defecto para cada *clave*:

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

 v: 3.7 ▼

3.11.2.4. 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=3.11, zope=5.2, plone=6.0)
>>> versiones.get("plone")
6.0
>>> versiones.get("php")
>>>
```

3.11.2.5. has_key()

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

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> versiones.has_key("plone")
True
>>> versiones.has_key("django")
False
```

3.11.2.6. items()

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

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> versiones.items()
dict_items([('python', 3.11), ('zope', 5.2), ('plone', 6.0)])
>>> for clave, valor in versiones.items():
...     print(clave, valor)
...
python 3.11
zope 5.2
plone 6.0
```

3.11.2.7. 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=3.11, zope=5.2, plone=6.0)
>>> print(versiones)
{'python': 3.11, 'zope': 5.2, 'plone': 6.0}
>>> versiones.keys()
dict_keys(['python', 'zope', 'plone'])
>>> for clave in versiones.keys():
...     print(clave)
...
zope
python
plone
>>> versiones_iter = iter(versiones)
>>> print(next(versiones_iter))
python
>>> print(next(versiones_iter))
zope
>>> print(next(versiones_iter))
plone
>>> print(next(versiones_iter))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

 v: 3.7 ▼

3.11.2.8. keys()

Este método devuelve una **lista** de las claves del **diccionario**:

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> versiones.keys()
dict_keys(['python', 'zope', 'plone'])
>>> for clave in versiones.keys():
...     print(clave)
...
python
zope
plone
```

3.11.2.9. 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=3.11, zope=5.2, plone=6.0)
>>> versiones
{'zope': 5.2, 'python': 3.11, 'plone': 6.0}
>>> versiones.pop("zope")
5.2
>>> versiones
{'python': 3.11, 'plone': 6.0}
>>> versiones.pop("django")
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
KeyError: 'django'
```

 v: 3.7 ▼

3.11.2.10. popitem()

Este método remueve y devuelve algún par (clave, valor) del **diccionario** como una 2 **tuplas**. Lanza una excepción **KeyError** si el **diccionario** esta vacío.

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> versiones
{'zope': 5.2, 'python': 3.11, 'plone': 6.0}
>>> versiones.popitem()
('zope', 5.2)
>>> versiones
{'python': 3.11, 'plone': 6.0}
>>> versiones.popitem()
('python', 3.11)
>>> versiones
{'plone': 6.0}
>>> versiones.popitem()
('plone', 6.0)
>>> versiones
{}
>>> versiones.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

3.11.2.11. setdefault()

Este método es similar a **get(key, default_value)**, pero además asigna la clave **key** al valor por **default_value** para la clave si esta no se encuentra en el **diccionario**.

```
D.setdefault(key[, default_value])
```

A continuación un ejemplo de como trabaja el método **setdefault()** cuando la clave esta en el diccionario:

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> zope = versiones.setdefault("zope")
>>> print("Versiones instaladas:", versiones)
Versiones instaladas: {'zope': 5.2, 'python': 3.11, 'plone': 6.0}
>>> print("Versión de Zope:", zope)
Versión de Zope: 5.2
```

A continuación un ejemplo de como trabaja el método **setdefault()** la clave no esta en el diccionario:

```
>>> paquetes = {"python": 3.11, "zope": 5.2}
>>> print(paquetes)
{'python': 3.11, 'zope': 5.2}
>>> plone = paquetes.setdefault("plone")
>>> print("paquetes: ", paquetes)
paquetes: {'python': 3.11, 'zope': 5.2, 'plone': None}
>>> print("plone: ", plone)
plone: None
```

 v: 3.7 ▼

Si el valor no es proveído, el valor `default_value` será el tipo objeto integrado `None`.

A continuación un ejemplo de como trabaja el método `setdefault()` la clave no esta en el diccionario pero esta vez el `default_value` es proveído:

```
>>> pkgs = {"python": 3.11, "zope": 5.2, "plone": None}
>>> print(pkgs)
{'python': 3.11, 'zope': 5.2, 'plone': None}
>>> django = paquetes.setdefault("django", 4.2.7)
>>> print("paquetes =", pkgs)
paquetes = {'python': 3.11, 'zope': 5.2, 'plone': None}
>>> print("django =", django)
django = 4.2.7
```

A continuación otro ejemplo en donde puedes agrupar N `tuplas` por el valor el cual se repite más y construir un diccionario que cuyas claves son los valores más repetidos y cuyos valores este agrupados en tipo `listas`:

```
>>> PKGS = (
...     ("zope", "Zope"),
...     ("zope", "zope.pagetemplate"),
...     ("plone", "Plone"),
...     ("plone", "ZODB3"),
...     ("plone", "plone.volto"),
... )
>>>
>>> paquetes = {}
>>> for clave, valor in PKGS:
...     if paquetes.has_key(clave):
...         paquetes[clave].append(valor)
...     else:
...         paquetes[clave] = [valor]
...
>>> print(paquetes)
{'zope': ['Zope', 'zope.pagetemplate'], 'plone': ['Plone', 'ZODB3', 'plone.volto']}
```

En el tipo `tupla` `PKGS` los elementos más repetidos son `'zope'` y `'plone'` estos se convierten en clave del diccionario `paquetes` y los otros elementos se agregan en listas como sus respectivos valores.

A continuación un mejor aprovechamiento implementando el método `setdefault()`:

```
>>> PKGS = (
...     ("zope", "Zope"),
...     ("zope", "zope.pagetemplate"),
...     ("plone", "Plone"),
...     ("plone", "ZODB3"),
...     ("plone", "plone.volto"),
... )
>>> paquetes = {}
>>> for clave, valor in PKGS:
...     paquetes.setdefault(clave, []).append(valor)
...
>>> print(paquetes)
{'zope': ['Zope', 'zope.pagetemplate'], 'plone': ['Plone', 'ZODB3', 'plone.volto']}
```

 v: 3.7 ▼

En el ejemplo anterior puede ver que el aprovechamiento del método `setdefault()` a comparación de no usar el respectivo método.

3.11.2.12. 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=3.11, zope=5.2, plone=6.0)
>>> print(versiones)
{'zope': 5.2, 'python': 3.11, 'plone': 6.0}
>>> versiones_adicional = dict(django=4.2.7)
>>> print(versiones_adicional)
{'django': 4.2.7}
>>> 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': 5.2, 'python': 3.11, 'plone': 6.0, 'django': 4.2.7}
```

3.11.2.13. values()

Este método devuelve una **lista** de los valores del **diccionario**:

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> versiones.values()
dict_values([3.11, 5.2, 6.0])
>>> for valor in versiones.values():
```



```
...     print(valor)
...
3.11
5.2
6.0
```

 v: 3.7 ▼

3.11.3. Funciones

Los objetos de tipo **diccionario** tienen disponibles una serie de *funciones* integradas en el interprete Python para su tratamiento, a continuación algunas de estas:

3.11.3.1. len()

Esta función es la misma función integrada `len()` en el interprete Python pero aplicada al uso de la secuencia de tipo **diccionario**.

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0)
>>> len(versiones)
3
```

3.11.4. Sentencias

Los objetos de tipo **diccionario** tienen disponibles una serie de *sentencias* integradas en el interprete Python para su tratamiento, a continuación algunas de estas:

3.11.4.1. del

Esta sentencia es la misma sentencia integrada `del` en el interprete Python pero aplicada al uso de la secuencia de tipo **diccionario**.

```
>>> versiones = dict(python=3.11, zope=5.2, plone=6.0, django=4.2.7)
>>> print(versiones)
{'python': 3.11, 'zope': 5.2, 'plone': 6.0, 'django': 4.2.7}
>>> del versiones["django"]
>>> print(versiones)
{'python': 3.11, 'zope': 5.2, 'plone': 6.0}
```

En el código fuente anterior se usa la sentencia `del` para eliminar un elemento del diccionario mediante su respectiva clave.

3.11.5. Convertir a diccionarios

Para convertir a *tipos diccionarios* debe usar la función `dict()` la cual [esta integrada](#) en el interprete Python.

Truco

 v: 3.7 ▼

Para más información consulte las funciones integradas para [operaciones de secuencias](#).

3.11.6. Ejemplos

A continuación, se presentan un ejemplo de su uso:

Definir un diccionario

```
1  datos_basicos = {
2      "nombres": "Leonardo Jose",
3      "apellidos": "Caballero Garcia",
4      "cedula": "26938401",
5      "fecha_nacimiento": "03/12/1980",
6      "lugar_nacimiento": "Maracaibo, Zulia, Venezuela",
7      "nacionalidad": "Venezolana",
8      "estado_civil": "Soltero",
9  }
```

Operaciones con tipo diccionario con funciones propias

```
1  print("\nClaves de diccionario:", datos_basicos.keys())
2  print("\nValores de diccionario:", datos_basicos.values())
3  print("\nElementos de diccionario:", datos_basicos.items())
```

Iteración avanzada sobre diccionarios con función items

```
1  for key, value in iter(datos_basicos.items()):
2      print(f"Clave: {key}, tiene el valor: {value}")
```

Caso real de usar tipo diccionario

```
1  print("\n\nInscripción de Curso")
2  print("=====")
3
4  print("\nDatos de participante")
5  print("-----")
6
7  print("Cédula de identidad: ", datos_basicos["cedula"])
8  print("Nombre completo: " + datos_basicos["nombres"] + " " +
9        datos_basicos["apellidos"])
10 import datetime, locale, os
11
12 locale.setlocale(locale.LC_ALL, os.environ["LANG"])
```

```
13 print(  
14     "Fecha y lugar de nacimiento:",  
15     datetime.datetime.strftime(  
16         datetime.datetime.strptime(datos_basicos["fecha_nacimie  
17 "%d/%m/%Y"),  
18         "%d de %B de %Y",  
19     )  
20     + " en "
```

 v: 3.7 ▼

3.11.7. Ayuda integrada

Usted puede consultar toda la documentación disponible sobre los **diccionarios** desde la [consola interactiva](#) de la siguiente forma:

```
>>> help(dict)
```



Importante

Usted puede descargar el código usado en esta sección haciendo clic [aquí](#).



Truco

Para ejecutar el código `tipo_diccionarios.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
$ python tipo_diccionarios.py
```



Ver también

Consulte la sección de [lecturas suplementarias](#) del entrenamiento para ampliar su conocimiento en esta temática.

¿Cómo puedo ayudar?

¡Mi soporte está aquí para ayudar!

Mi horario de oficina es de lunes a sábado, de 9 AM a 5 PM. [GMT-4 - Caracas, Venezuela](#).

La hora aquí es actualmente 7:35 PM GMT-4.

Mi objetivo es responder a todos los mensajes dentro de un día hábil.

[Contáctenos en la sección de soporte](#)

 v: 3.7 ▼



What do you think?

2 Respuestas



Upvote



Funny



Love



Surprised



Angry



Sad

0 Comentarios

 1 Acceder ▼

G

Sé el primero en comentar...

INICIAR SESIÓN CON

O REGISTRARSE CON DISQUS 

Nombre