3.12. Tipo conjuntos

Un conjunto, es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas.

Clase	Tipo	Notas	Ejemplo
`set`	Conjuntos	Mutable, sin orden, no contiene duplicados.	`set([4.0, 'Carro', True])`
`frozenset`	Conjuntos	Inmutable, sin orden, no contiene duplicados.	`frozenset([4.0, 'Carro', True])`

3.12.1. Métodos

Los objetos de tipo **conjunto mutable** y **conjunto inmutable** integra una serie de métodos integrados a continuación:

3.12.1.1. add()

Este método agrega un elemento a un **conjunto mutable**. Esto no tiene efecto si el elemento ya esta presente.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> set_mutable1.add(22)
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11, 22])
```

3.12.1.2. clear()

Este método remueve todos los elementos desde este **conjunto mutable**.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> print(set_mutable1)
```

```
set([1, 2, 3, 4, 5, 7, 11])
>>> set_mutable1.clear()
>>> print(set_mutable1)
set([])
```

v: 3.7

3.12.1.3. copy()

Este método devuelve una copia superficial del tipo **conjunto mutable** o **conjunto inmutable**:

```
>>> set_mutable = set([4.0, "Carro", True])
>>> otro_set_mutable = set_mutable.copy()
>>> set_mutable == otro_set_mutable
True
```

3.12.1.4. difference()

Este método devuelve la diferencia entre dos **conjunto mutable** o **conjunto inmutable**: todos los elementos que están en el primero, pero no en el argumento.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> print(set_mutable2)
set([2, 4, 5, 8, 9, 11])
>>> print(set_mutable1).difference(set_mutable2)
set([1, 3, 7])
>>> print(set_mutable2.difference(set_mutable1))
set([8, 9])
```

3.12.1.5. difference_update()

Este método actualiza un tipo **conjunto mutable** llamando al método `difference_update()` con la diferencia de los conjuntos.

```
>>> proyecto1 = {"python", "Zope", "ZODB3", "zope.pagetemplate"}
>>> proyecto1
set(['python', 'zope.pagetemplate', 'Zope', 'ZODB3'])
>>> proyecto2 = {"python", "Plone", "plone.volto"}
>>> proyecto2
set(['python', 'plone.volto', 'Plone'])
>>> proyecto1.difference_update(proyecto2)
>>> proyecto1
set(['zope.pagetemplate', 'Zope', 'ZODB3'])
```

Si `proyecto1` y `proyecto2` son dos conjuntos. La diferencia del conjunto `proyecto1` y conjunto `proyecto2` es un conjunto de elementos que existen solamente en el conjunto `proyecto1` pero no en el conjunto `proyecto2`.

3.12.1.6. discard()

Este método remueve un elemento desde un **conjunto mutable** si esta presente.

v: 3.7

```
>>> paquetes = {"python", "zope", "plone", "django"}
>>> paquetes
set(['python', 'zope', 'plone', 'django'])
>>> paquetes.discard("django")
>>> paquetes
set(['python', 'zope', 'plone'])
```

El **conjunto mutable** permanece sin cambio si el elemento pasado como argumento al método `discard()` no existe.

```
>>> paquetes = {"python", "zope", "plone", "django"}
>>> paquetes.discard("php")
>>> paquetes
set(['python', 'zope', 'plone'])
```

3.12.1.7. intersection()

Este método devuelve la intersección entre los **conjuntos mutables** o **conjuntos inmutables**: todos los elementos que están en ambos.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> print(set_mutable2)
set([2, 4, 5, 8, 9, 11])
>>> print(set_mutable1).intersection(set_mutable2)
set([2, 11, 4, 5])
>>> print(set_mutable2.intersection(set_mutable1))
set([2, 11, 4, 5])
```

3.12.1.8. intersection_update()

Este método actualiza un **conjunto mutable** con la intersección de ese mismo y otro **conjunto mutable**.

El método `intersection_update()` le permite arbitrariamente varios numero de argumentos (conjuntos).

```
>>> proyecto1 = {"python", "Zope", "zope.pagetemplate"}
>>> proyecto1
set(['python', 'zope.pagetemplate', 'Zope'])
>>> proyecto2 = {"python", "Plone", "plone.volto", "plone.restapi"}
>>> proyecto2
set(['python', 'plone.restapi', 'plone.volto', 'Plone'])
```

```
>>> proyecto3 = {"python", "django", "django-filter"}
>>> proyecto3
set(['python', 'django-filter', 'django'])
>>> proyecto3.intersection_update(proyecto1, proyecto2)
>>> proyecto3
set(['python'])
```

v: 3.7

La intersección de dos o más conjuntos es el conjunto de elemento el cual es común a todos los conjuntos.

3.12.1.9. isdisjoint()

Este método devuelve el valor `True` si no hay elementos comunes entre los **conjuntos mutables** o **conjuntos inmutables**.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> print(set_mutable2)
set([2, 4, 5, 8, 9, 11])
>>> print(set_mutable1).isdisjoint(set_mutable2)
```

3.12.1.10. issubset()

Este método devuelve el valor `True` si el **conjunto mutable** es un *subconjunto* del **conjunto mutable** o del **conjunto inmutable** argumento.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> set_mutable3 = set([11, 5, 2, 4])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> print(set_mutable2)
set([2, 4, 5, 8, 9, 11])
>>> print(set_mutable3)
set([2, 11, 4, 5])
>>> print(set_mutable2.issubset(set_mutable1))
False
>>> print(set_mutable3.issubset(set_mutable1))
True
```

3.12.1.11. issuperset()

Este método devuelve el valor `True` si el **conjunto mutable** o el **conjunto inmutable** es un *superset* del **conjunto mutable** argumento.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
```

```
>>> set_mutable3 = set([11, 5, 2, 4])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> print(set_mutable2)
set([2, 4, 5, 8, 9, 11])
>>> print(set_mutable3)
set([2, 11, 4, 5])
>>> print(set_mutable1).issuperset(set_mutable2)
False
>>> print(set_mutable1).issuperset(set_mutable3)
True
```

v: 3.7

3.12.1.12. pop()

Este método remueve arbitrariamente y devuelve un elemento de **conjunto mutable**. El método `pop()` no toma ningún argumento. Si el **conjunto mutable** esta vacío se lanza una excepción KeyError.

```
>>> paquetes = {"python", "zope", "plone", "django"}
>>> paquetes
set(['python', 'zope', 'plone', 'django'])
>>> print("Valor aleatorio devuelto es:", paquetes.pop())
Valor aleatorio devuelto es: python
>>> paquetes
set(['zope', 'plone', 'django'])
>>> print("Valor aleatorio devuelto es:", paquetes.pop())
Valor aleatorio devuelto es: zope
>>> paquetes
set(['plone', 'django'])
>>> print("Valor aleatorio devuelto es:", paquetes.pop())
Valor aleatorio devuelto es: plone
>>> paquetes
set(['django'])
>>> print("Valor aleatorio devuelto es:", paquetes.pop())
Valor aleatorio devuelto es: django
>>> print("Valor aleatorio devuelto es:", paquetes.pop())
Valor aleatorio devuelto es:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

Tenga en cuenta que usted podría obtener diferente salida devueltas usando el método `pop()` por que remueve aleatoriamente un elemento.

3.12.1.13. remove()

Este método busca y remueve un elemento de un **conjunto mutable**, si debe ser un miembro.

```
>>> paquetes = {"python", "zope", "plone", "django"}
>>> paquetes
set(['python', 'zope', 'plone', 'django'])
```

```
>>> paquetes.remove("django")
>>> paquetes
set(['python', 'zope', 'plone'])
```

v: 3.7

Si el elemento no es existe en el **conjunto mutable**, lanza una excepción KeyError. Usted puede usar el método discard() si usted no quiere este error. El **conjunto mutable** permanece sin cambio si el elemento pasado al método `discard()` no existe.

Un conjunto es una colección desordenada de elementos. Si usted quiere remover arbitrariamente elemento un conjunto, usted puede usar el método pop().

3.12.1.14. symmetric_difference()

Este método devuelve todos los elementos que están en un **conjunto mutable** e **conjunto inmutable** u otro, pero no en ambos.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> print(set_mutable2)
set([2, 4, 5, 8, 9, 11])
>>> print(set_mutable1).symmetric_difference(set_mutable2)
set([1, 3, 7, 8, 9])
```

3.12.1.15. symmetric_difference_update()

Este método actualiza un **conjunto mutable** llamando al método `symmetric_difference_update()` con los conjuntos de diferencia simétrica.

La diferencia simétrica de dos conjuntos es el conjunto de elementos que están en cualquiera de los conjuntos pero no en ambos.

```
>>> proyecto1 = {"python", "plone", "django"}
>>> proyecto1
set(['python', 'plone', 'django'])
>>> proyecto2 = {"django", "zope", "pyramid"}
>>> proyecto2
set(['zope', 'pyramid', 'django'])
>>> proyecto1.symmetric_difference_update(proyecto2)
>>> proyecto1
set(['python', 'zope', 'pyramid', 'plone'])
```

El método `symmetric_difference_update()` toma un argumento simple de un tipo **conjunto mutable**.

3.12.1.16. union()

Este método devuelve un **conjunto mutable** y **conjunto inmutable** con todos los elementos que están en alguno de los **conjuntos mutables** y **conjuntos inmutables**.

v: 3.7

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print(set_mutable1)
set([1, 2, 3, 4, 5, 7, 11])
>>> print(set_mutable2)
set([2, 4, 5, 8, 9, 11])
>>> print(set_mutable1).union(set_mutable2)
set([1, 2, 3, 4, 5, 7, 8, 9, 11])
```

3.12.1.17. update()

Este método agrega elementos desde un **conjunto mutable** (pasando como un argumento) un tipo tupla, un tipo lista, un tipo diccionario o un tipo **conjunto mutable** llamado con el método `update()`.

A continuación un ejemplo de agregar nuevos elementos un tipo **conjunto mutable** usando otro tipo **conjunto mutable**:

```
>>> version_plone_dev = set([5.1, 6])
>>> version_plone_dev
set([5.1, 6])
>>> versiones_plone = set([2.1, 2.5, 3.6, 4])
>>> versiones_plone
set([2.5, 3.6, 2.1, 4])
>>> versiones_plone.update(version_plone_dev)
>>> versiones_plone
set([2.5, 3.6, 4, 6, 5.1, 2.1])
```

A continuación un ejemplo de agregar nuevos elementos un tipo **conjunto mutable** usando otro tipo cadena de caracteres:

```
>>> cadena = "abc"
>>> cadena
'abc'
>>> conjunto = {1, 2}
>>> conjunto.update(cadena)
>>> conjunto
set(['a', 1, 2, 'b', 'c'])
```

A continuación un ejemplo de agregar nuevos elementos un tipo **conjunto mutable** usando otro tipo diccionario:

```
>>> diccionario = {"key": 1, 2: "lock"}
>>> diccionario.items()
dict_items([('key', 1), (2, 'lock')])
>>> conjunto = {"a", "b"}
>>> conjunto.update(diccionario)
```

```
>>> conjunto
{'a', 2, 'key', 'b'}
```

v: 3.7

3.12.2. Convertir a conjuntos

Para convertir a *tipos conjuntos* debe usar las funciones set() y frozenset(), las cuales están integradas en el interprete Python.

> **Truco**
>
> Para más información consulte las funciones integradas para operaciones de secuencias.

3.12.3. Ejemplos

3.12.3.1. Conjuntos set

A continuación, se presentan un ejemplo de conjuntos `set`:

```
1  # crea un conjunto sin valores repetidos y lo asigna la variable
2  para_comer = {"pastel", "tequeno", "papa", "empanada", "mandoca"}
3  print(para_comer, type(para_comer))
4  para_tomar = {"refresco", "malta", "jugo", "cafe"}
5  print(para_tomar, type(para_tomar))
6
7  # usa operaciones condicionales con operador in
8  hay_tequeno = "tequeno" in para_comer
9  hay_fresco = "refresco" in para_tomar
10
11 print("\nTostadas A que Pipo!")
12 print("====================")
13
14 # valida si un elemento esta en el conjunto
15 print("Tenéis tequeno?:", "tequeno" in para_comer)
16
17 # valida si un elemento esta en el conjunto
18 print("Tenéis pa' tomar fresco?:", "refresco" in para_tomar)
19
20 if hay_tequeno and hay_fresco:
21     print("Desayuno vergatario!!!")
22 else:
23     print("Desayuno ligero")
```

3.12.3.2. Conjuntos frozenset

A continuación, se presentan un ejemplo de conjuntos `frozenset`:

```
>>> versiones_plone = frozenset([6, 2.1, 2.5, 3.6, 4, 5, 4, 2.5])
>>> print(versiones_plone, type(versiones_plone))
frozenset([2.5, 4, 5, 6, 2.1, 3.6]) <type 'frozenset'>
```

v: 3.7

Los elementos de un set son únicos (sin repeticiones dentro del `set`), y deben ser objetos inmutables: números, cadena de caracteres, tuplas y sets inmutables, pero no listas ni sets mutables.

3.12.4. Ayuda integrada

Usted puede consultar toda la documentación disponible sobre los **conjuntos set** desde la consola interactiva de la siguiente forma:

```
>>> help(set)
```

Usted puede consultar toda la documentación disponible sobre los **conjuntos frozenset** desde la consola interactiva de la siguiente forma:

```
>>> help(frozenset)
```

Para salir de esa ayuda presione la tecla `Q`.

Importante

Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco

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

```
$ python tipo_conjuntos.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.

v: 3.7

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