Tuplas

En Python, una tupla es un conjunto ordenado e inmutable de elementos del mismo o diferente tipo. Una vez que se crea una tupla, como es inmutable, no se puede cambiar ni su contenido ni su tamaño. Los elementos se escriben entre paréntesis y separados por comas.

```
>>> (1, "z", 3.1416)
(1, 'z', 3.14)
```

En realidad no es necesario escribir los paréntesis para indicar que se trata de una tupla, basta con escribir las comas, pero Python escribe siempre los paréntesis:	La función len() devuelve el número de elementos de una tupla:	Una tupla puede no contener ningún elemento, es decir, ser una tupla vacía.

>>> 1, "z", 3.1416
(1, 'z', 3.1416)
	>>> len((1, "z", 3.1416))
	3
		>>> ()
		()
		>>> len(())
		0

Una tupla puede incluir un único elemento, pero para que Python entienda que nos estamos refiriendo a una tupla es necesario escribir al menos una coma. El ejemplo siguiente muestra la diferencia entre escribir o no una coma.

>>> (3)
3
	>>> (3,)
	(3,)
		>>> (3,)
		(3,)
		>>> len((3,))
		1

En el primer caso Python interpreta la expresión como un número.	En este segundo Python interpreta como una tupla de un único elemento.	Python escribe una coma al final en las tuplas de un único elemento para indicar que se trata de un tupla, pero esa coma no indica que hay un elemento después:

Operaciones con tuplas

Concatenación: La concatenación es la combinación de tuplas. Ejemplo:

Creamos 2 tuplas:	Si queremos concatenar la tuple1 con la tuple2 utilizamos el operador +

>>> tuple1 = (1,2,3,4,5)
>>> tuple2 = (6,7,8,9,10)
>>> tuple1
(1, 2, 3, 4, 5)
>>> tuple2
(6, 7, 8, 9, 10)
	>>> tuple1 + tuple2
	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Repetición	Comprobar elementos	Buscar	Contador
La repetición de tuplas se puede llevar a cabo mediante el uso del operador *. Ejemplo:	Para comprobar si un elemento existe en la tupla, simplemente hay que utilizar la palabra clave **in**. Ejemplo:	Si queremos saber en qué índice está localizado un valor concreto solo hay que usar index. Ejemplo:	Se puede contar el número de veces que un elemento existe en la tupla usando count().Ejemplo:

>> tuple1 * 3
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
>>>
	(1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
	>>> 5 in tuple1
	True
	>>> 7 in tuple1
	False
		>>> tuple1
		(1, 2, 3, 4, 5)
		>>> tuple1.index(3)
		2
		>>> tuple1.index(5)
		4
		>>>
			>>> tuple3 = (65,67,5,67,34,76,67,231,98,67)
			>>> tuple3
			(65, 67, 5, 67, 34, 76, 67, 231, 98, 67)
			>>> tuple3.count(67)
			4

Indexación: La indexación es el proceso de acceder al elemento de una tupla mediante un índice.

Si queremos acceder a la quinta posición de tupla, hay que recordar que los índices de la tupla comienzan en 0 (siempre que no esté vacía):	Un índice también puede ser negativo, si el recuento lo iniciamos desde la derecha de la tupla se inicia con -1. Si se escribe -0 es lo mismo que indicar 0.	Qué pasaría si hacemos referencia a un índice fuera de rango?.

>>> tupla
(65, 67, 5, 67, 34, 76, 67, 231, 98, 67)
>>> tupla [5]
76
	>>> tupla
	(65, 67, 5, 67, 34, 76, 67, 231, 98, 67)
	>>> tupla [-3]
	231
	>>> tupla [-0]
	65
		>>> tupla
		(65, 67, 5, 67, 34, 76, 67, 231, 98, 67)
		>>> tupla [-12]
		Traceback (most recent call last):
		File "<stdin>", line 1, in <module>
		IndexError: tuple index out of range

La tuplas no tienen métodos, append(), extend() para agregar elementos, ni remove() o pop() para eliminar porque son inmutables:

>>> tupla
('a', 1, 'Python', 4, 5.6, 'Juanito')
>>> tupla.append('tupla')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
	>>> tupla
	('a', 1, 'Python', 4, 5.6, 'Juanito')
	>>> tupla.remove(5.6)
	Traceback (most recent call last):
	File "<stdin>", line 1, in <module>
	AttributeError: 'tuple' object has no attribute 'remove'

Particionado

El particionado (en inglés slicing) de una tupla funciona como en las listas. Una partición de una tupla es una nueva tupla con los elementos seleccionados. Lo que diferencia a las tuplas de las listas es que las primeras no se pueden modificar. Se puede particionar una tupla porque en realidad se crea una nueva tupla.

>>> tupla
('a', 1, 'Python', 4, 5.6, 'Juan')
>>> tupla[1:3]
(1, 'Python')
	>>> tupla[:3]
	('a', 1, 'Python')
	>>> tupla[0:3]
	('a', 1, 'Python')
		>>> tupla[3:]
		(4, 5.6, 'Juan')
		>>> tupla[3:0]
		()
		>>> tupla[2:4]
		('Python', 4)

Las tuplas se pueden utilizar en un contexto booleano

>> def es_True_o_False(tupla): ... if (tupla): ... print("Es verdadero") ... else: ... print("Es falso")	
>>> es_True_o_False(()) Es falso	Una tupla vacía siempre es false en un contexto booleano.
>>> es_True_o_False(('a', 1, "Python", 4, 5.6, "Juanito")) Es verdadero	Una tupla con al menos un valor vale true.
>>> es_True_o_False(('Es tupla?',)) Es verdadero	Para crear una tupla con un único elemento, hay que escribir una coma después del valor.
>>> type(('Es tupla?',)) <class 'tuple'>	
>>> type(('Es tupla?'))	Sin la coma Python asume que lo que se está

<class 'str'>	haciendo es poner un par de paréntesis a una expresión, por lo que no se crea una tupla

Empaquetado y desempaquetado de tuplas

Empaquetado: (pack) Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por todos los valores asignados. A esta operación se la denomina empaquetado de tuplas:

```
>>> a = 12.34
>>> b = "@#"
>>> c = "Madrid"
>>> tupla= a, b, c
>>> len(tupla)
3
>>> tupla
(12.34, '@#', 'Madrid')
```

Desempaquetado: (unpack) Si se tiene una tupla de longitud k, se puede asignar la tupla a k variables distintas y en cada variable quedará una de las componentes de la tupla. A esta operación se la denomina desempaquetado de tuplas.

>>> tupla = ('a', 1, 'Python', 4, 5.6, 'Juanito')	Tupla tiene 6 elementos
>>> (r, s, t, x, y, z) = tupla	(r, s, t, x, y, z) es una tupla de 6 variables

>>> r
'a'
>>> s
1
>>> t
'Python'
>>> x
4
>>> y
5.6
>>> z
'Juanito'
	Al asignar una tupla a otra lo que sucede es que cada una de las variables se queda con el valor de los elementos de la primera tupla que corresponde a su posición.

Esta prestación tiene toda clase de usos, por ejemplo si se quiere asignar nombres a un rango de valores, se puede combinar la función range() con la asignación múltiple para hacerlo de una forma rápida:

>> (LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO) = range (7)	La función interna range() genera una secuencia de números enteros

>>> LUNES
0
>>> MARTES
1
>>> MIERCOLES
2
>>> JUEVES
3
>>> VIERNES
4
>>> SABADO
5
>>> DOMINGO
6
	Ahora cada variable tiene un valor: LUNES vale 0, MARTES vale 1, etc.

Tuplas que contienen listas

Una tupla puede contener distintos tipo de datos, incluyendo listas.

```
>>> tupla_de_listas = (["Jorge",6.7, "Peru", 28], ['a', 1, 'Python', 4, 5.6,
'Juanito'])
>>> tupla_de_listas
(['Jorge', 6.7, 'Peru', 28], ['a', 1, 'Python', 4, 5.6, 'Juanito'])
>>> tupla_de_listas[1][4]
5.6
>>> tupla_de_listas[1][2:4]
['Python', 4]
>>>
```

Tuplas que contienen otras tuplas

Una tupla puede contener distintos tipo de datos, incluyendo tuplas	Por ejemplo, una persona puede ser descripta por su nombre, su cuil y su fecha de nacimiento:	Para evitar crear variables innecesarias, que no se van a utilizar se suele asignar estos valores a la variable _. Por ejemplo, si sólo nos interesa el mes:

>>> tupla = ("Paris", 6.7, "Uruguay", 59, ('a', 1, 'Python', 4, 5.6, 'Juanito'))
>>> tupla
('Paris', 6.7, 'Uruguay', 59, ('a', 1, 'Python', 4, 5.6, 'Juanito'))
>>> tupla[2]
'Uruguay'
>>> tupla[4][1:5]
(1, 'Python', 4, 5.6)
>>> tupla[4][2:5]
('Python', 4, 5.6
	>>> persona = ('Fulano', '20-12345678-9', (1980, 5, 14))
	>>> nombre, cuil, (a, m, d) = persona
	>>> m
	5
		>>> _, _, (_, mes, _) = persona
		>>> mes
		5

Conversión

Las tuplas se pueden convertir en listas, y viceversa. La función tuple toma una lista y devuelve una tupla con los mismos elementos, y la función list toma una tupla y devuelve una lista.

Tupla a Lista	Lista a Tupla

>>> tupla
('a', 1, 'Python', 4, 5.6, 'Juanito')
>>> tupla_a_lista = list(tupla)
>>> tupla_a_lista
['a', 1, 'Python', 4, 5.6, 'Juanito']
	>>> lista
	['a', 1, 'Python', 4, 5.6, 'Juan']
	>>> lista_a_tupla = tuple(tupla_a_lista)
	>>> lista_a_tupla
	('a', 1, 'Python', 4, 5.6, 'Juan')

Comparación de tuplas

Dos tuplas son iguales cuando tienen el mismo tamaño y cada uno de sus elementos correspondientes tienen el mismo valor:	Para determinar si una tupla es menor que otra, se utiliza el orden lexicográfico. Si los elementos en la primera posición de ambas tuplas son distintos, ellos determinan el ordenamiento de las tuplas:	Si los elementos en la primera posición son iguales, entonces se usa el valor siguiente para hacer la comparación:	Si los elementos respectivos siguen siendo iguales, entonces se sigue probando con los siguientes uno por uno, hasta encontrar dos distintos. Si a una tupla se le acaban los elementos para comparar antes que a la otra, entonces es considerada menor que la otra:

>>> (1, 2) == (3/2, 1 + 1)
False
>>> (6, 1) == (6, 2)
	>>> (1, 4, 7) < (2, 0, 0, 1)
	True
		>>> (6, 1, 8) < (6, 2, 8)
		True
		>>> (6, 1, 8) < (6, 0)
			>>> (1, 2) < (1, 2, 4)
			True
			>>> (1, 3) < (1, 2, 4)

False >>> (6, 1) == (6, 1, 0) False	>>> (1, 9, 10) < (0, 5) False	False	False
	La primera comparación es True porque 1 < 2. La segunda comparación es False porque 1 > 0. No importa el valor que tengan los siguientes valores, o si una tupla tiene más elementos que la otra.	La primera comparación es True porque 6 == 6 y 1 < 2. La segunda comparación es False porque 6 == 6 y 1 > 0.	La primera comparación es True porque 1 == 1, 2 == 2, y ahí se acaban los elementos de la primera tupla. La segunda comparación es False porque 1 == 1 y 3 < 2; en este caso sí se alcanza a determinar el resultado antes que se acaben los elementos de la primera tupla.

Iteración sobre tuplas

Una tabla de datos puede representarse como una lista de tuplas. Por ejemplo, la información de los alumnos que están cursando una carrera puede ser representada así:

```
alumnos = [
    ('Pachi', 'Mengano', '20-451189132-5', 'Biomédica'),
    ('Tato', 'Sultano', '20-251199122-6', 'Industrial'),
    ('Paqui', 'De Tal',  '27-303149987-7', 'Electrónica'),
]
```

En este caso, se puede desempaquetar los valores automáticamente al recorrer la lista en un ciclo for:	Y como el apellido y el cuil no son usados:	Al igual que las listas, las tuplas son iterables:
>>> for nombre, apellido, cuil, carrera in alumnos: ... print (f"{nombre} estudia {carrera}") ... Pachi estudia Biomédica Tato estudia Industrial Paqui estudia Electrónica	>>> for nombre, _, _, carrera in alumnos: ... print (f"{nombre} estudia {carrera}") ... Pachi estudia Biomédica Tato estudia Industrial Paqui estudia Electrónica	>>> for valor in (6, 1): ... print(f"{valor ** 2}") ... 36 1

Listas versus Tuplas

Listas (mutables)	Tuplas (inmutables)
Las listas se utilizan en las situaciones en las que los elementos a agrupar pueden ir variando a lo largo del tiempo. Por ejemplo, para representar un las notas de un alumno en diversas materias, los inscriptos para un evento o la clasificación de los equipos en una competencia.	Las tuplas son más rápidas que las listas. Se pueden usar como claves en un diccionario porque son inmutables, pero si se tiene una tupla de listas, eso cuenta como mutable. Las tuplas se usan siempre que es necesario agrupar valores. Generalmente, conceptos del mundo real son representados como tuplas que agrupan información sobre ellos. En otros lenguajes, las tuplas reciben el nombre de **registros**.
[ ]	( )
+	+
append()	-
insert()	-
extend()	-
remove()	-
pop()	-

Conjuntos

Un conjunto es una colección de objetos únicos. Los conjuntos son ampliamente utilizados en lógica y matemática. Es una estructura mutable. Los conjuntos no pueden tener elementos duplicados. Los elementos de un conjunto deben ser inmutables.

Creación de un conjunto

Para crear un conjunto especificamos sus elementos entre llaves	Al igual que otras colecciones, sus miembros pueden ser de diversos tipos:	No obstante, un conjunto no puede incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos.	Python no puede dirimir el siguiente caso, Por defecto, la asignación siguiente crea un diccionario.

>>> conjunto = {1,2,3,4}
>>> conjunto
{1, 2, 3, 4}
	>>> conjunto = {True, 3.14, None, False, "Hola mundo", (1, 2)}
	>>> conjunto
	{False, True, (1, 2), 3.14, None, 'Hola mundo'}
	>>>
		>>> conjunto = {[1,2,3]}
		Traceback (most recent call last):
		File "<stdin>", line 1, in <module>
		TypeError: unhashable type: 'list'
			>>> conjunto ={}
			>>>

set

Para generar un conjunto vacío, directamente creamos una instancia de la clase set:	De la misma forma podemos obtener un conjunto a partir de cualquier objeto iterable:	Un set puede ser convertido a una lista y viceversa. En este último caso, los elementos duplicados son unificados.

>> conjunto = set()	>>> set1 = set([1, 2, 3, 4])
	>>> set1
	{1, 2, 3, 4}
	>>> set2 = set(range(10))
	>>> set2
	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
		>>> lista = list({1, 2, 3, 4})
		>>> lista
		[1, 2, 3, 4]
		>>> set_ = set([1, 2, 2, 3, 4])
		>>> set_
		{1, 2, 3, 4}

Elementos

Los conjuntos son objetos mutables. Podemos agregar o eliminar elementos.

Agregar elementos		Eliminar elementos	

>>> conjunto = {1,2,3,4}
>>> conjunto
{1, 2, 3, 4}
>>> conjunto.**add**(5)
>>> conjunto
{1, 2, 3, 4, 5}
	>>> conjunto = {1,2,3,4}
	>>> conjunto
	{1, 2, 3, 4}
	>>> conjunto.**update**({2,3,9},{4,3,5,8,13})
	>>> conjunto
	{1, 2, 3, 4, 5, 8, 9, 13}
		>> conjunto
		{1, 2, 3, 4, 5}
		>>> conjunto.**discard**(2)
		>>> conjunto
		{1, 3, 4, 5}
		>>>
			>> conjunto
			{1, 2, 3, 4, 5, 8, 9, 13}
			>>> conjunto.**remove**(4)
			>>> conjunto
			{1, 2, 3, 5, 8, 9, 13}
			>>>

Si el elemento pasado como argumento a discard() no está dentro del conjunto es simplemente ignorado. En cambio, el método remove() opera de forma similar pero en dicho caso da la excepción KeyError.

Otras funciones y métodos

Para determinar si un elemento pertenece a un conjunto, utilizamos la palabra reservada in	La función clear() elimina todos los elementos.	El método pop() remueve los miembros de un conjunto:	Para obtener el número de elementos aplicamos función len():

>>> conjunto
{1, 3, 4, 5}
>>> conjunto
	>>> conjunto = {1,2,3,4,5}
	>>> conjunto
		>>> conjunto
		{1, 2, 3, 4, 5}
		>>> conjunto.pop()
			>>> conjunto
			{1, 2, 3, 4, 5}
			>>> len(conjunto)

{1, 3, 4, 5}
>>> 3 in conjunto
True
>>> 2 in conjunto
False
	{1, 2, 3, 4, 5}
	>>> conjunto.clear()
	>>> conjunto
	set()
	>>>
		1
		>>> conjunto
		{2, 3, 4, 5}
		>>> conjunto.pop()
		2
		>>> conjunto
		{3, 4, 5}
			5

remove() y pop() lanzan la excepción KeyError cuando un elemento no se encuentra en el conjunto o bien éste está vacío, respectivamente.

Operaciones principales

Algunas de las propiedades más interesantes de los conjuntos radican en sus operaciones principales: ***unión, intersección y diferencia.***

Unión	Intersección	Diferencia
La unión se realiza con el caracter | y retorna un conjunto que contiene los elementos que se encuentran en al menos uno de los dos conjuntos involucrados en la operación. Es simétrica	La intersección opera de forma análoga, pero con el operador & y retorna un nuevo conjunto con los elementos que se encuentran en ambos. Es simétrica	La diferencia, por último, retorna un nuevo conjunto que contiene los elementos de conjunto1 que no están en conjunto2. No es simétrica.

>> conjunto1 = {1,2,3,4,5}
>>> conjunto1
{1, 2, 3, 4, 5}
>>> conjunto2 = {0,1,2,5,6,7}
>>> conjunto2
{0, 1, 2, 5, 6, 7}
>>> conjunto1 | conjunto2
{0, 1, 2, 3, 4, 5, 6, 7}
	>>> conjunto1
	{1, 2, 3, 4, 5}
	>>> conjunto2 = {0,1,2,5,6,7}
	>>> conjunto2
	{0, 1, 2, 5, 6, 7}
	>>> conjunto1 & conjunto2
	{1, 2, 5}
		>>> conjunto1 = {1,2,3,4,5}
		>>> conjunto1
		{1, 2, 3, 4, 5}
		>>> conjunto2 = {0,1,2,5,6,7}
		>>> conjunto2
		{0, 1, 2, 5, 6, 7}
		>>> conjunto1 - conjunto2
		{3, 4}

>>> conjunto1 = {1,2,3,4,5}
>>> conjunto1
{1, 2, 3, 4, 5}
>>> conjunto2 = {0,1,2,5,6,7}
>>> conjunto2
{0, 1, 2, 5, 6, 7}
>>> conjunto1.**union**(conjunto2)
{0, 1, 2, 3, 4, 5, 6, 7}
	>>> conjunto1 = {1,2,3,4,5}
	>>> conjunto1
	{1, 2, 3, 4, 5}
	>>> conjunto2 = {0,1,2,5,6,7}
	>>> conjunto2
	{0, 1, 2, 5, 6, 7}
	>>> conjunto1.**intersection**(conjunto2)
	{1, 2, 5}
		>>> conjunto1
		{1, 2, 3, 4, 5}
		>>> conjunto2 = {0,1,2,5,6,7}
		>>> conjunto2
		{0, 1, 2, 5, 6, 7}
		>>> conjunto1.**difference**(conjunto2)
		{3, 4}

>>> conjunto1 = {1,2,3,4,5}
>>> conjunto1
{1, 2, 3, 4, 5}
>>> conjunto2 = {0,1,2,5,6,7}
>>> conjunto2
{0, 1, 2, 5, 6, 7}
>>> conjunto1.union(conjunto2)
== conjunto2.union(conjunto1)
True
	>>> conjunto1 = {1,2,3,4,5}
	>>> conjunto1
	{1, 2, 3, 4, 5}
	>>> conjunto2 = {0,1,2,5,6,7}
	>>> conjunto2
	{0, 1, 2, 5, 6, 7}
	>>> conjunto1.intersection(conjunto2)
	==
	conjunto2.intersection(conjunto1)
	True
		>>> conjunto1 = {1,2,3,4,5}
		>>> conjunto1
		{1, 2, 3, 4, 5}
		>>> conjunto2 = {0,1,2,5,6,7}
		>>> conjunto2
		{0, 1, 2, 5, 6, 7}
		>>> conjunto1.difference(conjunto2)
		==
		conjunto2.difference(conjunto1)
		False

Dos conjuntos son iguales si y solo si contienen los mismos elementos (a esto se lo conoce como ***principio de extensionalidad***:

>>> conjunto1 = {1,2,3,4,5}
>>> conjunto1
{1, 2, 3, 4, 5}
>>> conjunto2 = {5,4,3,2,1}
>>> conjunto2
{1, 2, 3, 4, 5}
>>> conjunto1 == conjunto2
	>> conjunto1 = {1,2,3,4,5}
	>>> conjunto1
	{1, 2, 3, 4, 5}
	>>> conjunto2 = {0,1,2,5,6,7}
	>>> conjunto2
	{0, 1, 2, 5, 6, 7}
	>>> conjunto1 == conjunto2

True	False

Se dice que conjunto2 es un **subconjunto** de conjunto1 cuando todos los elementos de aquél pertenecen también a éste. Python puede determinar esta relación con el método issubset().	Inversamente, se dice que conjunto1 es un **superconjunto** de conjunto2.	La definición de estas dos relaciones nos lleva a concluir que todo conjunto es al mismo tiempo un subconjunto y un superconjunto de sí mismo

>>> conjunto1 = {1,2,3,4,5}
>>> conjunto1
{1, 2, 3, 4, 5}
>>> conjunto2 = {1,2,5}
>>> conjunto2
{1, 2, 5}
>>>
conjunto2.**issubset**(conjunto1)
True
>>> conjunto2 < conjunto1
True
>>>
	>>> conjunto1 = {1,2,3,4,5}
	>>> conjunto1
	{1, 2, 3, 4, 5}
	>>> conjunto2 = {1,2,5}
	>>> conjunto2
	{1, 2, 5}
	>>>
	conjunto1.**issuperset**(conjunto2)
	True
	>>> conjunto1 > conjunto2
	True
	>>>
		>> conjunto1
		{1, 2, 3, 4, 5}
		>>> conjunto2 = {1,2,5}
		>>> conjunto2 {1, 2, 5}
		>>>
		conjunto1.issubset(conjunto1)
		True
		>>>
		conjunto1.issuperset(conjunto1)
		True
		>>>
		conjunto2.issubset(conjunto2)
		True
		>>>
		conjunto2.issuperset(conjunto2)
		True

La **diferencia simétrica** (^) retorna un nuevo conjunto el cual contiene los elementos que pertenecen a alguno de los dos conjuntos que participan en la operación pero no a ambos. Podría entenderse como una unión exclusiva.

>>> conjunto1 = {1,2,3,4,5}
>>> conjunto1
{1, 2, 3, 4, 5}
>>> conjunto2 = {1,2,5,6,7,8}
>>> conjunto2
{1, 2, 5, 6, 7, 8}
>>> conjunto1.symmetric_difference(conjunto2)
{3, 4, 6, 7, 8}
>>> conjunto1 ^ conjunto2
{3, 4, 6, 7, 8}
	>>> conjunto1 = {1,2,3,4,5}
	>>> conjunto1
	{1, 2, 3, 4, 5}
	>>> conjunto2 = {1,2,5,6,7,8}
	>>> conjunto2
	{1, 2, 5, 6, 7, 8}
	>>> conjunto2.symmetric_difference(conjunto1)
	{3, 4, 6, 7, 8}
	>>> conjunto2 ^ conjunto1
	{3, 4, 6, 7, 8}

Por último, se dice que un conjunto es **disconexo** o **disyunto** respecto de otro si no comparten elementos entre sí.

```
>>> conjunto1 = {1,2,3,4,5}
>>> conjunto1
{1, 2, 3, 4, 5}
>>> conjunto2 = {1,2,5,6}
>>> conjunto2
{1, 2, 5, 6}
>>> conjunto3 = {7,8,9}
>>> conjunto3
{8, 9, 7}
>>> conjunto1.isdisjoint(conjunto2)
False
>>> conjunto1.isdisjoint(conjunto3)
True
```

En otras palabras, dos conjuntos son disconexos si su intersección es el conjunto vacío.

Conjuntos inmutables

frozenset es una implementación similar a set pero inmutable. Es decir, comparte todas las operaciones de conjuntos provistas anteriormente a excepción de aquellas que implican alterar sus elementos (add(), discard(), etc.). La diferencia es análoga a la existente entre una lista y una tupla.

```
>>> conjunto1 = frozenset({1,2,3,4,5})
>>> conjunto1
frozenset({1, 2, 3, 4, 5})
>>> conjunto2 = frozenset({1,2,5,6})
>>> conjunto2
frozenset({1, 2, 5, 6})
```

```
>>> conjunto1 & conjunto2
frozenset({1, 2, 5})
>>> conjunto1 | conjunto2
frozenset({1, 2, 3, 4, 5, 6})
>>> conjunto1.isdisjoint(conjunto2)
False
```

Listas versus Tuplas versus Conjuntos

Listas (mutables)	Tuplas (inmutables)	Conjuntos (mutables) salvo en el caso de los frozenset
Las listas se utilizan en las situaciones en las que los elementos a agrupar pueden ir variando a lo largo del tiempo. Por ejemplo, para representar un las notas de un alumno en diversas materias, los inscriptos para un evento o la clasificación de los equipos en una competencia.	Las tuplas son más rápidas que las listas. Se pueden usar como claves en un diccionario porque son inmutables, pero si se tiene una tupla de listas, eso cuenta como mutable. Las tuplas se usan siempre que es necesario agrupar valores. Generalmente, conceptos del mundo real son representados como tuplas que agrupan información sobre ellos. En otros lenguajes, las tuplas reciben el nombre de **registros**.	En informática un conjunto es una estructura de datos que representa los conjuntos finitos que ya conocemos en matemáticas y los cuales son una colección de diferentes valores sin ningún orden alguno ni valores duplicados.
[ ]	( )	{}
+	+	add()
append()	count()	clear()
clear()	index()	copy()
copy()	-	difference()
count()	-	Intersection()
extend()	-	Isdisjoint()
pop()	-	Issubset()
index()		Issuperset()
insert()		pop()
pop()		remove()
remove()		symmetric_difference()
reverse		update()
sort()		union()
		Y mas…

Diccionarios

Un diccionario es un tipo de datos que sirve para asociar pares de objetos, puede ser visto como una colección de claves (llaves), cada una de las cuales tiene asociada un valor. Las claves (llaves) no están ordenadas y no hay claves repetidas. La única manera de acceder a un valor es a través de su clave (llave) a diferencia de las listas y tuplas, cuyos elementos se identifican por su posición. Las claves suelen ser números enteros o cadenas, aunque cualquier otro objeto inmutable puede actuar como una clave. Los valores, por el contrario, pueden ser de cualquier tipo, incluso otros diccionarios.

Crear diccionarios

Los diccionarios se crean usando llaves { : , : } La llave y el valor van separados por dos puntos:

En este ejemplo, las llaves son ‘Juan’, 'Jose' y 'Pedro', y los valores asociados a ellas son, respectivamente, 23, 20 y 26.	Un diccionario vacío puede ser creado usando {} o con la función dict():

>>> diccionario= {"Juan" : 23, "Jose" : 20, "Pedro" : 26}
>>> print(diccionario)
{'Juan' : 23, 'Jose' : 20, 'Pedro' : 26}
>>> type(diccionario)
	>> d = {}
	>>> d = dict()
	>>> d
	{}

<class 'dict'>	

Otras formas de crear diccionarios

	El método estático fromkeys() genera un diccionario a partir de un conjunto de claves y les asigna a todas ellas el mismo valor (None por defecto). El primer argumento puede ser, de hecho, cualquier objeto iterable.

>>> d = dict(Python=1991, C=1972, Java=1996)
>>> d
{'Python': 1991, 'C': 1972, 'Java': 1996}
	>>> dict.fromkeys(["Python", "C", "Java"], 0)
	{'Python': 0, 'C': 0, 'Java': 0}

Restricciones sobre las llaves

No se puede usar cualquier objeto como llave de un diccionario. Las llaves deben ser de un tipo de datos inmutable. Por ejemplo, no se puede usar listas:

```
>>> d = {[1, 2, 3]: 'hola'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Acceder a los elementos

El valor asociado a la llave k en el diccionario d se puede obtener mediante diccionario[k]:	A diferencia de los índices de las listas, las llaves de los diccionarios no necesitan ser números enteros. Si la llave no está presente en el diccionario, ocurre un error de llave (KeyError):	Una forma alternativa para obtener un valor es el método get(), indicando la clave como argumento. Agregando is None cuando la clave no existe el valor retornado es True:

>> diccionario = {"Juan":23,"Jose":20,"Pedro":26}
>>> print(diccionario)
{'Juan': 23, 'Jose': 20, 'Pedro': 26}
>>> diccionario["Jose"]
20
	>>> diccionario = {"Juan":23,"Jose":20,"Pedro":26}
	>>> print(diccionario)
	{'Juan': 23, 'Jose': 20, 'Pedro': 26}
	>>> diccionario["Pepe"]
	Traceback (most recent call last):
	  File "<stdin>", line 1, in <module>
	KeyError: 'Pepe
		>>> print(diccionario)
		{'Juan': 'Ingeniero', 'Jose': 'Medico', 'Pedro': 'Abogado'}
		>>> diccionario.get("Jose")
		'Medico'
		>>> diccionario.get("Pepe") is None
		True
		>>> diccionario.get("Pepe", "No está en el diccionario")
		'No está en el diccionario'

Agregar, modificar y eliminar elementos

Agregar	Modificar	Eliminar
Se puede agregar una llave nueva simplemente asignándole un valor:	Si se asigna un valor a una llave que ya estaba en el diccionario, el valor anterior se sobreescribe. Recuerde que un diccionario no puede tener llaves repetidas pero sí el valor:	Para borrar una llave, se puede usar la sentencia del:

>>> diccionario = {"Juan":23,"Jose":20,"Pedro":26}
>>> print(diccionario)
{'Juan': 23, 'Jose': 20, 'Pedro': 26}
>>> diccionario["Pepe"] = 28
>>> print(diccionario)
{'Juan': 23, 'Jose': 20,
	{'Juan': 23, 'Jose': 20, 'Pedro': 26, 'Pepe': 28}
	>>> diccionario["Pedro"] = 19
	>>> print(diccionario)
	{'Juan': 23, 'Jose': 20, 'Pedro': 19, 'Pepe': 28}
		>>> diccionario
		{'Juan': 23, 'Jose': 20, 'Pedro': 19, 'Pepe': 28}
		>>> del diccionario['Pedro']
		>>> diccionario
		{'Juan': 23, 'Jose': 20, 'Pepe': 28}

'Pedro': 26, 'Pepe': 28}		

El método clear() elimina todas las claves y valores:	len(diccionario) entrega cuántos pares llave-valor hay en el diccionario:

>>> diccionario
{'Juan': 23, 'Jose': 20, 'Pedro': 26}
>>> diccionario.clear()
>>> diccionario
{}
	>>> diccionario
	{'Juan': 23, 'Jose': 20, 'Pedro': 26}
	>>> len(diccionario)
	3

Otros métodos

Podemos actualizar un diccionario o bien unir dos de ellos con update():	La método pop() elimina un elemento una vez retornado:	El método popitem() retorna un elemento aleatorio y después lo elimina

>>> diccionario1 = {"Juan":"Ingeniero","Caro":"Medico","Pedro":"Abogado"}
>>> print(diccionario1)
{'Juan': 'Ingeniero', 'Caro': 'Medico', 'Pedro': 'Abogado'}
>>> diccionario2 = {"Maria":23,"Pepa":22,"Kika":27}
>>> print(diccionario2)
{'Maria': 23, 'Pepa': 22, 'Kika': 27}
>>> diccionario1.update(diccionario2)
>>> diccionario1
{'Juan': 'Ingeniero', 'Caro': 'Medico', 'Pedro': 'Abogado', 'Maria': 23, 'Pepa': 22, 'Kika': 27
	>>> diccionario
	{'Juan': 'Ingeniero', 'Caro': 'Medico', 'Pedro': 'Abogado', 'Maria': 23, 'Pepa': 22, 'Kika': 27}
	>>> diccionario.pop("Caro")
	'Medico'
	>>> diccionario
	{'Juan': 'Ingeniero', 'Pedro': 'Abogado', 'Maria': 23, 'Pepa': 22, 'Kika': 27}
		>>> diccionario
		{'Juan': 'Ingeniero', 'Caro': 'Medico', 'Pedro': 'Abogado', 'Maria': 23, 'Pepa': 22, 'Kika': 27}
		>>> diccionario.popitem()
		('Kika', 27)
		>>> diccionario
		{'Juan': 'Ingeniero', 'Caro': 'Medico', 'Pedro': 'Abogado', 'Maria': 23, 'Pepa': 22}

Operadores in y not in

k in diccionario permite saber si la llave k está en el diccionario:	Para saber si una llave no está en el diccionario, se usa el operador not in:

>>> patas = {'gato': 4, 'humano': 2, 'pulpo': 8, 'perro': 4, 'ciempies': 100}
>>> patas
{'gato': 4, 'humano': 2, 'pulpo': 8, 'perro': 4, 'ciempies': 100}
>>> 'perro' in patas
True
>>> 'lombriz' in patas
False
	>>> patas = {'gato': 4, 'humano': 2, 'pulpo': 8, 'perro': 4, 'ciempies': 100}
	>>> patas
	{'gato': 4, 'humano': 2, 'pulpo': 8, 'perro': 4, 'ciempies': 100}
	>>> 'lombriz' not in patas
	True

Es posible crear listas de llaves o valores:

```
>>> diccionario
{'Juan': 23, 'Jose': 20, 'Pedro': 26}
>>> lista = list(diccionario)
>>> lista
['Juan', 'Jose', 'Pedro']
>>> valores = list(diccionario.values())
>>> valores
[23, 20, 26]
```

Iteraciones

Los diccionarios son iterables:

Al iterar sobre un diccionario en un ciclo for, se obtiene las llaves:	Para iterar sobre las llaves, se usa values():	Para iterar sobre las llaves y los valores simultáneamente, se usa el método items():

>>> diccionario = {"Juan":23,"Jose":20,"Pedro":26}
>>> print(diccionario)
{'Juan': 23, 'Jose': 20, 'Pedro': 26}
>>>
>>> for k in diccionario:
... print(k)
Juan
Jose
Pedro
	>> diccionario = {"Juan":23,"Jose":20,"Pedro":26}
	>>> print(diccionario)
	{'Juan': 23, 'Jose': 20, 'Pedro': 26}
	>>> for v in diccionario.values():
	... print(v)
	... # nombrediccionario = {a1:a,...}
	23
	20
	26
		>>> print(diccionario)
		{'Juan': 23, 'Jose': 20, 'Pedro': 26}
		>>>
		>>> for k, v in diccionario.items():
		... print('La edad de ', k, 'es', v)
		... # nombrediccionario = {a1:a,...}
		...
		La edad de Juan es 23
		La edad de Jose es 20
		La edad de Pedro es 26

Listas versus Tuplas versus Conjuntos versus Diccionarios

Listas (mutables)	**Tuplas (inmutables)**	**Conjuntos (mutables) salvo en el caso de los frozenset**	**Diccionarios (mutables)**
Las listas se utilizan en las situaciones en las que los elementos a agrupar pueden ir variando a lo largo del tiempo. Por ejemplo, para representar un las notas de un alumno en diversas materias, los inscriptos para un evento o la clasificación de los equipos en una competencia.	Las tuplas son más rápidas que las listas. Se pueden usar como claves en un diccionario porque son inmutables, pero si se tiene una tupla de listas, eso cuenta como mutable. Las tuplas se usan siempre que es necesario agrupar valores. En otros lenguajes, las tuplas reciben el nombre de **registros**.	En ciencias de la computación un conjunto es una estructura de datos que representa los conjuntos finitos que ya conocemos en matemáticas y los cuales son una colección de diferentes valores sin ningún orden alguno ni valores duplicados.	Son una herramienta muy versátil. Se puede utilizar un diccionario, por ejemplo, para contar cuántas apariciones de cada palabra hay en un texto, o cuántas apariciones de cada letra. En general, los diccionarios sirven para crear bases de datos muy simples, en las que la clave es el identificador del elemento, y el valor son todos los datos del elemento a considerar.
[ ]	( )	{ }	{:,:}
+	+	add()	clear()
append()	count()	clear()	copy()
clear()	index()	copy()	get()
copy()		difference()	items()
count()		Intersection()	keys()
extend()		Isdisjoint()	pop()
pop()		Issubset()	popitem()
index()		Issuperset()	setdefault()
insert()		pop()	update()
pop()		remove()	values()
remove()		symmetric_difference()	
reverse		update()	
sort()		union()	
…	…	…	…