Argentina Programa 4.0

Universidad Nacional de San Luis

Desarrollador Python

Lenguaje de Programación Python: Tipos de Dato Colección

Autor:

Dr. Mario Marcelo Berón

UNIDAD 3

LENGUAJE DE PROGRAMACIÓN PYTHON: TIPOS DE DATO COLECCIÓN

3.1. Introducción

En la unidad anterior se dieron a conocer los tipos de datos primitivos que posee el lenguaje Python. En esta unidad se extenderán los conceptos para aprender a agrupar grandes cantidades de datos en colecciones de datos. Python proporciona varios tipos de colecciones que son muy útiles para la construcción de programas. En las secciones siguientes se describirán los conceptos y operaciones básicas cada una de ellas.

3.2. Tipo de Dato Secuencia

Un tipo de dato secuencia es aquel que soporta las operaciones de *membresía* (*in*), *longitud* (*len*) y *rebanadas* (slices). Si bien Python provee varios tipos secuencia esta sección estará orientada a describir tres de ellas:

tuplas, *tuplas nombradas* y *listas*.

3.2.1. Tuplas

Una tupla es una secuencia de cero o más referencias a objetos. Las tuplas poseen las siguientes características:

- Son inmutables no se pueden reemplazar ni borrar sus elementos.
- Se puede crear una tupla usando el constructor del tipo *tuple()* el cual cuando se invoca sin argumentos crea una tupla vacía. Cuando se invoca con una tupla como argumento crea una copia superficial de dicha tupla y con cualquier otro argumento lo intenta convertir a tupla. El constructor acepta solo un argumento.
- Las tuplas son heterogéneas.
- Una tupla también se puede crear sin el constructor. Si se desea una tupla vacía se coloca (). Si se desea una tupla con uno o más elementos se colocan los paréntesis y los elementos separados por comas.

Representación de una Tupla y sus Índices

t[-5]	t[-4]	t[-3]	t[-2]	t[-1]
'venus'	-28	'green'	'21'	19.74
t[0]	t[1]	t[2]	t[3]	t[4]

Las tuplas se indexan de la misma forma que lo hacen los strings, la diferencia radica en que en las tuplas cada posición puede contener un objeto y no solo un carácter como en el caso de los strings.

Las tuplas proveen dos métodos *t.count(x)* y *t.index(x)*. El primero retorna el número de veces que el objeto x aparece en la tupla t. El segundo devuelve como resultado el índice de la ocurrencia de más a la izquierda del objeto x en la tupla t. Si el objeto x no está en t entonces el método dispara

una excepción *ValueError*. Con las tuplas también se pueden usar los operadores + (concatenación), * (réplica), [] (rebanadas-slice), *in* y *not in* para comprobar membresía. Los operadores += y *= se pueden utilizar a pesar de que las tuplas son inmutables. Las tuplas se pueden comparar utilizando los operadores relacionales tradicionales (<, <=, ==, !=, >= y >) en estos casos las comparaciones se realizan elemento por elemento (y recursivamente para elementos anidados tales como tuplas de tuplas).

Operaciones con Tuplas

```
>>> cabello = "negro", "marrón", "rubio", "rojo"
>>> cabello[2]
'rubio'
>>> cabello[-3:] # igual que: cabello[1:]
('marron', 'rubio', 'rojo')
>>> cabello[:2], "gris", cabello[2:]
(('negro', 'marrón'), 'gris', ('rubio', 'rojo'))
```

Tuplas Nombradas

Una *Tupla Nombrada* se comporta como una tupla y tiene el mismo desempeño. Sin embargo, la característica que incorpora es la posibilidad de referirse a los campos por los nombres tan bien como con índices.

El constructor del tipo se llama *namedtuple()* y se encuentra definido en el módulo *collections*. El constructor recibe dos argumentos:

- El nombre del tipo personalizado que se desea crear.
- Un string de nombres separados por comas uno por cada elemento del tipo de datos personalizado.

El valor de retorno es una *clase personalizada*[1] un tipo de dato que se puede usar para crear tuplas nombradas.

[1]Este concepto es de programación orientada a objetos pero con el fin de que se pueda comprender el lector lo debe relacionar con la definición de un tipo de dato.

Definición del Tipo Venta y Creaciones de Instancias del Tipo

```
Venta = collections.namedtuple("Venta",
                               """IDProducto
                                  IDCliente
                                  Fecha
                                  Cantidad
                                  Precio""")
primerVenta=Venta(432, 921, "2008-09-14", 3, 7.99)
segundaVenta=Venta(419, 874, "2008-09-15", 1, 18.49)
```

3.2.2. Ejercicios

Nota: Asuma una cantidad específica de elementos cuando el ejercicio no lo especifique.

Ejercicio 1: Realice las siguientes actividades:

1. Defina una dupla d donde los elementos están inicializados en 0.
2. Defina una tupla de un único elemento.
3. Defina una tupla con n elementos inicializados en 0.

Ejercicio 2: Defina las duplas op0 y op1 y luego construya la tupla r cuyos elementos son la suma de los elementos de op0 y op1.
Ejemplo: Si op0=(10,20) y op1=(8,20) la tupla r tiene que contener r=(18,40).

Ejercicio 3: Escriba un ejemplo que muestre que las tuplas son inmutables.

Ejercicio 4: Escriba un programa que dada una tupla t con 5 elementos y un número n produzca como resultado una nueva tupla con todos los elementos de la tupla t multiplicados por el número n.

Ejercicio 5: Escriba un programa que almacene el valor de tres variables ingresadas por el usuario en una tupla.

Ejercicio 6: Escriba un programa que:

1. Permita que el usuario ingrese cuatro números, los almacene una tupla t.
2. Genere una tupla s la cual se obtiene sumando a cada elemento de t un valor ingresado por el usuario.
3. Genere una tupla r la cual se obtiene restando a cada elemento de t un valor ingresado por el usuario.
4. Imprima: con leyendas adecuadas la tupla t, s y r.

Ejercicio 7: Defina una tupla y muestre:

1. ¿Cómo se accede a un elemento de la tupla?
2. ¿Qué sucede si se intenta acceder a una posición inexistente de la tupla?
3. ¿Cómo se calcula la longitud de una tupla?

Ejercicio 8: Construya un programa que permita que el usuario ingrese una dupla y luego desempaquete la tupla en dos variables a y b. Luego el programa debe imprimir las variables a y b.

Ejercicio 9: Escriba un programa que permite que el usuario ingrese dos valores en las variables a y b y luego empaquete dichos valores en una tupla. Finalmente, el programa debe imprimir la tupla resultado.

Ejercicio 10: Escriba un programa que permite que el usuario ingrese un número a y una tupla t. Luego el programa debe imprimir True si el número a está en t y False en otro caso.

Ejercicio 11: Escriba un programa que permita que el usuario ingrese un número a y una tupla t. Luego el programa debe imprimir por pantalla la posición del número a en la tupla t. En caso de que el número a no se encuentre en t el programa debe imprimir -1.

Ejercicio 12: Realice las siguientes actividades:

1. Explique el concepto de rodaja.
2. Explique el concepto de zancada.
3. Por cada concepto explicado de ejemplos.

Ejercicio 13: Escriba un programa que permita que el usuario ingrese un número a y una tupla t. Luego el programa debe mostrar por pantalla la cantidad de veces que aparece el número a en la tupla t.

Ejercicio 14: Escriba un programa que permita que el usuario ingrese una tupla t y un elemento e. El programa debe informar si e está en la tupla t.

Ejercicio 15: Escriba un programa que permita que el usuario ingrese una tupla t y un elemento e. El programa debe informar si e no está en t.

Ejercicio 16: Escriba un programa que permita que el usuario ingrese dos tuplas t y r. El programa debe imprimir por pantalla la concatenación de t y r.

Ejercicio 17: Escriba un programa que:

1. Permita que el usuario ingrese una tupla t de cinco números.
2. Sume los números pares.
3. Sume los números impares.

3.2.3. Listas

Una lista es una secuencia de cero o más referencias a objetos. Las listas poseen las siguientes características:

- Soportan las mismas operaciones de rebanadas (slicing) y zancadas (striding) que los strings y las tuplas.
- A diferencia de las tuplas y los strings las listas son mutables.
- Es posible insertar, reemplazar y borrar rebanadas de listas.

- El constructor de listas se llama *list()*. Cuando se invoca sin argumentos retorna como resultado la lista vacía. Cuando el argumento es otra lista retorna una copia superficial del argumento. Con cualquier otro tipo de argumento lo intenta convertir a listas.
- Una lista vacía se puede crear con corchetes que abren y corchetes que cierran [].
- Una lista también se puede crear escribiendo sus elementos separados por comas: [1,10,30,"hola"].
- Las listas son heterogéneas.
- Las listas se pueden comparar con los operadores relacionales tradicionales: (<, <=, ==, !=, >=,>). Las comparaciones se aplican ítem por ítem y recursivamente para los ítems anidados tales como tuplas y listas.
- Las listas soportan operaciones de mebresía con *in* y *not in*.
- Las listas se pueden concatenar + (+=) y replicar con * (*=)
- A las listas se les puede calcular la longitud con *len()* y borrar elementos con *del*.

Operaciones con Tuplas

L[-6]	L[-5]	L[-4]	L[-3]	L[-2]	L[-1]
-17.5	'kilo'	49	'V'	['ram', 5, 'echo']	7
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k'
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] =='echo'
L[4][2][1] == L[4][2][-3] == c
L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

Si bien es posible usar rebanadas (slice) para acceder a los elementos de una lista en algunas situaciones se desea tomar uno o dos ítems. Esta tarea puede ser llevada a cabo utilizando el operador de desempaquetado. Cualquier iterable (listas, tuplas, etc) puede ser desempaquetado utilizando el operador de desempaquetado (*). Cuando este operador se usa con dos o más variables sobre el lado izquierdo de una asignación la variable que está precedida por * se inicializa con el sobrante del iterable mientras que las otras se inicializan con el valor correspondiente.

Operaciones con Tuplas

```
>>> primero, *resto = [9, 2, -4, 8, 7]
>>> primero, resto
(9, [2, -4, 8, 7])
>>> primero, *medio, último = """ General San Martín
                          Manuel Belgrano """.split()
>>> primero, medio, último
('General', ['San','Martín','Manuel'],'Belgrano')
>>> *carpetas, ejecutable =
            "/usr/local/bin/gvim".split("/")
>>> carpetas, ejecutable
(['', 'usr', 'local', 'bin'], 'gvim')
```

La tabla 3.1 describe algunas métodos que se pueden utilizar con las listas.

Comentario Importante

Los métodos antes mencionados no son los únicos el tipo lista tiene una gran cantidad de métodos y librerías.

l.append(x)	Agrega el ítem x al final de la lista l.
l.count(x)	Retorna como resultado la cantidad de veces que x aparece en la lista.
l.extend(m)	Agrega todos los ítems del iterable m al final de la lista. El operador += hace la misma tarea.
l.index(x,com,fin)	Retorna el índice donde se encuentra la primer ocurrencia de x comenzando desde la izquierda. Cuando se especifican com y fin la búsqueda se realiza en la rebanada correspondiente. Si x no se encuentra en la lista o rebanada según corresponda el método retorna una excepción ValueError.
l.insert(i,x)	Inserta en la lista l en la posición i el elemento x. i debe ser entero.
l.pop()	Retorna y elimina el elemento de más a la derecha de l.
l.pop(i)	Retorna y elimina el elemento de la posición i de l.
l.remove(x)	Remueve la primer ocurrencia de x (de izquierda a derecha) de l. Si x no se encuentra en l dispara una excepción ValueError.
l.reverse()	Invierte la lista l.

Cuadro 3.1: Métodos de Listas

3.2.4. Ejercicios

Nota: Asuma una cantidad específica de elementos cuando el ejercicio no lo especifique.

Ejercicio 1: Dada la siguiente lista l=[10,"hola",2.5,20,"que",3.5,30,"tal",4.5] se pide recuperar:

1. el 30
2. "hola"
3. 10,"hola",2.5
4. Los strings

5. Los flotantes
6. Los enteros

Ejercicio 2: Realice las siguientes actividades:

1. Defina una lista l de tres números donde cada número es 0.
2. Defina una lista de un único elemento.
3. Defina una lista con n 0s.

Ejercicio 3: Defina las listas l0 y l1 cada una con dos elementos numéricos y luego construya la lista r cuyos elementos son la suma de los elementos de l0 y l1. Ejemplo: Si l0=[10,20] y l1=[8,20] la tupla r tiene que contener r=[18,40].

Ejercicio 4: Escriba un ejemplo que muestre que las listas son mutables.

Ejercicio 5: Escriba un programa que dada una lista t con 5 elementos y un número n produzca como resultado una nueva lista con todos los elementos de la lista t multiplicados por el número n.

Ejercicio 6: Escriba un programa que almacene el valor de tres variables ingresadas por el usuario en una lista.

Ejercicio 7: Escriba un programa que:

1. Permita que el usuario ingrese cuatro números, los almacene una lista l.
2. Genere una lista s la cual se obtiene sumando a cada elemento de l un valor ingresado por el usuario.
3. Genere una lista r la cual se obtiene restando a cada elemento de l un valor ingresado por el usuario.
4. Imprima: con leyendas adecuadas la tupla l, s y r.

Ejercicio 8: Cree una lista y muestre:

1. El acceso a un elemento de la lista.

2. Qué sucede si se intenta acceder a una posición inexistente de la lista.
3. Cómo se calcula la longitud de una lista.

Ejercicio 9: Construya un programa que permita que el usuario ingrese una lista de dos elementos y luego desempaquete la lista en dos variables a y b. Luego el programa debe imprimir las variables a y b.

Ejercicio 10: Escriba un programa que permita que el usuario ingrese dos valores en las variables a y b y luego empaquete dichos valores en una lista. Luego el programa debe imprimir la tupla resultado.

Ejercicio 11: Escriba un programa que permita que el usuario ingrese un número a y una lista l. Luego el programa debe imprimir True si el número a está en l y False en otro caso.

Ejercicio 12: Escriba un programa que permita que el usuario ingrese un número a y una lista l. Luego el programa debe imprimir por pantalla la posición del número a en la lista l. En caso de que el número a no se encuentre en l el programa debe imprimir -1.

Ejercicio 13: Realice las siguientes actividades:

1. Explique el concepto de rodaja.
2. Explique el concepto de zancada.
3. Por cada concepto explicado de ejemplos.

Ejercicio 14: Escriba un programa que permita que el usuario ingrese un número a y una lista l. Luego el programa debe mostrar por pantalla la cantidad de veces que aparece el número a en la lista l.

Ejercicio 15: Dada la lista l=[34, 3.2, ”Juan”, “Pedro”,-2] se pide:

1. Agregue al final de l un string ingresado por el usuario.
2. Solicite al usuario un elemento y cuente la cantidad de veces que aparece dicho elemento en l.

3. Pida al usuario una lista s e incorpórela al final de l.
4. Invierta la lista l.

Ejercicio 16: Construya un programa que:

1. Permita que el usuario ingrese una lista l de números enteros l.
2. Ordene la lista
3. Almacene en la variable mayor el mayor elemento de la lista
4. Almacene en la variable menor el menor elemento de la lista.
5. Imprima por pantalla la lista l y el elemento mayor y el elemento menor.

Ejercicio 17: Escriba un programa que:

1. Permita que el usuario ingrese una lista l.
2. Pida al usuario en elemento e.
3. Pida al usuario una posición p válida.
4. Inserte en la lista l el elemento e en la posición p.

3.2.5. Tipos de datos Conjuntos

Un conjunto es un tipo de dato colección que soporta el operador de membresía *in*, la función de tamaño *len()* y es iterable. Además, los tipos conjuntos proveen un método *set.isdisjoint()* y permite realizar comparaciones. En el caso de == y != se mantiene el significado, los otros operadores relacionales realizan las comparaciones de *subconjunto* y *superconjunto*.
Python provee dos tipos de conjuntos el tipo *set* que es mutable y el tipo *frozenset* que es inmutable. Cuando en un conjunto se recorre los ítems son devueltos en un orden arbitrario. Los tipos de elementos que se pueden insertar en un conjunto tienen que cumplir ciertas características cuya descripción está fuera del alcance de este curso. No obstante, los tipos: *int*, *float*, *str*, *frozenset* y *tuple* cumplen con esas características y por consiguiente se pueden insertar en el conjunto. No pasa lo mismo con los tipos: *list*, *dict* y *set*.

3.2.6. Conjunto

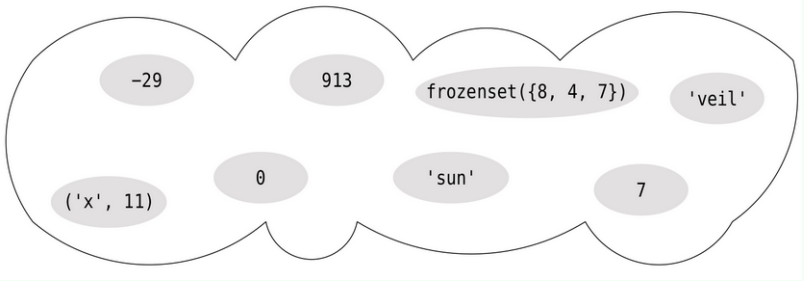
Un conjunto es una colección desordenada de referencias a objetos admisibles por el tipo (hasheable).

Los conjuntos poseen las siguientes características:

- Son mutables.
- Son desordenados y no tienen la noción de índice por consiguiente no se pueden aplicar rebanadas y zancadas.
- El constructor del tipo se llama *set()*. Cuando se invoca sin argumentos se crea el conjunto vacío. Con un argumento realiza una copia superficial del mismo.
- Los conjuntos no vacíos se pueden crear sin utilizar el constructor del tipo.
- Un conjunto con uno o más ítem se puede crear listando los ítems separados por comas y encerrados con llaves.
- Se puede usar *in* y *not in* para chequear membresía.
- Se puede usar la función *len()* con lo cual el resultado que se obtiene es la cardinalidad del conjunto.

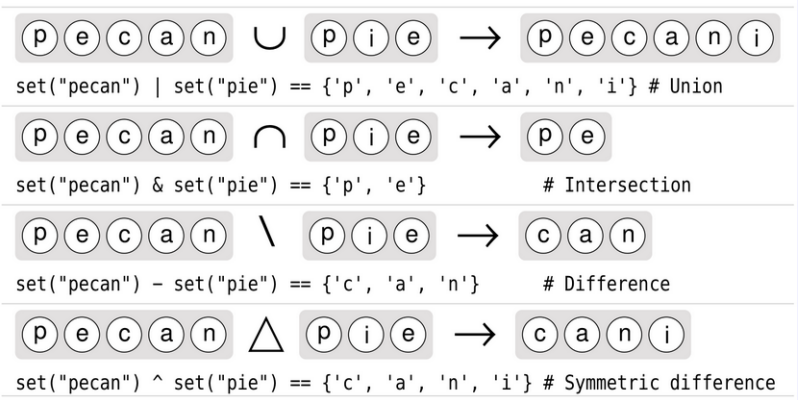
Ejemplo de Conjunto

```
S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}
```



Operaciones sobre Conjuntos

Con los conjuntos también se pueden realizar las operaciones de: *Unión*, *Intersección*, *Diferencia* y *Diferencia Simétrica* tal como se indica en la figura que se muestra a continuación.



Además de las operaciones mencionadas con anterioridad en la tabla 3.2 se describen otros métodos interesantes de conjuntos.

3.2.7. Conjuntos Congelados

Un Conjunto Congelado (*Frozen Set*) es un conjunto que una vez creado no se puede modificar.

Los conjuntos congelados tienen las siguientes características:

- Los conjuntos congelados puede solamente ser creados por medio del uso del constructor del tipo *frozenset()*.
- Cuando el constructor se invoca sin argumentos crea un conjunto congelado vacío.
- Cuando se invoca con un *frozenset* como argumento crea una copia superficial del argumento. Cuando se invoca con otro tipo intenta realizar una conversión a *frozenset* y luego realiza una copia superficial. El constructor del tipo acepta un único argumento.
- Son inmutables.

- Si un operador binario se usa con un *frozenset* el tipo de dato del resultado es el mismo tipo de dato del operando de más a la izquierda. De este modo si f es un frozenset y s es un set f & s producirá un frozenset y s & f producirá un set.
- En el caso de == y != el orden de los operandos no importa dado que el resultado es booleano.

3.2.8. Ejercicios

Nota: Asuma una cantidad específica de elementos cuando el ejercicio no lo especifique.

Ejercicio 1: Escriba un programa que:

1. Defina el conjunto universal el cual contiene las provincias de Argentina.
2. Pida al usuario una provincia.
3. Calcule el complemento del conjunto single que contiene la provincia ingresada por el usuario.

Ejercicio 2: Escriba un programa que:

1. Defina el conjunto universal U el cual está compuesto por los números del 1 al 20.
2. Pida al usuario dos conjuntos A y B.
3. Calcule la unión de A y B.
4. Calcule la intersección de A y B.
5. Calcule el complemento de las unión e intersección de A y B.

Ejercicio 3: Realice las siguientes actividades en caso de que sea posible:

1. Pase un conjunto a un diccionario.
2. Pase un conjunto a un conjunto congelado.
3. Pase un conjunto a una lista.

4. Ordene un conjunto.
5. Pase un conjunto a una tupla.

Ejercicio 4: Escriba un programa que a partir de dos conjuntos de números enteros ingresados por el usuario cree el conjunto universal el cual está formado por todos los números que pertenecen a los conjuntos usados en el programa.

Ejercicio 5: Escriba un programa que:

1. Ingrese un conjunto.
2. Imprima por pantalla: la cardinalidad, el mínimo y el máximo.

Ejercicio 6: Escriba un programa que permita que el usuario ingrese un conjunto y un valor. El programa debe informar si valor pertenece a conjunto.

Ejercicio 7: Escriba un programa que permita que el usuario ingrese un conjunto y tres valores. El programa debe incorporar esos valores al conjunto. Luego imprimir el conjunto resultado por pantalla.

Ejercicio 8: Se puede incorporar a un conjunto los elementos de una lista sin utilizar iteraciones.

Ejercicio 9: Escriba un programa que permita que el usuario ingrese un conjunto c y un valor v y si v está en c lo elimine de c. Luego imprima c.

Ejercicio 10: Escriba un programa que:

1. Permita que el usuario ingrese un conjunto A.
2. Permita que el usuario ingrese un conjunto B.
3. Informe si A es un subconjunto de B o B es un subconjunto de A.

3.3. Tipo de Dato Mapeo

Un tipo de mapeo es uno que soporta el operador de *membresía* (*in*) y la *función de tamaño* (*len()*). Los mapeos son *colecciones*

de pares clave-valor. Cuando se intenta recorrer un mapeo los elementos se van obteniendo de forma aleatoria. Python 3.0 provee dos tipos de mapeo el *Dict* (*Diccionario*) y el *collection.defaultdict* (*Diccionario por Defecto*). Python 3.1 incorpora el tipo *collections.OrderedDict* (*Diccionario Ordenado*) es decir un diccionario que tiene las mismas propiedades que un *dict* pero almacena sus ítems en el orden de inserción. Los objetos que pueden ser utilizados como claves en el diccionario son los tipos de datos inmutables (en realidad puede ir un objeto hasheable pero su explicación está fuera de los alcances de este curso.) tales como: *float*, *frozenset*, *int*, *str* y *tuple* mientras que los tipos mutables no pueden ser utilizados como clave. Como valor pueden ir objetos de cualquier tipo. Los tipos diccionarios pueden ser comparados usando los operadores de comparación estándar (== y !=) las comparaciones se aplican ítem por ítem y recursivamente para los ítems anidados tales como tuplas o diccionarios dentro de diccionarios. Los operadores de comparación restantes no son soportados por el tipo mapeo.

3.3.1. Diccionarios

Un diccionario es una colección de cero o más pares clave-valor cuyas claves son referencias a objetos inmutables y los valores son referencias a objetos de cualquier tipo.

Los diccionarios poseen las siguientes características:

- Los diccionarios son mutables.
- Los diccionarios están desordenados con lo cual no tienen la noción de índice y, por lo tanto, no es posible realizar rebanadas ni zancadas.
- El constructor del tipo se denomina *dict()*.
    - Cuando el constructor no recibe argumentos retorna un diccionario vacío.

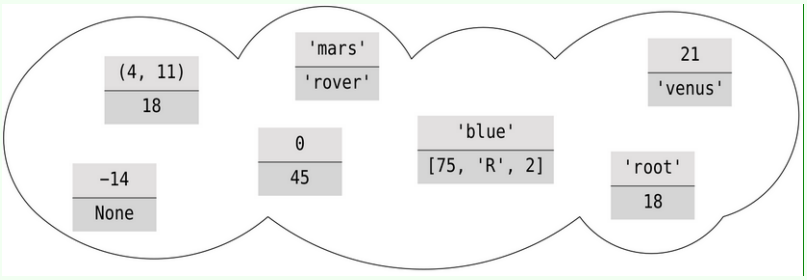
  - Cuando el constructor recibe un argumento retorna un diccionario basado en este.
  - Si el constructor recibe un diccionario como argumento retornará una copia superficial del argumento.
  - También se puede usar una secuencia como argumento del constructor compuesta por pares, en este caso la primera componente de cada par es la clave y la segunda componente es el valor.

- No puede haber claves repetidas.
- Si se quiere incorporar un par clave-valor con una clave que ya existe entonces el efecto producido es cambiar el valor asociado dicha clave.
- Los corchetes se usan para el acceso a los valores así d[20] retorna como resultado el valor asociado a la clave 20 del diccionario d. Si 20 no es una clave se produce una excepción: *KeyError*.
- d[clave]=valor incorpora un par clave: valor a d siempre que clave no sea una clave existente de d. Caso contrario, cambia el valor asociado a clave.
- del d[clave] elimina el ítem clave: valor de d siempre que clave exista en d caso contrario dispara una excepción: *KeyError*.
- Los diccionarios también se pueden crear usando {}. En este caso se crea el diccionario vacío. Cuando se quiere crear un diccionario con elementos usando ésta opción los mismos tienen que ser pares clave:valor separados por coma.
- Se puede usar *len()* con diccionarios.
- Se puede probar membresía con las claves con *in* y *not in*.

Ejemplo de Diccionarios

```
d1 = dict({"id": 1948, "nombre": "Lavadora", \
           "Tam": 3})
d2 = dict(id=1948, nombre="Lavadora", tam=3)
d3 = dict([("id",1948),("nombre","Lavadora"), \
           ("Tam",3)])
d4 = {"id": 1948, "nombre": "Lavadora", "Tam": 3}
```

Ejemplo de Diccionarios




```
d = {"root":18,"blue":[75,"R",2],21:"venus", \
     -14:None,"mars":"rover",(4,11):18,0:45}
```

En la tabla 3.3 se describen algunas operaciones utilizadas cuando se trabaja con diccionarios.

Vistas de Diccionarios

Los métodos *dict.items()*, *dict.keys()* y *dict.values()* retornan una vista de un diccionario. Una vista es un objeto iterable de solo lectura que mantiene ítems de un diccionario, claves o valores dependiendo del tipo de vista que se haya utilizado. En general, las vistas se pueden tratar como iterables. Sin embargo, las mismas pueden hacer dos cosas diferentes respecto de un iterable:

- Si el diccionario al cual la vista se refiere cambia la vista también cambia.
- Las claves y los ítems soportan operaciones como las de un conjunto. De esta forma si v y w son vistas las operaciones soportadas son:

v & x	Intersección
v | x	Unión
v - x	Diferencia
v x̂	Diferencia Simétrica

Ejemplo de Vistas de Diccionarios

```
#d={'A':3,'B':3,'C':3,'D':3}
d={}.fromkeys("ABCD",3)
s=set("ACX") #s={'A','C','X'}
#concordancia={'A','C'}
matches=d.keys() & s
```

3.3.2. Diccionarios por Defecto

Los diccionarios por defecto son diccionarios, es decir tienen los mismos operadores y métodos que proveen los diccionarios. La diferencia principal con los diccionarios es que permiten manejar las claves que no se encuentran en el diccionario. Si un diccionario no tiene un ítem con clave m, el código d[m] disparará una excepción *KeyError*. Pero si d es un diccionario por defecto, si una clave m está en el diccionario se retorna el valor correspondiente. No obstante, si m no está en el diccionario se crea un nuevo ítem cuya clave es m y cuyo valor es un valor definido por defecto. Dicho valor es el que se retorna como resultado. Cuando un diccionario por defecto se crea se puede pasar como parámetro al constructor de la clase una *función factoría*. Una función factoría es una función que cuando se invoca retorna un objeto de un tipo particular. Todos los tipos primitivos de Python pueden ser usados como funciones factorías. Por ejemplo *str* tiene el constructor de la clase *str()*. Este tipo de funciones puede ser utilizada para crear valores por defecto.

Diccionario por Defecto

```
d=collections.defaultdict(int)

defaultdict(, {})
d["Pedro"]
0
d
defaultdict(, {'Pedro': 0})
```

3.3.3. Diccionarios Ordenados

Los diccionarios ordenados *collections.OrderedDict* pueden ser usados como reemplazo directo de los diccionarios desordenados dado que proveen la misma API. La diferencia entre los dos es que los diccionarios ordenados

almacenan sus ítems en el orden en el cual ellos se insertaron.

Diccionario Ordenado

```
d=collections.OrderedDict\
        ([('z',-4),('e',19),('k',7)])
list(d.keys())
['z','e','k']

tasks = collections.OrderedDict()
tasks[8031]="Backup"
tasks[4027]="Scan Email"
tasks[5733]="Build System"
list(tasks.keys())
[8031, 4027, 5733]
```

Comentarios Importante

Otra buena característica de los diccionarios por defecto es que si se cambia el valor de un ítem, es decir si se inserta un ítem con una clave que existe, el orden no se cambia.

3.3.4. Ejercicios

Nota: Asuma una cantidad específica de elementos cuando el ejercicio no lo especifique.

Ejercicio 1: Muestre diferentes formas de crear un diccionario vacío.

Ejercicio 2: Dado el siguiente diccionario d={1:"Daniel", 2:"Germán", 3:"Analía", 4:"José", 5:"Gabriel"} se pide:

1. Devuelva el valor asociado a la clave 3.
2. Calcule la longitud del diccionario.

3. Devuelva las claves del diccionario.
4. Devuelva los valores del diccionario.

Ejercicio 3: Escriba un programa que permita que el usuario ingrese una lista de duplas ln. Cada dupla tiene como primer componente un nombre y como segunda componente un número. Luego cree un diccionario cuyas claves son los nombres en ln y cuyo valor sean enteros.

Ejercicio 4: Escriba un programa que:

1. Permita que el usuario ingrese un diccionario d.
2. Permita que el usuario ingrese un elemento e.
3. Cuente cuántas veces aparece e en los valores de d.

Ejercicio 5: Escriba un programa que:

1. Permita que el usuario ingrese un diccionario d.
2. Permita que el usuario ingrese una clave c.
3. Imprima por pantalla si la clave c está en el diccionario d.

Ejercicio 6: Escriba un programa que permita que el usuario ingrese un diccionario. El programa debe imprimir una lista de tuplas donde en cada tupla tiene como primer elemento la clave y como segundo elemento el valor asociado a la clave.

Ejercicio 7: Escriba un programa que permita almacenar en un diccionario tres personas. Por cada persona se registra: el dni, nombre, domicilio y edad. Use como clave para el diccionario el dni.

Ejercicio 8: Escriba un ejemplo que muestre que los diccionarios son mutables.

Ejercicio 9: Defina un diccionario y muestre:

1. Cómo se accede a un elemento de un diccionario

2. Qué sucede si se intenta acceder al diccionario con una clave inexistente.
3. ¿Cómo se calcula la longitud de un diccionario?

Ejercicio 10: Escriba un programa que permite que el usuario ingrese dos valores en las variables a y b y luego determina si dichos valores se encuentran almacenados como valor en el diccionario d. El diccionario d es ingresado por el usuario.

Ejercicio 11: Escriba un programa que permita que el usuario ingrese un número a y una tupla t. Luego el programa debe insertar en el diccionario d el par a,t.

Ejercicio 12: Se pueden sacar rodajas en los diccionarios.

Ejercicio 13: Se pueden hacer zancadas en los diccionarios.

Ejercicio 14: Escriba un programa que permita que el usuario ingrese dos diccionarios a y b y a partir de ellos cree las siguientes vistas:

1. u el cual contiene la unión de la vista de claves de a con la vista de claves de b.
2. i el cual contiene la intersección de la vista de claves de a con la vista de claves de b.
3. d la cual contiene la diferencia entre la vista de claves de a con la vista de claves de b.
4. ds la cual contiene la diferencia simétrica de la visa de claves de a con la vista de claves de b.

3.4. Iteración y Copia de Colecciones

Una vez que se tienen todos los elementos en colecciones es natural necesitar recorrer y realizar operaciones sobre cada ítem de la colección. En esta sección se explica el uso de iteradores los cuales son utilizados para

acceder a los elementos de una colección. El recorrido de una colección se realiza con las sentencias de iteración las cuales son explicadas en la próxima unidad.

3.4.1. Iteradores, Operaciones Iterables y Funciones

Un iterable es un tipo de dato que puede retornar cada uno de sus ítems uno a la vez. Cualquier objeto que tenga un método __item__(), o cualquier secuencia, es decir un objeto que tiene el método __getitem()__ que tome un entero que comienza en 0, es un iterable y provee un iterador. Un iterador es un objeto que provee el método __next__() el cual retorna un ítem a la vez y dispara una excepción *StopIteration* cuando no hay más ítems. La tabla 3.4 muestra los operadores y funciones que comúnmente pueden ser usados con iterables. El orden en el cual los ítems se retornan dependen del iterable subyacente. En el caso de las listas, tuplas y strings los ítems se retornan de manera secuencial comenzando por el ítem que está en la posición 0. Pero algunos iteradores retornan ítems en orden arbitrario como es el caso de los iteradores de diccionarios y conjuntos. El constructor de *iterador iter()* tiene dos comportamientos completamente diferentes:

- Cuando recibe como parámetro un tipo colección o una secuencia retorna un iterador al objeto que recibe como parámetro o dispara una excepción *TypeError* si el objeto no es iterable. Esto por lo general se utiliza cuando se definen estructuras de datos personalizada.

- El segundo comportamiento aparece cuando el iterador recibe una función o un método y un *sentinela*. En este caso la función o método pasado como parámetro se invoca en cada iteración y retorna un valor cada vez que se invoca o dispara una excepción *StopIteration* si el valor de retorno es igual al *sentinela*.

Ejemplo de Iterador

```
>>>l=[20,30,40]
>>> i=iter(l)
>>> i
>>> next(i) #20
>>> i.__next__() #30

a = 0

def x():
 global a
 a = a + 2
 return a

iterator = iter(x, 10)
for item in iterator:
 print(item)
```

3.4.2. Ejercicios

Ejercicio 1: Muestre como se obtienen los elementos de una lista con un iterador.

Ejercicio 2: Muestre como se obtienen los elementos de una tupla con un iterador.

Ejercicio 3: Muestre como se pueden obtener las claves un diccionario con un iterador.

Ejercicio 3: Muestre como se pueden obtener las claves un diccionario con un conjunto.

3.5. Copia de Colecciones

Debido a que Python usa referencias a objetos cuando se usa el operador de asignación = en realidad no se está realizando una copia sino más bien lo que se copia es la referencia.

- Si lo que está sobre el lado derecho de una asignación es un literal se copia una referencia a dicho objeto que está almacenado en memoria.
- Si el operando sobre el lado derecho de la asignación es una referencia un objeto lo que se copia es la referencia y en consecuencia el objeto tendrá dos referencias que apuntan hacia él.

Ejemplos de Copias Superficiales

```
>>> canciones= ["Because","Boys","Carol"]
>>> beatles= canciones
>>> beatles,canciones
(['Because','Boys','Carol'], \
 ['Because','Boys','Carol'])
```

En el ejemplo anterior las dos referencias *canciones* y *beatles* es decir solo se ha copiado la dirección del objeto, una copia del objeto en sí no se ha llevado a cabo. De esta manera si por alguna razón una referencia se modifica el objeto dicha modificación tendrá efecto en las dos referencias.

Ejemplo de Modificación en Copia Superficial

```
>>> beatles[2]="Cayenne"
>>> beatles,canciones
(['Because','Boys','Cayenne'], \
 ['Because','Boys','Cayenne'])
```

Cuando se necesitan copias independientes de las colecciones se pueden utilizar funciones de librería como *copy.deepcopy(c)* o en todo caso la debe

implementar el programador.

Comentario Importante

Se pueden copiar colecciones de diferentes maneras. Lo que el programador debe tener presente es "qué es lo que necesita?"si son copias superficiales, es decir tener referencias a un mismo objeto u objetos que comparten referencias con otros objetos, o crear objetos totalmente independientes entre sí. Es importante tener presente que tener alias (es decir varias referencias a un mismo objeto) no es aconsejable porque accidentalmente se pueden realizar cambios por una referencia y en consecuencia la otra también cambiará. Esta característica puede hacer muy complicado depurar el programa.

3.5.1. Ejercicios

Ejercicio 1: Dada la siguiente estructura de datos:

```
l = [[1, 2], 10, 20, [30, 49]]
```

Se pide:

- Haga una copia superficial.
- Haga una copia profunda.

Ejercicio 2: Dada la siguiente estructura de datos:

```
d = {1: [1, 2], "A": 10, "B:: 20, (1, 2): [30, 49]}
```

Se pide:

- Haga una copia superficial.
- Haga una copia profunda.

Ejercicio 3: ¿Se puede hacer una copia superficial y una copia profunda de un string o de una tupla? ¿Tiene utilidad?

Ejercicio 4: Dada la siguiente estructura de datos:

$$l = \{[1, 2], \{10, 20\}, [30, 49]\}$$

Se pide:

- Haga una copia superficial.
- Haga una copia profunda.

s.add(x)	Si x no se encuentra en s agrega x a s.
s.clear()	Elimina todos los elementos del conjunto s.
s.copy()	Retorna una copia superficial del conjunto s.
s.difference(t) o s-t	Retorna un nuevo conjunto que tiene todos los elementos que están en s y que no están en t.
s.diference_update(t) o s-=t	Elimina del conjunto s los elementos que se encuentran en el conjunto t.
s.discard(x)	Elimina el elemento x del conjunto s.
s.intersection(t) o s & t	Retorna un nuevo conjunto que contiene los elementos que están en s y t.
s.intersection_update(t) o s&=t	El conjunto s contienen la intersección de sí mismo con t.
s.isdisjoint(t)	Retorna True si s y t no tienen elementos en común.
s.issubset(t) o s<=t	Retorna True si s es igual o un subconjunto de t. Use s <t para verificar que s es un subconjunto propio de t.
s.superset(t) o s >=t	Retorna True si s es igual o un super conjunto de t. Use s>t para verificar si s es un subconjunto propio de t.
s.pop()	Retorna y elimina un item de forma aleatoria de s. Dispara una excepción KeyError si s está vacío.
s.remove(x)	Elimina el ítem x de s. Dispara una excepción KeyError si x no está en s.
s.symmetric_difference(t) o s ^ t	Retorna un nuevo conjunto que tiene todos los elementos que están en s y todos los que están en t excluyendo los ítems que están en ambos conjuntos.
s.symmetric_difference _update(t) o s ^ =t	Hace que s contenga la diferencia simétrica de sí mismo y t.
s.union(t)	Retorna un nuevo conjunto que tiene todos los ítems de s y todos los ítems de t que no están en s.
s.update(t) o s |=t	Agrega todo ítem que está en t y que no está en s a s.

Cuadro 3.2: Operaciones con Conjuntos

d.clear()	Elimina todos los ítems del diccionario d.
d.copy()	Retorna una copia superficial del diccionario d.
d.fromkeys(s,v)	Retorna un diccionario cuyas claves son los ítems que están en la secuencia s y cuyo valor es *None* o v en caso de que sea especificado.
d.get(c)	Retorna el valor asociado a la clave c o None si c no está en d.
d.get(c,v)	Retorna el valor asociado a la clave c o v si c no está en d.
d.items()	Retorna una vista de todos los pares clave:valor en d.
d.keys()	Retorna una vista de todas las claves en d.
d.pop(c)	Retorna el valor asociado a c y elimina el ítem cuya clave es c o dispara una excepción *KeyError* si c no está en d.
d.pop(c,v)	Retorna el valor asociado a c y elimina el ítem cuya clave es c o dispara retorna v si c no está en d.
d.popitem()	Retorna y elimina un par clave:valor que se encuentra en d o dispara una excepción *KeyError* si d es vacío.
d.setdefault(c,v)	Realiza la misma tarea que el método *dict.get()* excepto que si la clave c no está en el diccionario d se inserta un nuevo ítem en el diccionario con clave c y valor v o *None* si v no se especifica.
d.update(a)	Incorpora todo par clave:valor de a que no está en d. Para toda clave que está en d y a reemplaza el valor de d por el valor de a. " a"puede ser un iterable de pareces clave:valor.
d.values()	Retorna una vista de todos los valores en d.

Cuadro 3.3: Operaciones de Diccionarios

s+t	Retorna una secuencia que es la concatenación de la secuencia s con la secuencia t.
s * n	Retorna una secuencia que tiene n (int) concatenaciones de la secuencia s.
x in i	Retorna True si x está en el iterable i. Utilice not in para el test inverso.
all(i)	Retorna True si todo ítem en el iterable i evalúa a True.
any(i)	Retorna True si existe un ítem en el iterable i que evalúa a True.
enumerate(i,com)	Normalmente se utiliza en loops for...in para proporcionar una secuencia de tuplas (índice, item) con el índice que comienza en 0 o comienzo.
len(x)	Retorna la longitud de x. Si x es una colección retorna el número de ítems. Si x es un string retorna el número de caracteres.
max(i,key)	Retorna el ítem más grande del en el iterable i o el ítem con key(i) más grande si se provee la función key.
min(i,key)	Retorna el ítem más pequeño del iterable i o el ítem con key(i) más chico si se provee la función key.
range(comienzo, parada,paso)	Retorna un iterador entero. Con un argumento parada, el iterador va desde 0 a parada-1. Con dos argumentos (comienzo, parada) el iterador va desde comienzo a parada-1. Con tres argumentos el iterador va de comienzo a parada saltando de a paso.
reversed(i)	Retorna un iterador que retorna los ítems desde el iterador i en orden inverso.
sorted(i,key,reverse)	Retorna una lista ordenada de ítems usando el iterador i. Key se utiliza para proporcionar un ordenamiento DSU (Decore,Sort,Underscore). Si reverse is True la clasificación se realiza en orden inverso.
sum(i,start)	Retorna la suma de los ítems en el iterable i más start (el cual por defecto es 0), i no contiene strings.

Cuadro 3.4: Operadores y Funciones de Iterable