Clase Nº 10

Tuplas, Conjuntos y Diccionarios

Tuplas

- Las tuplas son similares a las listas, ya que se accede a ellas a través de un subíndice.
- Para crear una tupla se encierran sus elementos entre paréntesis en lugar de corchetes:

```
salsas = ("tártara", "criolla", "chimichurri")
```

Tuplas

- Si se escribe una secuencia de elementos sin paréntesis pero separados por comas, también se crea una tupla.

```
salsas = "tártara", "criolla", "chimichurri"
```


Tuplas

- Una tupla vacía se define con un par de paréntesis, sin nada en su interior.

```
aderezos = ( )
```

Tuplas

- Para crear una tupla con un solo elemento es necesario escribir una coma luego de éste.

```
misalsa = "barbacoa",
```

- Los paréntesis también son opcionales en este caso.

Tuplas

- La principal diferencia entre listas y tuplas es que **las tuplas *son inmutables***, y por lo tanto no pueden ser modificadas.

Tuplas

- Debido a que son inmutables, las tuplas carecen del método *append( )*.
- La única forma de agregar elementos es a través del operador de concatenación, creando una nueva tupla:

```
aderezos = aderezos + ("mayonesa",)
```


Tuplas

- El operador de repetición * también puede ser usado con tuplas.

```
binario = (0, 1) * 3
print(binario)  # (0, 1, 0, 1, 0, 1)
```

Tuplas

- En una misma tupla se pueden combinar distintos tipos de datos.

```
primavera = (21, "Septiembre")
```

- y también pueden ser anidadas.

```
alumno = (152408, "Luis Arce", "Lima 717", (1, "Mayo", 1998))
```


Tuplas

- Al igual que en las listas, en las tuplas se puede acceder a sus elementos mediante un subíndice (base 0), o más de uno si las tuplas están anidadas:

```
alumno = (152408, "Luis Arce", "Lima 717", (1, "Mayo", 1998))
print(alumno[1])      # Luis Arce
print(alumno[3][1])   # Mayo
```

Tuplas

- Tratar de modificar un elemento de una tupla mediante su subíndice provocará una excepción de tipo TypeError.

```
t = (1, 2, 3)
t[0] = 4
```

TypeError: 'tuple' object does not support item assignment

Tuplas

- Como todo iterable, las tuplas pueden ser recorridas mediante un ciclo *for*:

```
for dato in alumno:
    print(dato)
```

- y también pueden ser manipuladas mediante rebanadas:

```
print(alumno[ :2])  # (152408,"Luis Arce")
```

Funciones y métodos

- Las funciones **len()**, **max()**, **min()** y **sum()** operan con tuplas igual que lo hacen con listas.
- También actúan del mismo modo el operador **in** y los métodos **index** y **count**.
- Sin embargo, no están disponibles los métodos que trabajan *in situ*, como **append**, **remove**, **pop** o **sort**.

Conversión de tuplas

- Una lista puede ser convertida en tupla mediante la función ***tuple( )***.
- Una tupla puede ser convertida en lista con la función ***list( )***.

```
lista = [1, 2, 3]
tupla = tuple(lista)  # (1, 2, 3)
```

Aplicaciones

- Mediante el uso de tuplas es posible intercambiar el valor de dos variables sin necesidad de usar una variable auxiliar:

a, b = b, a

Aplicaciones

- Por lo general se prefiere utilizar tuplas cuando los elementos son heterogéneos, y listas cuando éstos son homogéneos.

fecha = (dia, mes, año)

- Aunque todos son números, su significado es heterogéneo.

Conjuntos

- Un *conjunto* es una colección de elementos sin orden ni duplicados.
- No entran dentro de la categoría *secuencia* porque carecen de orden interno.
- Se los suele utilizar cuando se desea eliminar elementos repetidos.

Conjuntos

- Para crear un conjunto se procede en forma similar a las listas, pero reemplazando los corchetes por llaves:

```
frutas = {"banana", "manzana", "naranja", "pera", "banana"}
print(frutas)   # {"banana", "manzana", "naranja", "pera"}
                #"banana" quedó sólo una vez
```

Conjuntos

- Un conjunto vacío se crea utilizando la función ***set( )***:

```
conjunto = set()
```

- Si se escriben dos llaves juntas se crea un *diccionario*.

Conjuntos

- Los conjuntos son **mutables**, es decir que *pueden ser modificados*.
- Sin embargo, los elementos que se agreguen a un conjunto deben ser de un tipo ***inmutable*** (números, strings o tuplas).

Conjuntos

- **Tratar de agregar un dato perteneciente a un tipo mutable (listas, diccionarios u otros conjuntos) provocará un error:**

```
>>> lenguajes = {["Python","Perl"], ["Java", "C++", "C#"]}
TypeError: unhashable type: 'list'
```


Operaciones con conjuntos

- **Para verificar si un elemento se encuentra dentro de un conjunto se utiliza el operador *in* (o *not in*):**

```
if "manzana" in frutas:
    print("Verde o roja?")
```

Operaciones con conjuntos

- Las operaciones habituales de conjuntos se realizan con los siguientes operadores:
  - | unión A U B
  - & intersección A ∩ B
  - – diferencia A – B
  - ^ diferencia simétrica A Δ B

Funciones

- Las funciones len(), max(), min() y sum() también operan con conjuntos.
- Por tratarse de iterables, los conjuntos pueden ser recorridos mediante un ciclo for:

```
conj = set(range(10))
for elem in conj:
    print(elem, end=" ")
```

Métodos

- El método add(<elem>) agrega un elemento al conjunto:

```
conjunto = {3, 4, 5}
conjunto.add(6)
print(conjunto)    # {3, 4, 5, 6}
```

- Equivale al *append* de las listas.

Métodos

- El método remove(<elem>) elimina un elemento identificado por su valor. Provoca una excepción *KeyError* si no está presente.

```
conjunto = {3, 4, 5}
conjunto.remove(4)
print(conjunto)    # {3, 5}
```

Métodos

- El método discard(<elem>) también elimina un elemento identificado por su valor. Es similar al anterior, pero no provoca una excepción si el elemento no se encuentra.

```
conjunto = {3, 4, 5}
conjunto.discard(4)
print(conjunto)    # {3, 5}
```


Métodos

- El método pop( ) elimina y devuelve un elemento del conjunto elegido al azar:

```
conjunto = {3, 4, 5}
x = conjunto.pop( )
print(conjunto)    # {?, ?}
```

- Provoca un *KeyError* si el conjunto está vacío.

Métodos

- El método clear( ) elimina todos los elementos del conjunto.

```
conjunto = {3, 4, 5}
conjunto.clear( )
print(conjunto)    # set( )
```


Ejemplo 1

Desarrollar un programa para simular la entrega de naipes a un jugador de póker, evitando la generación de naipes repetidos.

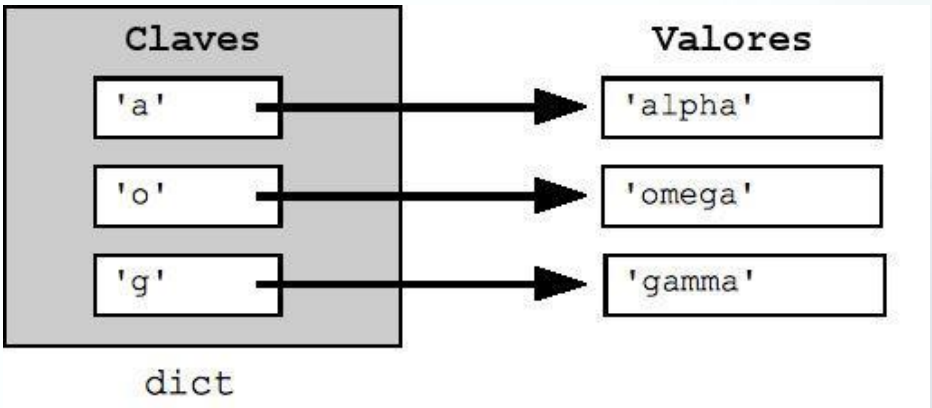
```
import random

simbolo = ("Trébol", "Pica", "Corazón", "Diamante")
mano = set( )
intentos = 0
while len(mano)<5:  # cinco cartas por jugador
    numero = random.randint(1, 13)  # 1 a 10 más J, Q, K
    palo = random.randint(0, 3)
    carta = (numero, simbolo[palo])
    mano.add(carta)
    intentos = intentos + 1
print(mano)
print("Intentos realizados:", intentos)
```


Diccionarios

- Son estructuras de datos que se utilizan para relacionar claves y valores.
- También se los conoce como *arreglos asociativos* (PHP), *tablas de hash* (Perl) o *hashmaps* (Java).

Diccionarios



Diccionarios

- **Cada elemento de un diccionario se representa mediante una dupla *clave:valor*.**
- **Se crean encerrando sus duplas entre llaves y separándolas por comas.**

```
edades = {"Dante":27, "Brenda":18, "Malena":23}
```

Diccionarios

- **Para acceder a sus elementos se utiliza la clave en lugar de un subíndice numérico.**

```
edades = {"Dante":27, "Brenda":18, "Malena":23}
print(edades["Brenda"])  # 18
```


Diccionarios

- **La clave de cada dupla debe pertenecer a un tipo *inmutable* (números, cadenas de caracteres, tuplas).**
- **Los valores asociados a cada clave pueden ser de *cualquier tipo*, incluyendo listas u otros diccionarios.**

```
colores = {"Rojo":[255,0,0], "Verde":[0,255,0], "Azul":[0,0,255]}
```

Diccionarios

- Los diccionarios no son secuencias, y por lo tanto no están ordenados.
- No se puede utilizar un subíndice para acceder a sus elementos.
- Funcionan como una lista a la que se accede mediante una clave.

Diccionarios

- Los diccionarios pueden definirse con un formato más claro y legible, colocando cada dupla debajo de la anterior.

```
colores = {
   "Rojo"   : [255,0,0],
   "Verde"  : [0,255,0],
   "Azul"   : [0,0,255]
}
```

Diccionarios

- Al igual que ocurre con las listas, también pueden crearse *diccionarios por comprensión*.

```
dic = {x:x**2 for x in range(1, 5)}
print(dic)  # {1: 1, 2: 4, 3: 9, 4: 16}
```

- En este caso se utilizan llaves en lugar de corchetes.

Diccionarios

- Las *rebanadas* no son aplicables a los diccionarios ya que carecen de orden interno.

- Las claves deben ser *únicas*; no se permiten claves duplicadas.

Diccionarios

- Asignar un valor a una clave reemplaza el valor existente o crea una clave nueva, dependiendo de si existía o no.

```
colores["Gris"] = [128,128,128]
```


Diccionarios

- Para recorrer un diccionario es posible utilizar la instrucción *for*. La variable del for recibe el valor de cada clave del diccionario.

```
for color in colores:
    print(color, "➔", colores[color])
```

Diccionarios

- No es posible acceder a una clave a través de su valor.
- Un mismo valor puede estar asociado a más de una clave.
- Tratar de acceder a un elemento con una clave inexistente provoca una excepción ***KeyError***.

Diccionarios

- Puede verificarse si una clave existe utilizando el operador ***in*** (o ***not in***).
- Tambien es posible utilizar el método **get( )**, que devuelve el valor asociado a una clave o ***None*** si la misma no se encuentra.

```
a = colores["Rojo"]          # Ambas líneas
b = colores.get("Rojo")      # son equivalentes
```

Métodos

- El método get( ) admite un segundo parámetro que será devuelto en lugar de *None* cuando la clave no esté presente.

```
a = colores.get("Cian", "No encontrado")
```


Métodos

- El método pop(<clave> [, <valor>]) busca <clave> dentro del diccionario y la elimina, devolviendo el valor asociado a esa clave. Si la clave no se encuentra devuelve <valor>. Si <valor> no se incluye se produce un ***KeyError.***

```
a = colores.pop("Verde")
print(a)  # [0, 255, 0]
```

Métodos

- El método keys( ) devuelve una secuencia con las claves presentes en el diccionario.

```
claves = list(colores.keys( ))
print(claves)   # ['Rojo', 'Verde', 'Azul']
```

- La función *list( )* es necesaria para convertir el objeto devuelto por el método.

Métodos

- El método values( ) devuelve una secuencia con los valores presentes en el diccionario.

```
valores = list(colores.values( ))
print(valores)
# [[255, 0, 0], [0, 255, 0], [0, 0, 255]]
```

- *list( )* cumple la misma tarea anterior.

Métodos

- El método **items( )** devuelve una secuencia de tuplas ***clave:valor***.

```
for color, RGB in colores.items( ):
    print(color, "➔", RGB)
```

- Esta secuencia puede ser convertida a lista mediante la función ***list( )***.

Método *fromkeys*

- **fromkeys(<secuencia> [, <valor>])** sirve para crear un diccionario a partir de una secuencia cualquiera (lista, cadena, tupla, rango...).
- Cada elemento de la secuencia se convierte en una clave del diccionario.
- El valor asociado será *None* u otro proporcionado por el programador.

Método *fromkeys*

- Este método no se aplica sobre una variable sino sobre la clase ***dict***.
- Devuelve el diccionario creado como valor de retorno.

Método *fromkeys*

```
dias = "Lunes", "Martes"  # Tupla
d1 = dict.fromkeys(dias)
print(d1)   # {'Lunes': None, 'Martes': None}

vocales = "aeiou" # String
d2 = dict.fromkeys(vocales, 0)
print(d2)   # {'a': 0, 'e': 0, 'i': 0, 'o': 0, 'u': 0}
```

Instrucción *del*

- Además del método pop, la instrucción **del** se utiliza para eliminar un elemento de un diccionario.

```
del colores["Rojo"]
```

- ...que también permite borrar el diccionario completo.

```
del colores
```


Ejemplo 2

Realizar un programa para ingresar una frase y mostrar un listado ordenado alfabéticamente con las palabras que contiene, eliminando las repetidas y añadiendo junto a cada una la cantidad de veces que se encontró.

```
frase = input("Ingrese una frase:\n")
listadepalabras = frase.split( )
dic = { }
for palabra in listadepalabras:
    if palabra not in dic:
        dic[palabra] = 1
    else:
        dic[palabra] = dic[palabra] + 1
listado = [ ]
for p in dic:
    listado.append("> "+p+": "+str(dic[p])+" veces")
listado.sort( )
for linea in listado:
    print(linea)
```


Ejemplo de ejecución:

```
Ingrese una frase:
buenos son los viejos amigos, pero solo los buenos llegan a viejos
> a: 1 veces
> amigos,: 1 veces
> buenos: 2 veces
> llegan: 1 veces
> los: 2 veces
> pero: 1 veces
> solo: 1 veces
> son: 1 veces
> viejos: 2 veces
```

```
dic = { }
for palabra in listadepalabras:
    if palabra not in dic:
        dic[palabra] = 1
    else:
        dic[palabra] = dic[palabra] + 1
```

El ciclo principal del programa puede ser *pythonizado* de la siguiente forma:

```
dic = { }
for palabra in listadepalabras:
    dic[palabra] = dic.get(palabra, 0) + 1
```


Aplicaciones

Algunas aplicaciones de los diccionarios son:

- **Agenda telefónica**
- **Control de stock**
- **Usuarios y contraseñas**
- **Listas de precios**

Ejercitación

- **Práctica 8: Completa**

Trabajo Práctico 8
Ejercitación por equipos

Tomar el número del grupo y calcular el resto de dividirlo por 3.

- **Resto 0: Ejercicios 2 y 6**
- **Resto 1: Ejercicios 3 y 11**
- **Resto 2: Ejercicios 4 y 9**